Query Caching and View Selection for XML Databases

Bhushan Mandhani   Dan Suciu

Dept of Computer Science
University of Washington
Seattle
USA
{bhushan,suciu}@cs.washington.edu

Abstract

In this paper, we propose a method for maintaining a semantic cache of materialized XPath views. The cached views include queries that have been previously asked, and additional selected views. The cache can be stored inside the database, or on a different host machine. We describe a notion of XPath query/view answerability, which allows us to reduce tree operations to string operations for matching a query/view pair. We show how to store and maintain the cached views in relational tables, so that cache lookup is very efficient. We also describe a technique for view selection, given a warm-up workload. Finally, we experimentally demonstrate the efficiency of our caching techniques, and performance gains obtained by employing such a cache.

1 Introduction

XML is increasingly being used in data intensive applications. Major database vendors are incorporating native XML support in the latest versions of their relational database products. Data meant for web services, and data exchange applications is often most conveniently stored directly as XML. In this scenario, the number and size of XML databases is rapidly increasing, and XML data becomes the focus of query evaluators and optimizers.

In a relational database system, the in-memory buffer cache is crucial for good performance. A similar buffer cache can, and is, employed in XML systems.

However, XML query processing presents a different set of challenges. Query execution on semistructured data is intrinsically harder to optimize. The buffer cache reduces the disk I/O cost, but not the computational cost. We propose maintaining a semantic cache of query results [DFJ+96]. It will address the computational cost and thus, complement the buffer cache. It is simple to have the semantic cache. In our scheme, it is maintained in three tables in the database (two purely relational, and one with an XML column). Further, the semantic cache can also be maintained on a different database system, on a remote host. Thus, unlike the page-based buffer cache, it can be employed in a distributed setting too.

We describe in this paper, a framework for maintaining and using a semantic cache of query results. The cached queries are basically materialized views, which can be used in query processing. Thus, at any moment, the semantic cache contains some views $\{V_1, \ldots, V_n\}$. When the system has to evaluate a new query Q, it inspects each view V in the cache and determines whether it is possible to answer Q from the result of V. In our setting, the views are XPath expressions, while the queries are either XPath or a restricted XQuery fragment (which we describe later). For now, we will take Q to be XPath, and describe the extension to XQuery, later. We say that view V answers query Q if there exists some other query C which, when executed on the result of V, gives the result of Q. We write this as C∘V = Q. We call C the Composing Query (CQ). When some cached view answers a posed query, we have a *hit*; otherwise we have a *miss*.

There are several applications for such a semantic cache. First, consider its use inside the XML database system. Suppose some query Q is answered by view V, with C being the CQ. Then Q is answered by executing C, which is simpler than Q, on the result of V, which is a much smaller XML fragment than the original data instance. This can result in a significant speedup, as we show in our experiments. The semantic cache can also be maintained at the application tier. Here, there will be additional savings for a hit, from not having to

connect to the backend database. For a heavily loaded backend server, these savings can be large. This kind of middle-tier caching has become popular for web applications using relational databases [LKM+02]. Finally, the semantic cache can also be employed in a setting like distributed XQuery [RBHS04] where subqueries of a query might refer to remote XML data sources, connected over a WAN. Here, a subquery that hits in the local cache, will not have to be sent over the network, and the savings can be huge.

Checking query/view answerability requires matching operations between the tree patterns of the query and view (we discuss this in Section 3). Looking up the semantic cache by iterating over all views, will be very inefficient when the number of views is large. We list below our main contributions:

- We show a method for checking query/view answerability $\exists C. Q \equiv C \circ V$, by string operations which capture the semantics of the required tree operations.

- We describe a novel cache organization, in which view expressions are stored in relational tables, and cache lookup is done by issuing SQL queries.

- We demonstrate that cache lookup is very efficient, even when there are several hundred thousand cached views.

- We describe a method for warm-up view selection, when given a warm-up query workload.

- We demonstrate impressive speedups for query workloads having locality.

Example: We now present some examples of how queries are answered from the cache. They will make clearer the challenges in doing efficient lookup in a large cache, and also illustrate query rewriting for cache hits. Suppose we have three cached views, as shown below.

V1	/a[u[@v]/w][x]
V2	/a[x//y][p//r]
V3	//a[@v>50]

Suppose query Q1 is **/a[x][u[w]/@v]/b**. It is clear V1 answers it, with the Composing Query C1 being **/*/b**. In this case, a recursive reordering of predicates will make V1 a prefix string of Q1. Consider query Q2 = **/a[p/q/r][x/y//z]/b**. It is answered by V2, with C2 being just Q2. In this case, we need to check that view predicate **[p//r]** "contains" query predicate **[p/q/r]**, and so on. Finally, let query Q3 be **//a[@v>100]**. It is answered by V3, with C3 being **/*[@v>100]**.

The rest of this paper is organized as follows. We start with related work in Section 2. Section 3 describes how we determine query/view answerability. Section 4 describes how our cache is stored, and looked up. Section 5 talks about view selection, based on the warm-up workload of queries. In Section 6, we present our experimental results, and then conclude in Section 7.

2   Related Work

A related problem is that of containment between XPath queries. In [MS02], this problem is shown to be coNP-complete. A polynomial time algorithm is also presented for checking containment, which is sound but not complete. However, note that V contains Q does not imply V answers Q. For example, let V = **/a/b**, and Q = **/a[x]/b**. It is clear that Q is contained in V. But it is not possible to answer Q from the result of V. [BOB+04] is the only other work we are aware of, that proposes using materialized XPath views. Their criterion for query/view answerability is exactly containment between them. Their version of what we call Composing Queries can require navigating up from the result nodes of the view being used. For each view, they store one or more of XML fragments, object ids, and typed data values.

We chose to restrict ourselves to just XML fragments, and defined query/view answerability accordingly. This choice allows us to maintain our cache outside the database too, and target applications like middle-tier caching and distributed XQuery, which we mentioned earlier. Application-tier caching for relational databases has received a lot of attention lately, in the context of database-driven web sites [LKM+02, YFIV00, FLSY99]. These sites serve dynamic content, obtained by querying the database. Caching query results and maintaining materializing views, at the middle-tier, has become increasingly common. Further, an XML database system maintaining the cache can easily choose to store object ids of the result nodes, instead of the entire result fragment. The techniques that we describe in this paper, will still remain applicable.

3   Using XPath Views

We now look at query/view answerability. The question that we consider is this: Given a view V and query Q, does V answer Q, and if yes, then what should C be so that C∘V ≡ Q. XPath queries are naturally represented as tree patterns. We are going to reason using these, to derive a sound but incomplete procedure for answering this question.

