Answering XML Queries Using Materialized Views Revisited

Xiaoying Wu
New Jersey Institute of Technology, USA
xw43@njit.edu

Dimitri Theodoratos
New Jersey Institute of Technology, USA
dth@cs.njit.edu

Wendy Hui Wang
Stevens Institute of Technology, USA
hwang@cs.stevens.edu

ABSTRACT

Answering queries using views is a well-established technique in databases. In this context, two outstanding problems can be formulated. The first one consists in deciding whether a query can be answered exclusively using one or multiple materialized views. Given the many alternative ways to compute the query from the materialized views, the second problem consists in finding the best way to compute the query from the materialized views. In the realm of XML, there is a restricted number of contributions in the direction of these problems due to the many limitations associated with the use of materialized views in traditional XML query evaluation models.

In this paper, we adopt a recent evaluation model, called inverted lists model, and holistic algorithms which together have been established as the prominent technique for evaluating queries on large persistent XML data, and we address the previous two problems. This new context revises these problems since it requires new conditions for view usability and new techniques for computing queries from materialized views. We suggest an original approach for materializing views which stores for every view node only the list of XML nodes necessary for computing the answer of the view. We specify necessary and sufficient conditions for answering a tree-pattern query using one or multiple materialized views in terms of homomorphisms from the views to the query. In order to efficiently answer queries using materialized views, we design a stack-based algorithm which compactly encodes in polynomial time and space all the homomorphisms from a view to a query. We further propose space and time optimizations by using bitmaps to encode view materializations and by employing bitwise operations to minimize the evaluation cost of the queries. Finally, we conducted an extensive experimentation which demonstrates that our approach yields impressive query hit rates in the view pool, achieves significant time and space savings and shows smooth scalability.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems−*query processing, textual databases*

General Terms: Algorithms, Performance

Keywords: XPath query evaluation, XML, materialized views

CIKM'09, November 2–6, 2009, Hong Kong, China.

1. INTRODUCTION

XML is by now the standard for exchanging, exporting and integrating data on the web. As increasing amounts of information are stored, exchanged, and exported using XML, it is becoming increasingly important to efficiently query XML data sources. Answering queries using views is a well-established technique in data integration, query caching and warehousing, where queries expressed over data sources are answered using materialized views defined over these data sources [14]. It is also (along with indexing) one of the best known techniques used for optimizing the evaluation of queries [7, 12]. The problem behind this technique can be formulated as follows: given a query and a set of materialized views along with their definitions, decide whether the query can be answered using the materialized views. If the answer is positive, usually there are alternative ways to compute the query from the materialized views inducing different evaluation costs. Consequently, another related problem consists in finding the best way to compute the query from the materialized views. These problems have been studied extensively in the realm of relational databases. However, there is a restricted number of contributions in that direction in the context of XML. The reason is the many limitations associated with the use of materialized views when a traditional way for evaluating queries on XML documents is adopted.

Limitations of Previous Approaches. The core of XPath consists of tree-pattern queries with one output node (TPQs). The answer of a TPQ is the set of subtrees rooted at the matches of the query output node against the XML document tree. The presence of an output node on queries and views, and the absence of complete structural information outside the subtrees in the view materializations, greatly reduces the chances of a query to have a hit of one or more views in the pool of materialized views that together can be used to answer the query. For this reason, some approaches suggest the materialization of additional information about the view answers, e.g. ancestor path information [2]. However, keeping this information only partially addesses the issue while increasing the size of data that needs to be stored. Storing, in addition, data values and references to XML data [2] assumes a centralized enviroment and is not appropriate when the queries need to be answered using only the materialized views (that is, when the base XML data is not accessible). Further, the size of the answer subtrees can be very large. When multiple views are materialized (and inevitably ovelapping portions of the XML document are repeatedly and redundantly stored), view materialization becomes unfeasible due to space limitations. Even if space limitations are met, usually the view materializations are unindexed fragments of the XML document making the computation of a query more expensive compared to computing it against the original XML document. For this reason, in the performance studies of both [19] and [28] an upper

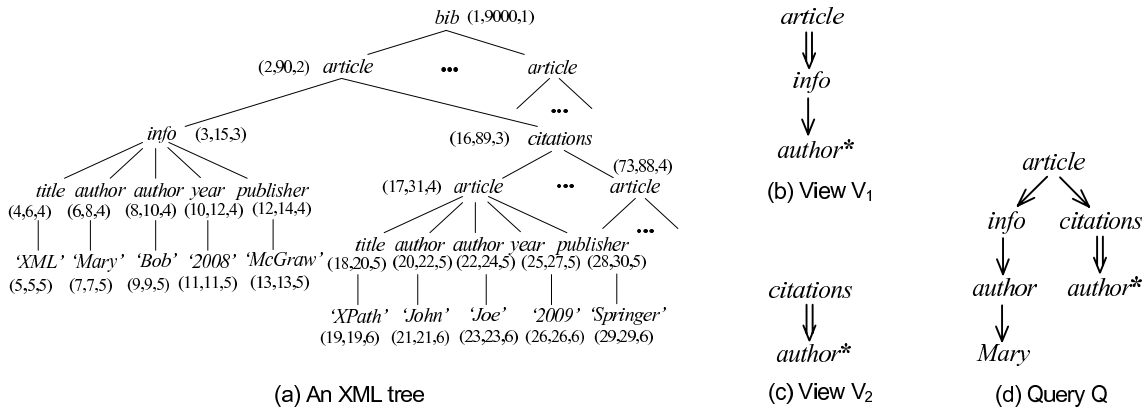


Figure 1: An XML tree with two views and one query on this tree

bound has been set on the size of the XML fragment per view that can be materialized. This restriction limits both (a) the chances to answer the query using only the materialized views, and (b) the chances to find an efficient evaluation plan for the query using the materialized views. These obstacles defy the reason for materializing views in the first place.