We first present an example showing how a XPath query is represented as a tree pattern. Figure 1 shows the tree pattern for Q = **a[v]/b[@w="val1"][x[.//y]]//c[z>val2]**. Child and descendant axes are respectively shown by single line and double line edges. The ellipse-shaped nodes are predicates qualifying their parent nodes. Note that the result node **"c"** of the query is marked by double
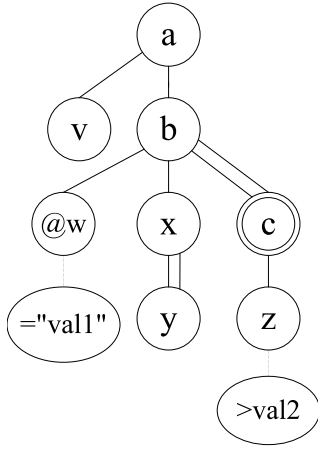
Figure 1: Example Tree Pattern

circles.

Defn 1 *The **Query Axis** is the path from the root node to the result, in the query tree pattern. Nodes on this path are the "axis nodes", while the others are "predicate nodes". The **Query Depth** is the number of axis nodes.*

Defn 2 *$Prefix(Q,k)$ is the query obtained by truncating query Q at its k-th axis node. The k-th axis node is included, but its predicates are not. $Preds(Q,k)$ is the set of predicates of the k-th axis node of Q.*

Example: Consider the query Q of Figure 1. It has depth three, with a,b,c respectively being its first, second and third axis nodes. $Prefix(Q,2) = \mathbf{a[v]/b}$, and $Preds(Q,2) = \{[\mathbf{@w="val1"}],[\mathbf{x[.//y]}]\}$.

The XPath fragment we cover includes the '/' and '//' axes, and '*' node labels. Predicates can be any of these: equalities with string or numeric constants, comparisons with numeric constants, or an arbitrary XPath expression from this fragment.

3.1 Criteria for Answerability

Given some C, we will check equivalence between C∘V and Q, by checking equivalence between their tree patterns. We first describe how the tree pattern for C∘V is obtained from C and V. Let the axis of V (from root to result) be $(x_1, x_2, \ldots, x_k)$ (see Figure 2). Observe that C is applied to the result of V. The label of the root of C must match that of the result of V if C∘V is to return anything. Let C have axis $(x_k, x_{k+1}, \ldots, x_n)$. Then, to obtain the tree for C∘V, we combine the trees of V and C by fusing the result node of V with the root node of C. Figure 2 illustrates this, and shows the two tree patterns we have to check for equivalence. Q_k denotes the subtree of Q rooted at its k-th axis node. We now briefly talk about XPath minimality and containment.
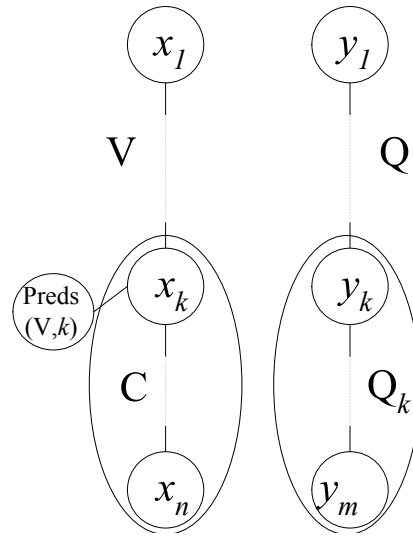
Figure 2: Tree Patterns for C∘V and Q

V answers Q if:	
1.	Prefix(V,k) and Prefix(Q,k) have isomorphic trees, where k is the query depth of V.
2.	∀ view–pred ∈ Preds(V,k) ∃ qry–pred ∈ Preds(Q,k) such that qry–pred ⊆ view–pred.

Table 1: Criteria for answerability

Defn 3 (XPath Minimality) *A XPath query Q is minimal [FFM03] if it is not possible to drop a subtree from its tree pattern, and get an equivalent query.*

XPath Containment Mappings: A containment mapping [MS02] from XPath query A to B, is a mapping from nodes in the A tree pattern to those in the B tree pattern such that:

- Labels of mapped nodes match.

- A's root and result nodes respectively go to B's root and result.

- Child edges go child edges, and descendant edges to downward paths.

The existence of such a mapping is sufficient, but not necessary, for B ⊆ A to hold. However, in most practical settings, we expect this mapping to exist when containment holds. In this paper, we check containment using these mappings.

Theorem 1 *If two tree patterns are minimal, and containment mappings exist both ways (so that they are equivalent), then they are isomorphic i.e., they are the same tree.*

Table 1 lists a set of sufficient (but not necessary) conditions for checking if view V answers query Q.

Example: Let $V = /a[u[@v]/w]/b[x//y][p//r]$, and $Q = /a[u[w]/@v]/b[x/y//z][p/q/r]/c$. V has depth 2. $\text{Prefix}(V,2) = /a[u[@v]/w]/b$, and $\text{Prefix}(Q,2) = /a[u[w]/@v]/b$. They are equivalent, and the first condition is satisfied. $\text{Preds}(V,2) = \{[x//y],[p//r]\}$, and $\text{Preds}(Q,2) = \{[x/y//z],[p/q/r]\}$. Containment between corresponding predicates can be easily checked by containment mappings, and the second condition too is satisfied. Thus, V answers Q.

We now present the intuition behind these conditions. Assume that V and Q are minimal. As in the relational case, we expect real, user-written queries to be minimal. Since we can choose to minimize the Composing Query C, we take C too to be minimal. By Theorem 1, we need the minimized tree of C∘V to be isomorphic to the tree of Q. In minimizing, suppose the tree for $\text{Prefix}(V,k)$ remains unchanged, where k is the query depth of V. Note that it is a subtree of V, and thus, minimal to start with. Then, we can see from Figure 2 that the tree for $\text{Prefix}(V,k)$ needs to be isomorphic to that of $\text{Prefix}(Q,k)$. This gives us the first condition. We now need to make the lower subtrees isomorphic. Note that the subtree rooted at x_k is simply C, with the predicates at its root node augmented by $\text{Preds}(V,k)$. Minimizing this subtree should give us Q_k, the subtree of Q rooted at its k-th axis node y_k. From Theorem 2 below, C is just Q_k. Thus, if we add the predicates $\text{Preds}(V,k)$ to the root node of Q_k, and then minimize, we get back Q_k. Then, from Lemma 5 in [FFM03], each of these predicates must contain some subtree rooted at a child of Q_k. This gives us the second condition.