EXAMPLE 1.1. *Consider the XML tree of Figure 1(a) which records bibliographic information (ignore for the moment the triplets associated with the tree nodes). Let's assume that the view* V_1 : $//article//info/author$*, which retrieves article authors, and the view* V_2 : $//citations//author$*, which retrieves citing authors, are materialized in the client cache. Views* V_1 *and* V_2 *are shown as TPQs in Figures 1(b) and 1(c) respectively, where an asterisk denotes an output node. Suppose the user issues the query* Q : $//article\,[info/author = 'Mary']/citations//author$ *against the client cache. The query asks for the authors who cite articles authored by Mary and is shown as a TPQ in Figure 1(d). One can see that query* Q *cannot be answered using* V_1 *and/or* V_2*. The reason is that no structural information is available outside the view answer subtrees in the view materializations. Query* Q *cannot be answered using* V_1 *and/or* V_2 *even if ancestor path information is stored along with the subtrees in the view materializations because the absence of node identifiers does not allow a structural join on the materializations of the two views. Query* Q *can be answered using* V_1 *if node* $article$ *is the output node of* V_1*. However, in this case, the materialization of* V_1 *is the whole base XML tree, and* V_2 *redundantly materializes part of it. Such a large materialization is likely prohibitive in the client cache, and if it is not, in the absence of an index on the materialization of* V_1*, it would probably be preferable to evaluate* Q *against the base XML data stored in the server instead of using the views materialized in the client cache.*

The Inverted Lists Evaluation Model. A recent approach for evaluating queries on large persistent XML data assumes that the data is preprocessed and the position of every node in the XML tree is encoded [4, 16]. Further, the nodes are partitioned by node label, and an index of inverted lists is built on this partition. In order to evaluate a query, the nodes of the relevant inverted lists are read in the pre-order of their appearance in the XML tree. We refer to this evaluation model as *inverted lists* model. All the relevant query evaluation algorithms in this model are based on stacks that allow encoding an exponential number of pattern matches in a polynomial space. Comparison studies on XML query evaluation techniques [21, 13] show that holistic algorithms [4, 9, 16, 33, 8, 31] in the inverted lists model are superior to other algorithms and evaluation models (streaming/navigational approaches [23] or sequential/string matching approaches [25]). In this paper, we assume that the inverted lists model and holistic evaluation algorithms are adopted. Note that in the inverted lists model, the answer of a TPQ is not a subtree of the XML tree but a set of tuples. The fields of the tuples correspond to the query nodes. Each tuple contains the (positional representation of) XML tree nodes that match the query nodes in an embedding of the query to the XML tree.

Problem Addressed. Driven by the prominence of the inverted lists evaluation model, we address the problem of answering TPQs using exclusively one or more materialized views in the context of this model. We also address the problem of the optimal evaluation of a TPQ using exclusively materialized views in the same context.

In this new context, query answerability by materialized views is not restricted by the presence of output nodes in queries and views since all query and view nodes can be seen as output nodes. As a consequence, queries have more chances to have a hit involving one or more materialized views in the view pool.

This new framework revises the "answering queries using materialized views" problem since previous conditions for query answerability are not valid anymore. Further, traditional approaches [2, 19, 32, 1, 28] evaluate queries by generating compensation TPQs over materialized views and look at the optimization of this evaluation as a problem of finding the lowest cost compensation TPQ. Unfortunately, these techniques are not applicable in the new context and novel stack-based techniques need to be devised for computing queries over view materializations.

Our Approach. We suggest a novel approach for materializing views where instead of materializing the view answer, we materialize sublists of the inverted lists for the labels of the view nodes. A query can be computed very efficiently using materialized views by running holistic stack-based algorithms over the inverted sublists of the view nodes.

Going back to Example 1.1, the triplets by the nodes of the XML tree of Figure 1(a) denote the positional representations of these nodes. As we show later, in the context of our approach, not only the TPQ Q of Figure 1(d) can be answered using the materializations of views V_1 and V_2 of Figures 1(b) and 1(c), but also this computation can be performed very efficiently. Moreover, view materialization takes minimal space and any redundancy is avoided.

Contribution. The main contributions of our paper are the following.

- We introduce a new setting for answering queries using views in the framework of the inverted lists query evaluation model. We suggest a novel approach for view materialization where instead of storing the answer of a view, we store for every view node only the list of XML nodes necessary for computing the answer

of the view. This way, an exponential number of solutions can be stored in polynomial space.
- We consider tree-pattern queries and views and we specify necessary and sufficient conditions for answering a query using exclusively one or multiple materialized views in terms of homomorphisms from the views to the query.
- In order to check the answerability of a query using views, we design an efficient stack-based algorithm for finding all view nodes that are mapped to the same query node through a homomorphism. Our algorithm runs in polynomial time and space by avoiding to enumerate a possibly exponential number of homomorphisms.
- We show how a query can be computed using views in the framework of our novel concept of materialized view by running state of the art holistic stack-based algorithms over the materialializations of the views that cover the query.
- We show that an additional advantage of our novel concept of view materializations is that they can be stored as bitmaps and therefore consume minimal space by avoiding redundantly materializing overlapping fragments of the XML data. Further, we show that query evaluation can be optimized by finding a maximal number of covering views in the view pool and by applying bitwise operations to their materializations.
- We conduct an extensive experimentation which shows that our approach is space efficient and obtains largely higher hit rates in the view cache compared to traditional approaches, it achieves significant performance gains compared to evaluating queries without using views, and scales very smoothly in terms of space and computational overhead when the number of materialized views in the view pool increases.

2. RELATED WORK

Because of the increasing importance of XML, a number of papers have recently addressed the important problems of XML query rewriting using views and of XML view selection [5, 2, 34, 27, 19, 32, 22, 1, 6, 29, 28]. A common assumption made by most of these works is that a view materialization is a set of subtrees rooted at the images of the view output nodes, or references to the base XML tree. In order to obtain the answer of the original query, downward navigation in the subtrees is needed.