Theorem 2 *Suppose view V of depth k answers query Q. Then the Composing Query C is just Q_k, the subtree of Q rooted at its k-th axis node.*

Proof: Denote by C', C with its root node augmented by $\text{Preds}(V,k)$. As observed above, minimizing C' should give us Q_k. Thus, they are equivalent, and $Q_k∘V \equiv$ C'∘V. Observe that C'∘V $\equiv$ C∘V. The former is just the latter, with the predicates at node x_k augmented by $\text{Preds}(V,k)$. But x_k already has this set of predicates, in the latter. Adding them again leaves the semantics of the query unchanged. Combining these equivalences, we get that $Q_k∘V \equiv$ C∘V. We can choose C to be Q_k. ∎

We now have the conditions for determining if V answers Q. If it does, we also know how to find C. The first condition requires checking isomorphism between two trees, and the second condition requires setting up containment mappings between trees representing predicates. Looking up a cache storing a large number of views, by checking these tree-based conditions for each view, will give a very high lookup overhead.

Normalize-Tree(T)
1. Let T have axis $(x_1,\dots,x_k)$.
2. For $i=1,\dots,k$ Normalize-Node(x_i).
3. Concatenate the node labels of $x_1,\dots,x_k$, with appropriate axes in between.
4. Return the query string formed.
Normalize-Node(x)
1. Let x have predicate node children $p_1,\dots,p_n$.
2. For $i=1,\dots,n$ Normalize-Node(p_i).
3. Sort $p_1,\dots,p_n$ lexicographically by their labels.
4. For $i=1,\dots,n$ append "[p_i.label]" to x.label.

Table 2: Normalizing a tree pattern

3.2 String-Based Answerability Checking

To achieve efficient cache lookup, we will look to check answerability using string matching. This will be cheaper, and more importantly, amenable to indexing. The first condition in Table 1 is not the same as string equality between $\text{Prefix}(V,k)$ and $\text{Prefix}(Q,k)$. This is because of reordered predicates, use of different syntax for the same predicate (e.g. "[x/y]" instead of "[x[y]]"), and so on. We describe in Table 2 a procedure for obtaining a unique XPath query string from a tree pattern T. Recall that axis nodes are the nodes lying on the query axis, while the others are predicate nodes.

Thus, the "normal form" of a query is obtained by normalizing its tree pattern. For example, the normal form of $/a[q][p]/b[x[z]/y]$ is $/a[p][q]/b[x[y][z]]$. It is clear from our definition that a given tree pattern has a unique normal form. Thus, if V and Q are rewritten in normal form, the first condition reduces to just string equality between $\text{Prefix}(V,k)$ and $\text{Prefix}(Q,k)$.

To check the second condition, for each result node predicate that the view has, we have to look for a containment mapping from its tree to trees of, potentially, each of the predicates in $\text{Preds}(Q,k)$. Having to do this for a large number of candidate views for a query, can be quite expensive. Note that, we need to check this condition only for views that satisfy the first condition. However, for a cache with a very large number of views, the number of views satisfying the first condition may not be small. We want to avoid the overhead of repeatedly setting up containment mappings. This is what we do. For the tree of each predicate in $\text{Preds}(Q,k)$, we generate all the trees that map to it. We normalize the root nodes of these trees, obtaining the set of all normalized predicates that contain some predicate in $\text{Preds}(Q,k)$. We denote this set by $\text{ConPreds}(Q,k)$. Now, checking the second condition is simple. We just need to check that every predicate in $\text{Preds}(V,k)$ is also present in $\text{ConPreds}(Q,k)$. This only requires string matching between predicate strings.

Generating Containing Trees: We now describe how, all the trees mapping to a given tree T can be

V answers Q if:
1. Prefix(V,k) = Prefix(Q,k) where k is the depth of V.
2. Preds(V,k) $\subseteq$ ConPreds(Q,k)

Table 3: Checking answerability by string matching

generated. Consider any subset S of the node set of T. Consider each pair of nodes x, $y \in$ S such that no other node on the path from x to y in T, is in S. If this path is a single child edge, put either a child or a descendant edge between x and y. Otherwise, put a descendant edge. This gives us a tree with node set S, which contains T. Iterating over all possible choices of S, and for each such S, taking all possible combinations of edges between such node pairs x, y in S, gives us all the trees that map to T.

Example: The set of all predicates containing the predicate **[x/y//z]** is listed in the table below. For the sake of clarity, we have not normalized them, thus avoiding the nested square brackets.

[x]	[x/y]	[x/y//z]
[x//y]	[x//y//z]	[x//z]
[.//y]	[.//y//z]	[.//z]

It is clear that the number of containing trees that we generate for any tree T is exponential in the number of nodes of T. However, tree patterns of individual predicates are typically very small. Further, these trees have to be generated only once for a given query. At the risk of getting more cache misses, we can choose to restrict ourselves to containing patterns that are linear paths. This is what we do in our implementation.

In Table 3, we restate the conditions for checking if view V answers query Q. They capture the same semantics as the tree-based conditions, and use only string operations. We assume V and Q have been rewritten in normal form.

3.3 Supporting Comparison Predicates

The last thing we need to do for our query/view matching procedure, is incorporate support for comparison predicates. So far, when generating containing trees, we had assumed that in a containment mapping, if two nodes match, their labels have to be identical. However, tree patterns can include nodes which are comparison predicates qualifying their parent node (see Figure 1). We will refer to them as Comparison Tags, or CTs. Our current scheme will return that view **//book[@price>50]** can't answer the query **//book[@price>100]**, which is clearly wrong. We need to allow two CT nodes to match in a containment mapping, if the node label in the containing tree is "more general" than that in the contained tree. Thus, label "> x" can map to "> y" provided $y \geq x$. Similarly, "< x" can go to "< y" provided $y \leq x$. This is what we do to allow this. For each

XmlData (<u>viewId</u> int, fragment XML)
Prefix (<u>prefixId</u> int, prefix string)
View (viewId int, prefixId int, pred string, allPreds string, CTs string)

Table 4: Schema of the cache tables

predicate being checked, we yank out all the CTs it contains, and store them separately. Thus, in the predicate sets involved in the second condition in Table 3, each predicate now has two components: the first stores the predicate string, and the second stores the CT's, and their positional information. For example, the predicate "@price>50" would be stored as "(@price,(6,>50))" indicating that CT with label ">50" occurs at position 6. When comparing some predicate p in Preds(V,k) with q in ConPreds(Q,k), we first match the string components. If they match, we then match the CT components, with CT labels being matched as described above. If the CTs also match, then $p \in$ ConPreds(Q,k).