Two types of XML query rewriting problems, namely, equivalent rewritings and contained rewritings have been considered. An equivalent rewriting produces all the answers to the original query using the given view materialization(s), whereas a contained rewriting may produce a subset of the answer to the query. The majority of the recent research efforts have been directed on rewriting XPath queries using materialized XPath views. Among them most works focus on the equivalent rewriting [2, 19, 32, 27, 1]. Balmin et al. [2] presented a framework for answering XPath queries using materialized XPath views. A view materialization may contain XML fragments, node references, full paths, and typed data values. A query rewriting is determined through a homomorphism from a view to the query and the view usability (or query answerability) depends on the availability of one or more of the four types of materializations. Mandhani and Suciu [19] presented results on equivalent TPQ rewritings when the TPQs are assumed to be minimized. Xu et al. [32] studied the equivalent rewriting existence problem for three subclasses of TPQs. Tang and Zhou [27] considered rewritings for TPQs with multiple output nodes. However, the rewritings are restricted to those obtained through a homomorphism from the view to the query which maps the query output nodes to the view output nodes (*output preserving* homomorphism). The problem of maximally contained TPQ rewritings was studied in [17] both in the absence and presence of a schema. All contributions in [2, 19, 32, 27, 17, 1] are restricted to query rewritings using a *single* materialized view. A common constraining requirement for view usability is the existence of a homomorphism that satisfies two conditions: (a) it maps the view output node to an ancestor-or-self node of the query output node, and (b) it is an isomorphism on query nodes that are not descendants of the image of the view output node.

The problem of equivalently answering XPath queries using multiple views has been studied in [1, 6, 29, 28]. Arion et al. [1] considered the problem in the presence of structural summaries and integrity constraints. As in [27], a query can have multiple output nodes, and a rewriting is obtained by finding output preserving homomorphisms from views to the query. Answers of views are tuples whose attributes include node ids of the original XML tree, XML subtrees, and/or nested tuple collections. The answer to a query is computed by combining the answers to the views through a number of algebraic operations. The materialization scheme of storing node ids together with XML subtress is also adopted by [6, 29]. Both papers assumed that output preserving homomorphisms exist among views and they presented rewriting algorithms which use intersection of view answers on node ids.

Tang et al. [28] addressed the multiple view rewriting problem based on the assumptions that structural ids in the form of extended Dewey codes [18] are stored with view materializations. This way, the common ancestors of nodes in different view fragments can be derived for checking view usability. Also, structural joins on the view fragments can be performed based on Dewey codes to produce query answers. The paper also studied a view selection problem defined as finding a minimal view set that can answer a given query. In [34, 22] the equivalent rewriting problem has been addresses but for queries and views which are XQuery expressions.

Our approach is orthogonal to all the previous traditional approaches.Note that structural encodings of XML data nodes are also employed in [1, 28]. However, unlike our approach, [1, 28] store node encodings toghether with view materializations (XML tree fragments) and use them mainly for combining answers of multiple views to produce query answers.

Philips et al. [24] consider materializing intermediate query results as sets of tuples in order to allow additional evaluation plans for structural joins. However, their context of view usability is very restricted and they do not address query answerability from materialized views issues.

3. DATA MODEL, QUERY LANGUAGE, AND EVALUATION MODEL

In this section, we briefly present the data model, the class of queries and views we consider, and the inverted lists evaluation model we adopt. We also introduce our novel concept of view materialization.

Data Model. An XML database is commonly modeled by a tree structure. Tree nodes represent and are labeled by elements, attributes, or values. Tree edges represent element-subelement, element-attribute, and element-value relationships. For simplicity, we do not distinguish here between element, attribute, and value nodes, and we denote by $\mathcal{L}$ the set of node labels in the XML tree.

For XML trees, we adopt the region encoding widely used for XML query processing [4, 16]. This encoding associates every node with a triplet (*begin*, *end*, *level*). This triplet is called *positional representation* of the node. The *begin* and *end* values of a node are integers which can be determined through a depth-first traversal of the XML tree, by sequentially assigning numbers to the first and the last visit of the node. The *level* value represents

the level of the node in the XML tree. The utility of the region encoding is that it allows efficiently checking structural relationships between two nodes in the XML tree. For instance, given two nodes n_1 and n_2, n_1 is an ancestor of n_2 iff $n_1.begin < n_2.begin$, and $n_2.end < n_1.end$.

Query and View Language. For simplicity of presentation and in order to highlight the novel features of our approach, we consider that queries and views are tree-pattern queries (TPQs). We comment later on how our approach can be applied to broader classes of queries e.g. queries with reverse axes and wildcards.Contrary to all previous approaches on answering queries using views [2, 19, 1, 17], we do not impose any restriction on the output nodes. Queries and views can have any number of output nodes and this does not affect the usability of the views for the evaluation of the queries. For this reason, in our definition below we do not explicitly refer to output nodes, and all the nodes of queries and views are considered to be output nodes. Our approach applies without modification to the case where arbitrary sets of nodes in queries and views are considered to be output nodes.

A *tree-pattern query* (TPQ) specifies a pattern in the form of a tree. Every node in a TPQ Q has a label from $\mathcal{L}$. There are two types of edges in Q. A single (resp. double) edge between two nodes in Q denotes a child (resp. descendant) structural relationship between the two nodes.

The answer of a TPQ on an XML tree is a set of tuples. Each tuple consists of XML tree nodes that preserve the child and descendant relationships of the query.

More formally: an *embedding* of a TPQ Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) a node in Q labeled by a is mapped by M to a node of T labeled by a; (b) if there is a single (resp. double) edge between two nodes X and Y in Q, $M(Y)$ is a child (resp. descendant) of $M(X)$ in T.

We call *image* of Q under an embedding M a tuple that contains one field per node in Q, and the value of the field is the image of the node under M. Such a tuple is also called *solution* of Q on T. The *answer* of Q on T is the set of solutions of Q under all possible embeddings of Q to T.

A view is a named query. The class of views we consider is not restricted. Any kind of query can be a view.

Outline of the Inverted Lists Evaluation Model. In the inverted lists evaluation model, the data is preprocessed and the position of every node in the XML tree is encoded. For every label in the XML tree, an inverted list of the nodes with this label is produced. Given an XML tree T, we use L to denote its set of inverted lists and L_a to denote the inverted list in L for label a. List L_a contains the positional representation of the nodes labeled by a in T ordered by their *begin* field.