4 Cache Organization and Use

This section describes how our semantic cache is stored, maintained and used. Our views are stored in relational tables, and their results are stored as XML fragments. The scheme we present here assumes a relational database with native XML support. It can trivially be adapted to the case when the relational and XML database systems are different. Table 4 shows the schema of the three tables storing the cache. Note that the "fragment" column is of type XML. In our experiments, we used the beta release of Microsoft SQL Server 2005, which supports storing and querying XML.

4.1 Inserting a View

We will illustrate how the cache is stored by showing how a new view V is inserted into the cache. We first execute V on the XML database, and fetch its result. This result fragment is inserted into the *fragment* column in table *XmlData*. The *viewId* field is a system generated key, and we record the value that it takes in the inserted tuple. We rewrite V in its normal form (see Section 3.2). Suppose V has depth k. We insert Prefix(V,k) in the *prefix* column in table *Prefix*, if it is not already present. *prefixId* is again a system generated key, and we record the value that it takes. We finish by inserting a tuple in table *View* for V. The *viewId* and *prefixId* fields in *View* are foreign keys referencing *XmlData* and *Prefix* respectively. They take the values we have just recorded for them. Now, consider Preds(V,k), the set of predicates at the result node of V. Each predicate has a string portion and a CTs (Comparison Tags) portion. Thus, we have a set of strings, and an associated set of CTs. From the

strings, we choose one to insert into the *pred* column. In Section 4.3, we describe how we make a "good" choice. The remaining strings are combined, and inserted into the *allPreds* column. One or both of these columns will be NULL, when Preds(V,k) respectively has one or zero elements. Finally, we combine all the CTs and insert them into the *CTs* column. This completes the tuple. The values for the *allPreds* and *CTs* columns are constructed so that the constituent strings and CTs can be recovered (we can insert suitable delimiters, or use any other such simple scheme). Thus, view V and its result XML can be recovered from the cache.

Example: Let V = **/a[v]/b[x][@y=50][z>100]**. If 1 and 2 are generated for *viewId* and *prefixId*, then tuple **(1,XML_Result)** goes into *XmlData*, tuple **(2,"/a[v]/b")** into *Prefix*, and tuple **(1,2,"@y=50","z|x","|(1,>100)|")** into *View*.

4.2 Cache Lookup

We now describe how the cache is looked up, for a given query Q. Recall from Table 3, the conditions we need to check to determine if some view V answers Q. Suppose Q has depth n. From the first condition, it is clear that any V of depth k can possibly work only if $k \leq n$. Further, we will prefer using a V with as large a k as possible. The larger k is, the simpler the Composing Query C is, and the smaller the view result fragment is. Thus, we want to try candidate views in order of decreasing depth. Also, observe that the query-dependent part of the conditions in Table 3 involves Prefix(Q,k) and ConPreds(Q,k). Once we fix k, these get fixed. Our approach will be this. We will iterate over k going from n to 1. For each k value, we will execute a single SQL query which will return a small set of all candidate views of depth k. We will inspect the returned views, and if any of them works, we have a hit. Otherwise we try the next value of k, and so on. Figure 3 lists this algorithm, including the SQL query used. If no view was found which answers Q, then we have a cache miss. Q is executed on the XML database, and inserted as a new view.

In the SQL query shown, ConPreds(Q,k) stands for the list of string portions of the predicates in ConPreds(Q,k). The query returns tuples in the *View* table. Each such tuple represents a single cached view V. Consider any such V returned. The *prefix* field for V is Prefix(Q,k). Thus, V satisfies the first condition for answering Q. Further, its *pred* field is either NULL, which means Preds(V,k) is empty, and the second condition is trivially satisfied. Or it matches the string portion of some predicate in ConPreds(Q,k), which means the second condition is partially satisfied. ConPreds(Q,k) will be tiny relative to the set of result node predicates of all cached views. We expect the selection on the *pred* field to make the SQL query very selective, and return a small set of candi-

Cache-Lookup(Q)

For $k = n,\ldots,1$
    Execute this SQL query:
        Select V.*
        From Prefix P, View V
        Where P.prefix = Prefix(Q,k) *and*
            P.prefixId = V.prefixId and
            (V.pred is NULL
            or V.pred in (ConPreds(Q,k)))
    Inspect the returned views.
    If some view answers Q, return it and exit.
Return null.

Figure 3: Cache lookup for query Q

date views for us to inspect. We sequentially examine the returned tuples for the second condition, till we find a view V that satisfies it. Recall that Preds(V,k) can be reconstructed from the returned tuple for V.

If Q is a cache hit, and some V of depth k answers it, the Composing Query C is just the subtree of Q rooted at its k-th axis node, as we saw in Theorem 2. We further simplify C by removing from its root node, those predicates which also occur in Preds(V,k).

Example: Suppose our cache has three views, as shown below.

V1	/a/b[z]/c
V2	/a/b[@y="str"][z>200]
V3	/a/b[w][@y="str"]

Suppose Q = **/a/b[w[x]][@y="str"][z>100]/c**. For $k = 3$, we get no candidate views. For $k = 2$, the SQL query returns V2 and V3. The matching of Comparison Tags for Preds(V2,2) fails, since **[z>200]** does not contain **[z>100]**. However, V3 is found to answer Q, with C being **/b[w[x]][z>100]/c**.

4.3 Cache Policies

To support our SQL query, we create clustered indexes on *Prefix(prefix)* and *View(prefixId,pred)*. We now describe how, when inserting views, the predicate to put in the *pred* column is chosen. The selectivity of our SQL query very much depends on this choice. Among all views having the right prefix, the query returns those whose *pred* field lies in ConPreds(Q,k). If we choose for the *pred* column a more popular predicate, then this view is likely to more often be retrieved as a false positive. On the other hand, if we choose a rarer predicate, then this view has a much lesser chance of being retrieved as a false positive. So, we need to identify the rarer predicates in Preds(V,k). Predicates can be equalities, comparisons, or paths (XPath expressions without any equality or comparison predicates). Equality predicates can take a lot of different values, since a typical attribute or element will take many different values in the XML data instance. Path predicates are however, restricted by what the DTD allows, and can take far fewer values. The CTs of comparison

predicates are stored separately, reducing the string portion to just a path predicate. Equality predicates are the best choice. Thus, if Preds(V,k) has any equality predicates, we randomly choose one of them as the one that goes into the *pred* column.