Let Q be a query. With every query node X in Q labeled by a, we associate the inverted list L_a in L. To access the nodes in L_a for X, we maintain a cursor C_X. Cursor C_X sequentially accesses the nodes in L_a starting with the first node.

With every query node X in Q, we also associate a stack S_X. At the beginning of the evaluation of a query, all stacks are empty. When the nodes in the inverted lists are accessed by the cursors, they are possibly stored in stacks. At any point in time, stack entries represent partial solutions of the query that can be extended to the solutions as the algorithm goes on.

In the following we ignore the XML tree T and we assume that the input for the evaluation of queries and views is the set of inverted lists L. When a query Q is evaluated on L, if the cursor of a node X in Q iterates over the inverted list L_Y we say that node X is *computed on L using the list L_Y*.

View Materialization. We now define our novel concept of view materialization.

DEFINITION 3.1. *Let V be a view, and L be a set of inverted lists. The* materialization $V(L)$ *of V on L is a set of sublists of the inverted lists in L–one for each view node in V. If X is a node in V labeled by a, L_X denotes its inverted list in $V(L)$ and it contains only those nodes of $L_a \in L$ that are images of X in a solution of V on L. Sublist L_X is called the* materialization of X in $V(L)$.

In this sense, the inverted lists in the materialization $V(L)$ contain only those nodes of the inverted lists in L that contribute to a solution of V on L.

Our approach for view materialization departs from all the previous approaches which consider materializing copies of XML tree fragments, typed values, ancestor paths, or references to the input XML tree [2, 19, 17, 1, 28]. Note that our approach is space efficient since the sublists can encode in linear space a number of solutions for the view which is exponential on the number of view nodes.

4. ANSWERING QUERIES USING VIEWS

Let Q be a query and X be a node in Q labeled by a. Recall that in order to evaluate Q on L, the cursor C_X of X iterates over the inverted list L_a in L. If there is a sublist, say L_X, of L_a such that Q can be computed on L by having C_X iterate over L_X instead of L_a, we say that node X *can be computed using L_X on L*. Let V be a view whose materialization on L is $V(L)$. The idea of our approach for answering Q using V on L is to identify nodes in Q that can be computed using the materializations of nodes in V for every L and use their materializations in $V(L)$ for computing the answer of Q on L instead of using the corresponding inverted lists in L.

4.1 Answering a Query Using a Single View

We start by defining what answering a query using a view means in our context of view materialization.

DEFINITION 4.1. *Let $V(L)$ be the materialization of a view V on a set of inverted lists L. A query Q* can be answered using *V if for a node X in Q there is a node Y in V with the same label as X, such that for every L, X can be computed using $L_Y \in V(L)$. In this case, we say that view node Y* covers *query node X, or that Y is a* covering node *of X.*

Let's assume that Q can be answered using V. If every node in Q is covered by a node in V, we say that Q can be answered completely *using V. Otherwise, we say that Q can be answered* partially *using V.*

When the answer of a query is computed using a view, a node of the query that is covered by a view node uses only the materialization of this view node. Since the materialization of the view node is a sublist of the inverted list for the node label, it is usually smaller than the inverted list. This reduces the cost for computing the answer of the query.

Deciding Whether a Query Can be Answered Using One View. In order to specify conditions for view usability, we need the concept of homomorphism between views and queries. A *homomorphism* from a view V to a query Q is a mapping that maps all the nodes of V to nodes with the same label in Q and preserves child and descendant relationships (preserving a descendant relationship means that it is mapped to a path of nodes).

Figure 2 shows a query Q and a view V and four homomorphisms h_1, h_2, h_3 and h_4 from V to Q.

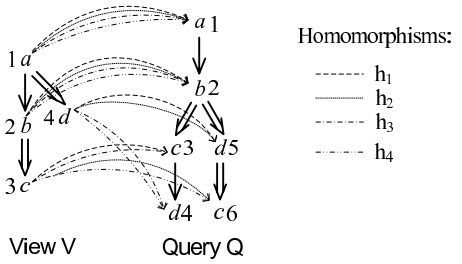


Figure 2: Four homomorphisms from view V to query Q

The following theorem relates node coverage to homomorphisms.

THEOREM 4.1. *Let Q be a query and V be a view. A node X in Q is covered by a node Y in V iff there is a homomorphism from V to Q that maps Y to X.*

Necessary and sufficient conditions for view usability based on homomorphisms are provided by the next collorary of Theorem 4.1.

COROLLARY 4.1. *Let Q be a query and V be a view. Query Q can be answered using V iff there is a homomorphism from V to Q.*

For instance, in the example of Figure 2, query Q can be answered using view V since there is at least one homomorphism from V to Q. Both nodes labeled by d in Q are covered by node d in V.

Notice that our definition of homomorphism is less restrictive than previous ones since we do not have to consider (and impose conditions on) output nodes [19, 32, 17]. This increases the chances for a homomorphism from a view to a query to exist. Based on Theoreom 4.1, it also increases the chances of the view to be useful in answering the query. This constitutes an important advantage of our approach compared to previous ones since it allows the exploitation of views when other approaches fail.

In order to guarantee that a query can be answered completely using a view, we need to make sure that *every* node of the query has a covering node in the view. The next corollary of Theorem 4.1 expresses this requirement in terms of homomorphisms from the view to the query.

COROLLARY 4.2. *Let Q be a query and V be a view. Query Q can be answered completely using V iff there are homomorphisms from V to Q such that every node of Q is the image of a node in V under some homomorphism.*

Based on Corollary 4.2, one can easily see that in the example of Figure 2, query Q can be answered completely using view V.

Computing the Answer of a Query Using One View. In the traditional approach for answering a query using a view [2, 19, 32, 1, 28], the query is rewritten using the view. That is, in order to compute the answer of the query, a compensation query is determined which is applied to the materialized view and computes the answer of the query. This compensation query does so by navigating in the view materialization which is a set of subtrees of the original XML tree.

In contrast, in our approach, we use the view materialization and compute the query answer by running stack-based evaluation algorithms over the materializations of the covering view nodes.