When query Q is a cache miss, we insert it as a new view in the cache. However, there is a catch here. We do not want to insert Q if its result size is very large. Our motivation for caching is to be able to execute simpler queries on small, cached fragments when we have a cache hit. Querying large fragments may not be of much benefit. Further, the XML database will typically maintain suitable indexes over the data. However, we don't maintain any indexes for the cached fragments. The overhead of indexing them was found to be more than the benefit. Running a query on a large, unindexed fragment could easily be worse than running it on the database itself. Further, the space cost for large fragments is also high. Thus, we keep a size limit on cached views. If some view has result size greater than this limit, we choose not to cache it. In our experiments, we used a size limit of 128 KB.

4.4   Answering XQuery

We first describe the XQuery fragment we cover. Given an XQuery X, we define any XPath query embedded in X to be a *base XPath* of X, if evaluating its value requires accessing the data, and can't be done from the current environment. For example, suppose X were this:

for $b in /site/regions
return $b//item

Then, the XPath in the "for" clause is a base XPath of X. It determines the binding for $b. However, the XPath in the "return" clause can be evaluated from the binding for $b, and is not a base XPath. Thus, the results of the base XPaths of X define all the XML data needed to evaluate X. The XQuery fragment we cover consists of those X whose base XPath queries do not contain references to other variables in X (they are meaningful queries independently), and which belong to the XPath fragment that we cover.

We now describe how we can use the cache in answering an XQuery X in this fragment. Suppose the base XPaths of X are $\{Q_1,\ldots,Q_k\}$. We lookup each Q_i in the cache, setting V_i to the view that answers it, and C_i to the corresponding CQ. If Q_i is a cache miss, it will be executed, and inserted as a new view. We then rewrite X by replacing each Q_i with the query C_i, over the result fragment of V_i. This rewritten query is equivalent to X, and can be evaluated using only the cached data. Due to our fragment size limit, it may not be possible to have a V_i in the cache that answers Q_i, for some i. In that case, we execute X directly on the XML database.

5   Warm-up View Selection

We now describe how we warm-up the cache, given a warm-up workload of queries. The conventional way would be to just pose the queries in the warm-up workload, to the cache. Those that are cache misses would be brought in as new views, and the cache would be populated. It turns out that we can do better by being more proactive in choosing which views to insert. If the warm-up workload is representative of the test workload (in other words, if it comes from the same hypothetical query generation process that generates the test workload), then we can use it to obtain a much larger workload S which the test workload is likely to have a significant overlap with. We can then warm-up by inserting an optimal set of views that answers all of S, and thus, formulate cache warm-up as a view selection problem.

5.1   Generating Potential Queries

We first look at how we obtain workload S from the warm-up workload W. The problem setting here is this: Given W, we want to obtain all the queries likely to be in the test workload. A possible approach that is likely to work well would be a machine learning one, which would use a probabilistic model for the query generation process, and learn its parameters using W. Then, potential queries could be generated in order of decreasing probabilities. Such an approach is beyond the scope of this paper. Here we use a much simpler technique, which is likely to work well for a query workload having locality.

We obtain the *template* of a XPath query by yanking out all string and numeric constants occurring inside predicates. The template is a parameterized query, which captures the structure of the original query. For example, the template for **/a[v]/b[w][@x="str1"][y/z>50]** is **/a[v]/b[w][@x=#][y/z>#]**, where the # symbols indicate missing parameters. An instance of this template is created by inserting a string value for the first parameter, and a numeric value for the second. We record all the distinct templates we see in the warm-up workload W. Our set S will consist of instances of exactly these templates.

We next describe how we obtain the parameter values, that we will use in instantiating these templates. The label for a template parameter is created by combining the labels of the corresponding predicate, and the axis node to which this predicate is attached. In the above example, the label of the first parameter is **b[@x]**, and that of the second is **b[y/z]**. Note that different query templates can have parameters with the same label. For example, the template **//b[@x=#]/c** has a parameter with label **b[@x]**. We treat such parameters with the same label, as the same. This is because, in most cases, the parameter label will identify a unique path from the root to the parameter, in

the XML tree. Thus, parameters with the same label will have the same domain of meaningful values that they can take. For each parameter, we record all the different values that it is seen to take in W, and the number of times each of these values occurs.

We now have a set of templates, and a set of parameter values, extracted from the warm-up workload W. Observe that the locality of a workload comes from locality in the templates and parameter values used, in creating the workload queries. Further, suppose the hypothetical query generation process independently selects the query template, and parameter values. This will often be the case. For example, we expect the "hot" values for some parameter like **book[@author]** to be the same, across different query templates having this parameter. This is how we we create the workload S: we create instances of each recorded template by trying all combinations of recorded values of its parameters, and then, take the union over all templates. Under the assumption of locality, the test workload will have a significant overlap with S. We now want to insert views so as to be able to answer all queries in S.

The main cost in warm-up, is that of executing queries on the XML database, to bring in new views. The size of S will typically be orders of magnitude larger than W. To ensure that the time taken for cache warm-up is reasonable, we put a bound M on the maximum number of views that we may insert. In our experiments, we set M to be three times the size of the warm-up workload. Under this constraint, we want to select a set of views that answers as large a subset of S as possible. This defines our view selection problem. We next describe the approach that we take.

5.2 View Selection

The set of views that we will consider is the queries of S itself. We define the **"utility"** of a view to be the cardinality of the subset of S that it answers. The larger this value, the better the view is for caching. The problem setting here is just a variant of the set cover problem, which is known to be NP-complete. Each view represents a subset of S. We want to pick subsets which together cover a maximal subset of S, under the constraint that we can pick at most M subsets. The algorithm we will use is a simplification of the popular greedy approximation algorithm for this problem [CLR90]. We sort the queries of S in order of decreasing utilities, and pick them in that order. If a picked query is a cache miss, it gets inserted as a new view. We stop when we have inserted M views, or reached the end of this sorted list. Unfortunately, the algorithm as described, is computationally very expensive. Computing the utilities of the views requires query/view matching between all pairs of queries of S, and S is very much larger than the warm-up workload W.

View-Select()

1. Compute Utility(T) for each template T.
2. Sort templates by decreasing utilities.
3. Suppose the sorted order is $(T_1, \ldots, T_n)$
4. Set *Inserted* = 0.
5. For $i = 1, \ldots, n$
      If T_i is marked covered, skip to next iteration.
      Obtain all query instances $(Q_1, \ldots, Q_k)$ of T_i
      For $j = 1, \ldots, k$
          Pose Q_j to cache.
          If its a cache miss, do *Inserted*++.
          If *Inserted* = M, exit.
      Mark as covered each T_i' that T_i answers.