Therefore, in order to perform the computation of the answer what is needed is an association of the query nodes with covering

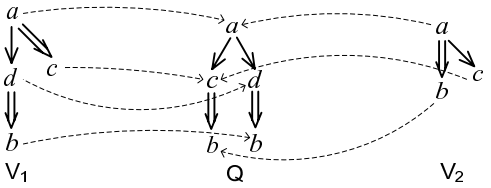


Figure 3: Query Q and views V_1 and V_2 and homomorphisms

view nodes. The set of covering view nodes of a given query node is determined by the homomorphism of Theorem 4.1 as follows:

Let $h_1, \ldots, h_k$ be the homomorphisms from a view V to a query Q and $Y_i^1, \ldots, Y_i^{m_k}$ be the nodes in V whose image under h_i is X. Then, the set $m(X)$ of covering nodes for X in V is

$$m(X) = \bigcup_{i \in [1,k],\ j \in [1,m_k]} \{Y_i^j\}$$

If $\exists X \in Q$, $m(X) \neq \emptyset$, Q can be answered using V. If $\forall X \in Q$, $m(X) \neq \emptyset$, Q can be answered completely using V. The materialization in $V(L)$ of any node in $m(X)$ can be used for computing X. However, we might also use the materializations of multiple (or all the) nodes in $m(X)$: let L_{X_1} and L_{X_2} be the materializations of two nodes X_1 and X_2 in $m(X)$. The *intersection* $L_{X_1} \cap L_{X_2}$ is the sublist of L_{X_1} and L_{X_2} which comprises the nodes that appear in both L_{X_1} and L_{X_2}. In order to compute the answer of Q using V any subset of $m(X)$ can be used: during the computation of the answer, X will be computed using the intersection of the materializations of the view nodes in this subset.

Note that a view V can have a number of homomorphisms to a query which is exponential in the number of view nodes. However, the number of covering nodes in $m(X)$ is bounded by the number of nodes in V.

4.2 Answering a Query Using Multiple Views

The presence of multiple views in the view pool increases the chances of a query to be answered using their materializations. We extend below our definition for answering a query using a view to multiple views. We first define the *union* of the materializations of two view nodes. Let X_1 and X_2 be two view nodes with the same label a, and L_{X_1} and L_{X_2} be their materializations. The *union* $L_{X_1} \cup L_{X_2}$ of L_{X_1} and L_{X_2} is the sublist of L_a which comprises exactly the nodes of both L_{X_1} and L_{X_2}.

DEFINITION 4.2. *Let $V_1(L), \ldots, V_n(L)$ be the materializations of views $V_1, \ldots, V_n$ on a set of inverted lists L. A query Q* can be answered using *$V_1, \ldots, V_n$ if for a node X in Q, there are nodes $Y_1, \ldots, Y_k$ in $V_1, \ldots, V_n$, such that, for every L, X can be computed using $L_{Y_1} \cup \ldots \cup L_{Y_k}$.*

Let's assume that Q can be answered using $V_1, \ldots, V_n$. If for every node X in Q, there are nodes $Y_1, \ldots, Y_k$ in $V_1, \ldots, V_n$, such that, for every L, X can be computed using $L_{Y_1} \cup \ldots \cup L_{Y_k}$ for every L, we say that Q can be answered completely *using $V_1, \ldots, V_n$. Otherwise, we say that Q can be answered* partially *using $V_1, \ldots, V_n$.*

Deciding Whether a Query Can be Answered Using Multiple Views. For the class of queries we consider here, checking whether a query can be answered using multiple views can be expressed in terms of checking whether a query can be answered using a single view.

THEOREM 4.2. *Let Q be a query and $\{V_1, \ldots, V_n\}$ be a set of views. Query Q can be answered using $V_1, \ldots, V_n$ iff for some V_i, $i \in [1, n]$, Q can be answered using V_i.*

Figure 3 shows a query Q and two views V_1 and V_2. Each of these views has a homomorphism to Q which is also shown in the figure. Based on Corollary 4.1, Q can be answered using V_1 (or V_2). Therefore, based on Theorem 4.2, Q can be answered using V_1, V_2.

For the case of answering completely a query using views we can state the following theorem.

THEOREM 4.3. *Let Q be a query and $\{V_1, \ldots, V_n\}$ be a set of views. Query Q can be answered completely using $V_1, \ldots, V_n$ iff it can be answered using $V_1, \ldots, V_n$ and for every node in Q, there is a covering node in some (not necessarily the same) V_i, $i \in [1, n]$.*

Based on Theorem 4.3, one can see that query Q of Figure 3 can be answered completely uisng the views V_1 and V_2 of the same figure.

Computing the Answer of a Query Using Multiple Views. In order to perform the computation of the answer of the query using a set of materialized views we associate query nodes with the set of corresponding covering nodes in the views. The set of covering nodes of a given query node in multiple views is defined in terms of the set of covering nodes of the query in a single view: let X be a node in query Q, and $m_1(X), \ldots, m_n(X)$ be the sets of covering nodes of X in $V_1, \ldots, V_n$, respectively. Then, the set $m(X)$ of covering nodes of X in $V_1, \ldots, V_n$ is

$$m(X) = \bigcup_{i \in [1,n]} m_i(X)$$

As with the case of a single view, if $\exists X \in Q$, $m(X) \neq \emptyset$, Q can be answered using $V_1, \ldots, V_n$. If $\forall X \in Q$, $m(X) \neq \emptyset$, Q can be completely answered using $V_1, \ldots, V_n$. The materialization of any node in $m(X)$ can be used for computing X. However, we might also use the materializations of some (or all the) nodes in $m(X)$: during the computation of the answer, X will be computed using the intersection of the materializations of these view nodes in $m(X)$.

In this paper, we focus on answering completely queries using views.

5. COMPUTING COVERING NODES

As discussed in Section 4, given a query Q and a view V, the covering nodes for a node of Q in V are defined in terms of the homomorphisms from V to Q. However, the number of these homomorphisms can be exponential on the size of V. In this section, we present a stack-based algorithm which computes in polynomial time and space the covering nodes of the nodes in Q without explicitly enumerating all the homomorphisms from V to Q.

Match Sets. In the algorithm we use a data structure, called *match set*, which is similar to those employed in [15, 20, 3] for encoding query pattern matches.

Let q be a node in query Q and v be a node in view V. We say that v *matches* q if v has the same label as q. Let T_v and T_q denote the subtrees rooted at v and q, respectively. Let also v_i be a child node of v in V and q_j be a node in the subtree T_q. We say that the pair (v, q) is *consistent* with (v_i, q_j), if v and v_i match q and q_j, respectively, and if $v/v_i \in V$, then $q/q_j \in Q$.

The match set $MS(V, Q)$ is a directed acyclic graph (dag) that compactly stores the set of homomorphisms from V to Q. The nodes of this dag correspond to node pairs (v, q) such that v matches q. Each node (v, q) is associated with an array $ptrsArr$ indexed by the children of v in V. Given a child v_i of v in V, $ptrsArr[v_i]$ is a set of pointers. Each of the pointers points to a node (v_i, q_j), where q_j is a node in T_q and (v, q) is consistent with (v_i, q_j). There is an edge in the dag from node (v, q) to node (v_i, q_j) iff there is a pointer from $ptrsArr$ of (v, q) to (v_i, q_j). We call match set of a node (v, q), denoted $MS(v, q)$, the node (v, q) along with the array $ptrsArr$ of (v, q). Note that node (v_i, q_j) can be a child of multiple nodes $(v, q_1), \ldots, (v, q_n)$, where $q_1, \ldots, q_n$ are ancestor nodes of q_j in Q. Let r_V and r_Q denote virtual roots of V and Q, respectively. Then, the match set dag $MS(V, Q)$ is rooted at the node (r_V, r_Q). As we show later, the size of the dag is polynomial in the size of V and Q.

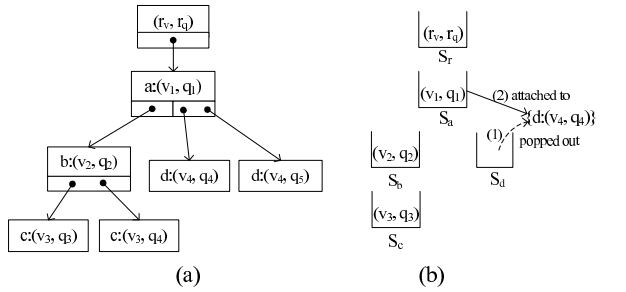


Figure 4: (a) The match set dag for the view and the query of Figure 2, (b) The snapshots of stacks after the query leaf node d has been visited during the execution of Algorithm $computeCovering$

Figure 4(a) shows the dag of the match set for the view V and query Q of Figure 2. In order to uniquely identify a node of the view or the query, every node of V and Q in Figure 2 is associated with a node id.

Given a match set dag $MS(V, Q)$, we can compute the set of homomorphisms from V to Q. Clearly, the time required for enumerating all the homomorphisms is exponential on the size of the view in the worst case. However, we do not need to enumerate all the homomorphisms in order to compute covering nodes of the query nodes. Instead, as we show below, we can compute covering nodes from the match set dag.

Computing Match Sets. The match set $MS(v, q)$ can be computed inductively by computing the match set of each child of v in V. If v is a leaf node of V, then $MS(v, q)$ consists of only node (v, q). Otherwise, suppose that we have computed all the match sets for each child v_i of v. Then, ptrsArr[v_i] of $MS(v, q)$ is populated by adding pointers to each child node (v_i, q_j) such that (v, q) is consistent with (v_i, q_j). If every ptrsArr[v_i] is non-empty after the population, we call the newly computed $MS(v, q)$ a *valid* match set.

Based on the above idea, we provide below an algorithm that efficiently computes match sets and covering nodes.

The Algorithm. Algorithm $computeCovering$, shown in Listing 1, takes a query Q and a view V as inputs and computes the covering nodes in V for each query node of Q. It is a stack-based algorithm which associates each view node of V with a stack. It proceeds in two steps. In the first step, it calls Procedure $constructMS$ (shown in Listing 2) to compute the match set dag $MS(V, Q)$ (line 2). In the second step, the dag is traversed top-down to determine the covering view nodes (lines 3-5).

Procedure $constructMS$ traverses the tree pattern Q in preorder, constructing the match sets as it visits nodes and traverses edges. When $constructMS$ visits a query node for the first time, it creates a match set for each matching view node. The created match set are pushed onto stacks. When $constructMS$ returns to a query node after traversing the entire subtree of this node, it determines whether the match sets created for the query node are valid and inserts into the arrays $ptrsArr$ of their parent nodes pointers that point to the corresponding nodes. When $constructMS$

Listing 1 Algorithm computeCovering

```
1  create a stack for each node of V and initialize the covering node set
   m(q) to be empty for each node q of Q.
2  constructMS(root(Q))
3  let visited be a boolean matrix where the rows are indexed by the
   nodes of V and the columns are indexed by the nodes of Q. Initialize
   each field of visited to be false
4  for (every node MS(v, q) encountered in the top down traversal of the
   match set dag of V and Q) do
5     if visited[v, q] is false, then add v to m(q), set visited[v, q] to
      true, and continue the traversal on the children of MS(v, q).
```

Listing 2 Procedure constructMS(q)

```
1  for (every v ∈ nodes(V) that matches q considered in post-order) do
2     let u be the parent of v in V
3     if (stack S_u is not empty) then
4        let (u,p) be in the top entry of S_u
5        if ((u, p) is consistent with (v, q)) then
6           create MS(v, q) and initialize ptrsArr[v_i] to be empty for
            every child v_i of v
7           push MS(v, q) to stack S_v
8  for (every child q' of q in Q) do
9     constructMS(q')
10 for (every v ∈ nodes(V) that matches q considered in pre-order) do
11    pop out the top entry e from stack S_v
12    if (e is a valid match set) then
13       let u be the parent of v in V
14       for (every stack entry e' ∈ S_u) do
15          add to ptrsArr[v] a pointer that points to the node of e
```

finishes the traversal of Q, $MS(r_V, r_Q)$ encodes all the homomorphisms from V to Q. We describe the process below in more detail.