Figure 4: Warm-up View Selection

It turns out that the same result can be achieved at a much lesser cost, if we work directly with the query templates, instead of their instances. Given templates T and T', we say T answers T' if, for every instance of T', there exists an instance of T which answers it. For example, **/a/b[p//r=#]** answers **/a/b[@x=#][p/q/r=#]/c**. We just need to make the values of the parameters **b[p//r]** and **b[p/q/r]** equal. It can be easily shown that if some instance q' of T' is answered by q of T, then T answers T'. Further, every parameter in T will have a matching parameter in T'. The instances of T' that some instance of T answers, are obtained by varying those T' parameters which either don't have matching parameters in T, or match with a T parameter which occurs in a result node comparison predicate in T (we explain this later). In the above example, there is a single such T' parameter, **b[@x]**. Thus, each instance of T answers exactly the same number of instances of T'. This implies that, for any template T, all instances of T answer the same number of queries in S, and thus have the same utility value. We call this the "utility" of the template, and its value is given by:

$$\text{Utility(T)} = \frac{\sum_{T'} \text{count(T')}}{\text{count(T)}}$$

where the sum is taken over all T' that T answers, and count(T') denotes the number of instances that T' generates. Observe that computing the utilities for all templates requires query/view matching between all pairs of templates, as opposed to all pairs of instances of these templates. To match templates, we just match canonical instances of them, obtained by choosing some fixed value for each parameter. Thus, the queries of S can be obtained in the required order by sorting the templates in order of decreasing utilities, and then replacing each template with all the instances it generates. Figure 4 shows the view selection algorithm.

5.3 Some Warm-up Heuristics

The above algorithm may fail to bring in views that cover all of the queries in workload S. We now briefly describe three heuristics that were found to improve the view selection, and give a higher cache hit rate on the test workload.

Recall that, when recording values taken by a parameter, we also record the number of times each of these values is seen. We can put a bound K on the maximum size that the value list for a parameter can take. If some parameter takes more than K values, we retain only the top K. Basically, we are removing queries from S which are less likely to occur in the test workload. We set K to 30 in our experiments.

Suppose some template parameter occurs in a comparison predicate at the result node. For example, the template **//book[@price>#][@author=#]** has such a parameter **book[@price]**. Suppose the smallest value in the value list for this parameter is v. Observe that if we can answer the instance **//book[@price>v][@author="str"]**, we can answer any other instance **//book[@price>w][@author="str"]**, since $w \geq v$ will hold. A similar comment holds when we have a "<" comparison. In that case, we choose v to be the largest value instead. Thus, for all templates, we can fix the values of such parameters (and thus, they are no longer considered parameters). When applicable for a template T, this reduces the number "count(T)" of instances that T generates, and increases its utility value. Thus, this optimization has an important effect on which views get inserted.

For selecting views, we have so far considered only the warm-up workload templates. However, it might be the case that there are other, better views. Recall that a view is good if it has a high utility value, and if its result size is less than our size limit (otherwise it won't be cached at all). We now describe a heuristic to add new templates to the set of templates we are considering. We "generalize" a template by truncating it as much as possible, while still retaining one parameterized predicate. For example, the generalization of **/a/b[@x=#][y/z>#]/c[v=#]** is **/a/b[@x=#]**. For each template T, we add its generalization G to our set of templates. We expect such a G to be a good view template. It clearly answers T. It will be a short template, and is likely to answer a few other warm-up templates too. Further, since it has a parameterized predicate, instances of G will, in most cases, not have very large results, and will be cacheable. Thus, the set of templates we consider in the view selection algorithm of Figure 4 now becomes larger. Everything else remains the same.

```
Generate-Path()
1. Set cur to the root node.
2. Set path = "/cur.label"; depth = 1.
3. With probability (depth/max_depth)², exit.
4. If cur has no outgoing edges, exit.
5. If cur has outgoing edges {e₁,...,eₖ}, take one,
   where probability of taking eᵢ is ∝ w(eᵢ).
6. Set cur to the new node reached.
7. Set path = path + "/cur.label"; depth++.
8. Go back to step 3.
```

Figure 5: Simple Path Generation

6 Experiments

6.1 XPath Generator

We implemented an elaborate XPath query generator, to create real-looking workloads for our experiments. To generate a query, it first generates a simple path, and then, creates and inserts predicates. A key feature of the query generator is that predicates are created using values actually taken by elements and attributes in the XML data. Thus, it generates meaningful queries, many of which return non-null results when executed. We used a 300 MB XML document generated by the XMark [SWK⁺01] generator, in our experiments. We now describe the steps involved in generating a query.

Generating a Simple Path: The document DTD is preprocessed, and converted into a directed graph the usual way. We insert an edge from x to y if y is a child or attribute of element x. Any path from the root node to any other node represents a simple path expression, without predicates. Each edge e, going from some node x to y, is assigned a weight $w(e)$ which is the average number of y children that each x element has, in the XML data instance (this is obtained by executing suitable "count" queries). We have a parameter max_depth which is the maximum depth of any generated query. Figure 5 shows how a simple path is generated. Note that we are not generating the descendant axis or wildcard labels.

Choosing Predicates: The number n of predicates that are inserted is given by $(r \times depth)$, rounded to the nearest integer, where $depth$ is the number of nodes in the generated path, and r is a real-valued parameter. The set of available predicate types (considering only structure) consists of path predicates, and attributes of the path nodes. We count a path predicate at each path node, as a single predicate type, just like an attribute predicate. Children of path nodes having textual (#PCDATA) content are also included among the attribute predicates, to give us more choices. We then randomly choose some n predicate types from this set. For each of them, a predicate is created, and inserted at the appropriate place in the query.

Creating Attribute Predicates: In a

preprocessing phase, for each attribute, we extract and store all the distinct values it takes in the XML data. We also store whether its a string or numeric attribute. To create a predicate for some chosen attribute, we take the attribute name, and append an '=' sign for string attributes, or randomly any of '=', '>' or '<' for a numeric attribute. We then select a value from the stored values for this attribute using a zipf distribution with exponent z (thus, the probability of choosing the i-th value is $\propto 1/i^z$), where z is a parameter. We then append this value, completing the predicate.

Creating Path Predicates: We restrict ourselves to linear paths, instead of arbitrary trees. Our approach here is similar to that used in generating simple paths. We first choose a child of the path node for which we are creating the predicate. This is done as in step 5 of Figure 5. We set the current node *cur* to this child, and set the predicate to "*cur.label*". If *cur* has some m attributes, then with an equal probability of $1/(m+2)$, we either choose one of the m attributes, or stop at *cur* itself, or choose to take an outgoing edge to some child of *cur* (again as in step 5 of Figure 5). If we choose an attribute, then we create an attribute predicate as described above, append it to complete the predicate, and stop. If we move to a child of *cur*, we set *cur* to that child, append "*cur.label*"" to the predicate, and proceed similarly as before.

Workloads Used: For the workloads we generated, the *max_depth* parameter was set to 7, and r to 0.6. Thus, queries of depths 3, 4 and 5 had 2, 2 and 3 predicates respectively. Below we show three example generated queries:

- /site/open_auctions/open_auction[initial][@id = "open_auction1001"]

- /site/people/person[homepage][name="Aenne Ermakov"]/emailaddress

- /site/regions/samerica/item[quantity>2][name][@id="item14898"]/location

6.2   Cache Lookup Performance

Cache lookup for a query Q involves returning the *viewId* (see Table 4) of a view V that answers Q, and the query C to be applied to result of V. We now see how much time it takes to lookup the semantic cache. Note that lookup does not include obtaining the result of Q, by executing C (for a cache hit) or Q (for a cache miss). In fact, for the experiments in this subsection, we skipped the execution of the XPath queries. Thus, when some Q missed in the cache, it was inserted as a new view, without its accompanying result.

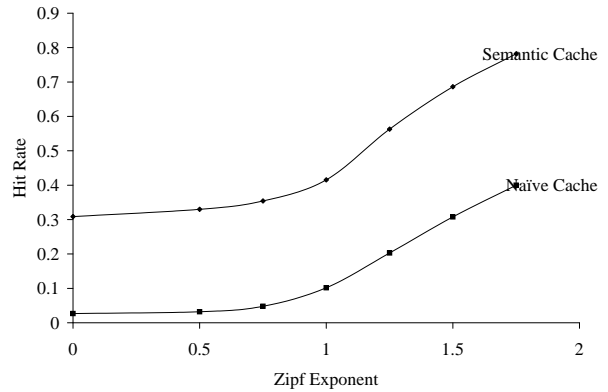The experiments were run on a Pentium 4 machine with 512 MB RAM, running Windows. We used the

Figure 6: Hit Rate vs Zipf Exponent

beta release of Microsoft SQL Server 2005, for both the cache and XML databases. The cache parameter M (maximum number of views to be inserted during warm-up) was set to three times the size of the warm-up workload. For warm-up heuristics (see Section 5.3), K was set to 30, and the "generalized templates" heuristic was not used.

We will compare our semantic cache with a naive cache, which stores (query, result) pairs in a table with schema (*query string, result XML*). A Query Q is a hit for the naive cache only if the exact same query string is present in the cache. Thus, the naive cache does not use its cached queries as views. Figure 6 shows how the hit rate varies with the zipf exponent z used for creating attribute predicates, in the workload generation (see Section 6.1). We used warm-up and test workloads of 5,000 and 50,000 queries respectively, for each z value. The semantic cache gives hit rates which are more than 30% higher. The query/view answerability we capture is thus, much richer than naive string matching. For the remaining experiments, we have used workloads generated using $z = 1.5$.

Figure 7 shows how the average cache lookup time varies with the size of the test workload. Here we are looking to determine how well lookup scales to a large number of stored views. In all cases, the same warm-up workload of size 20,000 was used. We can see that the lookup time for the semantic cache remains constant at around 13 ms, even as the workload size increases to half a million queries. This time is very small compared to the time taken to execute a typical XPath query in the workload. This is exactly what we would like to have. The naive cache takes a mere 0.47 ms per lookup. However, in query processing, this difference will be offset by the higher hit rate of the semantic cache, as we will see later.

Recall that lookup for a query Q of depth n will require executing some d SQL queries, where $d \leq n$, and inspecting their results (see Figure 3). The depth of the Composing Query C will be exactly d. If Q is a cache miss, then d will be n. Figure 8 shows lookup times separately for different subsets of the test work-
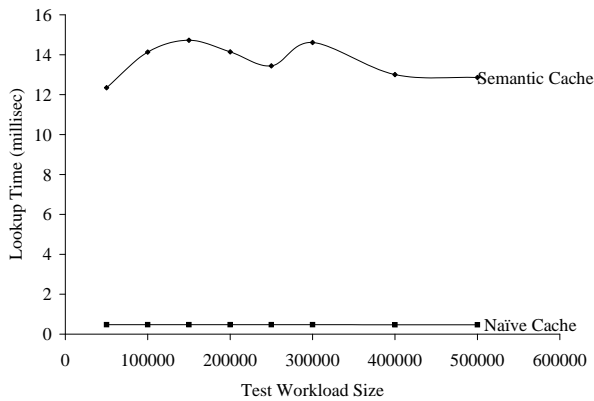
Figure 7: Cache Lookup Time vs Workload Size
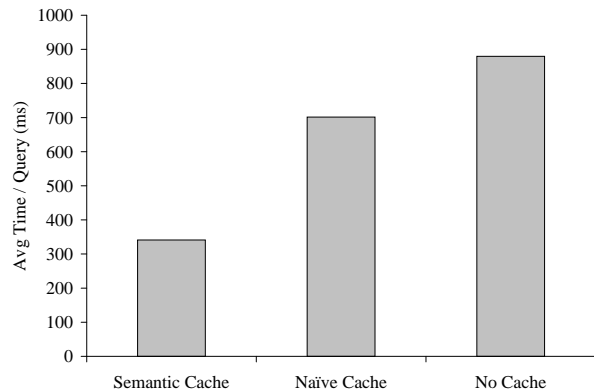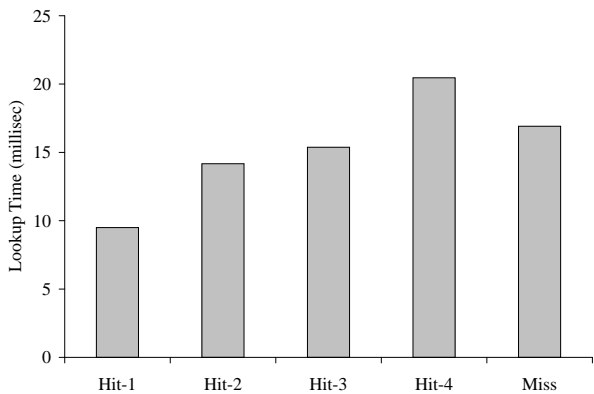