Initially, a match set $MS(r_V, r_Q)$ is pushed onto stack S_{r_V}, the stack of the virtual view root. For each query node q visited for the first time, $constructMS$ iterates in postorder over each view node v matching the query node (line 1). Let (u, p) be the node of the match set corresponding to the top entry of stack S_u. Procedure $constructMS$ checks whether (u, p) is consistent with (v, q). If this is the case, a match set $MS(v, q)$ is created and then pushed onto stack S_v (lines 2-7). Next, $constructMS$ recursively calls itself on each child node of q (lines 8-9). After the traversal of the subtree of q, for each v matching q considered in preorder, it pops out the top entry $MS(v, q)$ from stack S_v (lines 10-11). If $MS(v, q)$ is valid, for each entry in stack S_u, where u is the parent of v, a pointer that points to (v, q) is created and added to the entry's ptrsArr[v_i] (lines 12-15).

Figure 4(b) shows a snapshot of the view stacks during the execution of Algorithm $computeCovering$. After the query leaf node d (node id 4) has been visited, the corresponding match set is popped out from the stack S_d of view node d. Since it is valid, it is attached to the only match set in stack S_a of view node a.

Complexity. Let v be a node in V. We define the *prefix* query of v, denoted $prefix(V, v)$, as the path from the root of V to v. Given a query Q, we define the *recursion depth of node v in Q* as the maximum number of nodes in a path of Q that are images of v under all the possible embeddings to $prefix(V, v)$ in that path of Q. We define the *recursion depth* D of V in Q as the maximum recursion depth of the view nodes of V in Q.

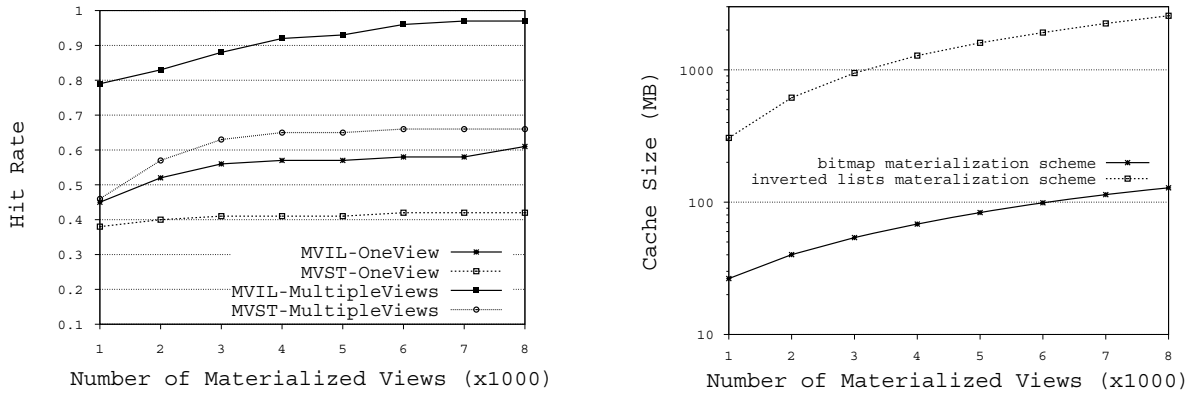
The number of query nodes matched by a view node is bounded by the number $|Q|$ of the nodes of Q. The total number of match sets constructed during execution is bounded by $|V| \times |Q|$. The number of incoming pointers to each constructed match set is bounded by D. Therefore, the space complexity of Algorithm $computeCovering$ is bounded by $O(|V| \times |Q| \times D)$.

The time complexity of Algorithm $computeCovering$ is determined by the time for processing stack entries (that is, match sets). The number of entries in each stack at any given time is bounded by D. Let v be a view node that matches a query node q under consideration. Procedure $constructMS$ spends $O(fanout(v) + D)$ on checking whether $MS(v, q)$ is valid and on visiting entries in the parent stack of v, where $fanout(X)$ denotes the out-degree of v in V. Since the number of view nodes that match node q is $O(V)$, the total time spent on processing stack entries for each node in Q is $O(|V|+|V| \times D)$, which is dominated by $O(|V| \times D)$. Therefore, the time complexity of Algorithm $computeCovering$ is bounded by $O(|V| \times |Q| \times D)$.

6. OPTIMIZATION ISSUES

Computation Time Issues. As discussed in Section 4, if a query Q can be answered completely using some views, and $m(X)$ is the set of all the covering nodes of a node X in Q with respect to these views, then X can be computed using the intersection of the materializations of the nodes in $m(X)$. If additional views that have a homomorphism to Q are discovered in the view pool, the set $m(X)$ of covering nodes for X with respect to all the views will potentially get new view nodes and the intersection of their materializations will potentially decrease in size making, of course, the computation of X cheaper. However, there is a cost associated with discovering additional views that have a homomorphism to Q. Therefore, if a set of views that answers a query Q has been discovered in the view pool, a question that arises is whether it is worth spending additional time to find other views that have a homomorphism to Q in an effort to reduce the overall computation cost of Q using the view materializations. Our experimental results in Section 7, show that the answer to this question is positive: the implementation of our algorithm of Section 5 takes minimal time to compute all the covering nodes of a query even with a large view pool. This is largely compensated by the benefit in computation time we obtain by finding additional views with homomorphisms to Q.