Figure 10: Query Processing Times

Figure 8: Time taken by Lookup Outcome. Hit-d denotes hits having Composing Queries of depth d
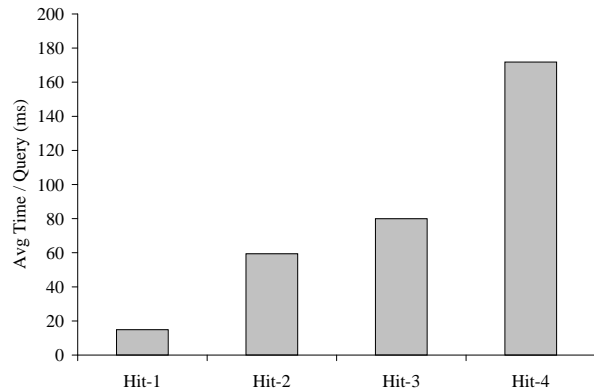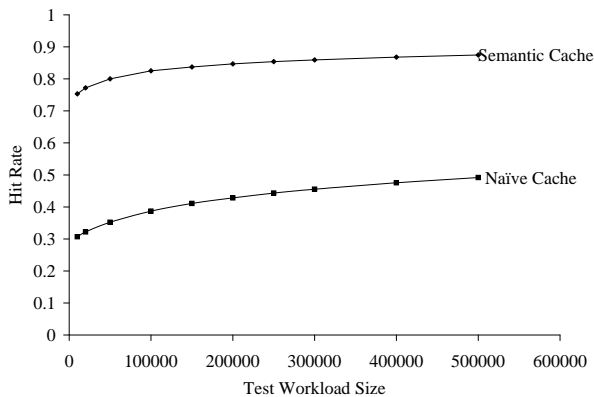

Figure 11: Query Times by Lookup Outcomes

Figure 9: Hit Rate vs Workload Size

load. Hit-d denotes the set of queries whose Composing Queries were of depth d. As expected, as d increases, the lookup time for Hit-d increases. Here, we had again used a test workload of 500,000 queries, and warmed up on 20,000. Finally, Figure 9 shows how the hit rate varies with the test workload size.

6.3 Query Processing Performance

We now show the speedup obtained in query processing, by employing the semantic cache. The experimen-

tal setup is the same as in the previous subsection, with a couple of differences. First, we turned on the "generalized templates" heuristic. Second, we filtered the generated queries, and included in the test workload only those which were found to execute in at most 5 seconds. Some of the generated queries were taking more than a minute on the 300 MB XML database (even after creating the path index for XML data that SQL Server allows [PCS+04]). These queries, when answered from the cache, could give us speedup results which were misleadingly high, and where the bulk of the speedup came from a few select cache hits. The workload zipf exponent z, used in creating attribute predicates, was again 1.5. This seems high. However, note that, for most attributes, we were sampling from tens of thousands of values. Further, our XPath generator generates query structure completely randomly, and a higher z value compensates for this absence of locality in structure.

We used warm-up and test workloads of 1500, and 8500 queries respectively. Figure 10 shows the average time per query, for three different configurations. When not caching, the queries took 880 milliseconds each. Having the naive cache brought this down to 700 ms, while employing the semantic cache brought this down to 340 ms, which is a speedup by a factor of 2.6. Recall that Hit-d denotes the subset of queries, having

	Semantic Cache	Naive Cache
Avg Time/Hit (ms)	26.11	5.03
Avg Time/Miss (ms)	1428.39	1255.22
Final Size (KB)	6037.29	3634.29
Hit Rate	0.78	0.44

Table 5: Cache measurements

Composing Queries of depth d. Figure 11 shows, for different values of d, the average time taken for queries in Hit-d. The increase in time, as d increases, is much sharper than what we saw for cache lookup alone in Figure 8. This confirms one of our main motivations for caching: small depth queries, executed on small fragments, run faster than larger depth queries, executed on larger fragments. A cache miss of depth n can be thought of as an extreme case, where a query of depth n is executed on a very large fragment, the entire XML data.

Finally, Table 5 shows some additional measurements. For a query which is a cache hit , the semantic cache needs to query a cached fragment. This is unlike the naive cache, which simply returns the whole fragment (at least for XPath workloads). Considering this, the average hit time of 26 ms for the semantic cache is impressive. An interesting observation is that the average time per miss for the semantic cache is 1428 ms, which is much higher than the average of 880 ms, for a query on the XML database. We know that cache lookup only takes an extra 15 ms. Thus, the workload queries that are cache misses, have a larger average execution time on the XML database, than those which are cache hits. This can be explained as follows. If query Q is a hit, and is answered using view V, then the database can clearly answer Q from the disk pages it needed for answering V. It is likely that some or all of these disk pages are still in memory, when Q is presented. Thus, when not using a semantic cache, queries which would have been cache hits, still execute faster on average than those which would have been misses. Finally, despite a higher hit rate, the semantic cache ends up with a larger size. This is because of its proactive view selection during warm-up.

7   Conclusions

We described a technique for employing a semantic cache of materialized XPath views. Our notion of query/view answerability gave a much higher hit rate than a naive string matching cache. We used string operations for cache lookup, and demonstrated the scalability of our lookup method to a large number of views. Cache hits were processed well over an order of magnitude faster than misses. We obtained impressive speedups on XPath workloads having locality. Semantic caching is likely to prove very useful, where applicable. Interesting directions for future work would be to answer a larger XQuery fragment, to study more so-

phisticated methods for view selection, and to explore methods for maintaining these views in the presence of XML database updates.

References

[BOB+04] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, pages 60–71, 2004.

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

[DFJ+96] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[FFM03] Sergio Flesca, Filippo Furfaro, and Elio Masciari. On the minimization of xpath queries. In *VLDB*, pages 153–164, 2003.

[FLSY99] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web sites. In *VLDB*, pages 627–638, September 1999.

[LKM+02] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay, and J. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD*, pages 600–611, 2002.

[MS02] Gerome Miklau and Dan Suciu. Containment and equivalence for an xpath fragment. In *Proceedings of PODS*, pages 65–76, 2002.

[PCS+04] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V. Zolotov. Indexing xml data stored in a relational database. In *VLDB*, pages 1134–1145, 2004.

[RBHS04] Chris Re, Jim Brinkley, Kevin Hinshaw, and Dan Suciu. Distributed xquery. In *Workshop on Information Integration on the Web (IIWeb)*, pages 116–121, 2004.

[SWK+01] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.

[YFIV00] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *The VLDB Journal*, pages 188–199, 2000.