Using Bitmaps. Consider two view nodes X_1 and X_2 both labeled by the same label a. The materializations L_{X_1} and L_{X_2} of X_1 and X_2 are sublists of the inverted list L_a. L_{X_1} and L_{X_2} might overlap. Instead of storing directly L_{X_1} and L_{X_2}, one can store the union $L_{X_1} \cup L_{X_2}$ of L_{X_1} and L_{X_2} along with two bitmaps B_{X_1} and B_{X_2} on $L_{X_1} \cup L_{X_2}$ for L_{X_1} and L_{X_2} respectively. Bitmap B_{X_i}, $i = 1, 2$, has a '1' bit at position x iff L_{X_1} comprises the XML tree node at position x of $L_{X_1} \cup L_{X_2}$. This idea can be applied to multiple view node materializations resulting in important space savings. Note that because the view node materializations $L_{X_1}, \ldots, L_{X_k}$ of the view nodes $X_1, \ldots, X_k$ having the same label are sorted on the *begin* value of the positional representation of their XML tree nodes, the intersection $L_{X_1} \cap \ldots \cap L_{X_k}$ can be computed by merge-joining $L_{X_1}, \ldots, L_{X_k}$. Using bitmaps, the intersection of $L_{X_1}, \ldots, L_{X_k}$ can be computed by bitwise ANDing $B_{X_1}, \ldots, B_{X_k}$ which produces a bitmap of the intersection $L_{X_1} \cap \ldots \cap L_{X_k}$ on $L_{X_1} \cup \ldots \cup L_{X_k}$. That is, the order is preserved. Besides the important space savings, the use of bitmaps also offers time saving for two reasons: (a) fetching into memory bitmaps of view nodes and the inverted list nodes corresponding to their bitwise AND has less I/O cost than fetching the materializations of these nodes, and (b) bitwise AND-ing bitmaps has less CPU cost than merge-joining the corresponding view node materializations.



(a) Hit rate (b) Cache size

Figure 5: Hit rate and cache size with with increasing number of materialized views

7. EXPERIMENTAL EVALUATION

We implemented our approach and ran experiments to study its time and space performance and scalability. We also ran experiments to compare our approach with traditional approaches. As traditional approaches assume a different evaluation model and answer sets, this comparison makes sense when it concerns the view cache hit rate.

7.1 Experimental Setup

Our implementation was coded in Java. All the experiments reported here were performed on an Intel Core 2 CPU 2.13 GHz processor with 2GB memory running JVM 1.6.0 in Windows XP Professional. The Java virtual machine memory size was set to 512MB. Both XML inverted lists and TPQ view definitions as well as the view materializations were stored in a commercial DBMS. Each displayed time value in the plots is averaged over 5 runs with a cold DBMS buffer cache.

We ran experiments both on an XML benchmark data set generated using $XMark$ [26] and on a synthetic dataset using IBM's XML Generator [11]. We used a 56.2MB XML benchmark data set generated using $XMark$ [26]. This XML document does not include recursive elements. It contains 74 distinct element labels. The total number of parsed element nodes (excluding attributes and text values) is 832911 and the size of their positional representations (i.e., the inverted lists) is 15.1MB. We also ran experiments on a highly recursive synthetic dataset, whose results are similar to those reported here and are omitted in the interest of space.

We used the XPath generator $YFilter$ [10] to produce queries. $YFilter$ generates XPath queries according to specified parameters, such as the maximum query depth, the probability of descendant edges ($//$), and the probability of branches. In order to create more general workloads, we modified $YFilter$ in the following two ways: (a) we removed the limitation on supporting only one level of nesting of path expressions, so that it can generate complex XPath queries with arbitrary nesting, and (b) we relaxed the restriction on the axis of a predicate path expression which allows only child axes ($/$).

7.2 Hit Rate

We first compare the view cache hit rate of our approach with that of previous approaches. The hit rate expresses the percentage of randomly generated queries that can be answered using one or multiple views materialized in the view cache. In order to compare with previous approaches where queries have output nodes we use the criterion for query answerability using a set of views of [28] which requires that: (a) the output node of a view in the view set is mapped to an ancestor-or-self node of the query output node through a homomorphism (in which case we say that this query node is covered by the view), and (b) each query node which is not covered by this view is covered by some other view in the set.

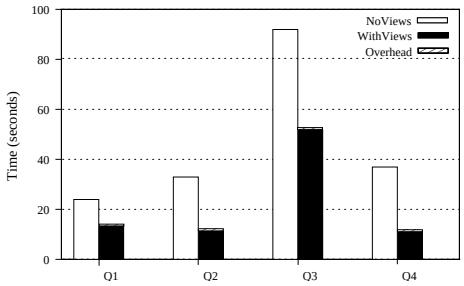
We generated a workload with 8000 views. We used the following setting for the workload: maximum view depth = 4, probability of descendant edges = 0.8, and probability of branches = 1. We also generated 100 random queries with the following setting: maximum query depth = 9, probability of descendant edges = 0.8, and probability of branches = 1. In the experiments, we scaled the number of views in the view pool from 1000 to 8000. To better illustrate the capacities of the different approaches under comparison, we also measured and compared the hit rate of these approaches when only one view can be used for answering the given query.

Figure 5(a) shows the hit rate of different approaches increasing the number of views in the view pool. We refer to our approach as MVIL (Materialized Views as Inverted Lists) and to the approach in [28] as MVST (Materialized Views as Subtrees). Our approach largely outperforms MVST both when one or multiple materialized views are used to answer the query. For the case of multiple views it outperforms MVST by at least 40% and achieves a hit rate of 97% for 7000 or more views in the view pool.

7.3 Space Performance

We also measured the space efficiency of our approach. We used the workload on the XMark dataset described above. Recall that the materialization of a view is stored as bitmaps, one per each view node. In addition, a set of inverted lists is stored, one inverted list per each distinct node label in the views of the view pool. Each such inverted list is the union of the materializations of all the view nodes with the same label in the view pool. We refer to this materialization scheme as *bitmap materialization scheme*. As a comparison, we also stored directly the materializations of the nodes of all the views and measured the total space used. We refer to the later scheme as *inverted lists materialization scheme*.

Figure 5(b) reports on the view cache size under the two materialization schemes as the number of materialized views increases from 1000 to 8000. The scale of the Y-axis is logarithmic. The total size of the view cache under the bitmap scheme rises from 26.45MB to 128.3MB as the number of views in the view pool increases from 1000 to 8000. In comparison, the size of the cache under the inverted lists scheme increases faster than the bitmap scheme from 305.8MB to 2563.63MB. Further, the inverted lists scheme consumes much more space, up to 20 times more than the



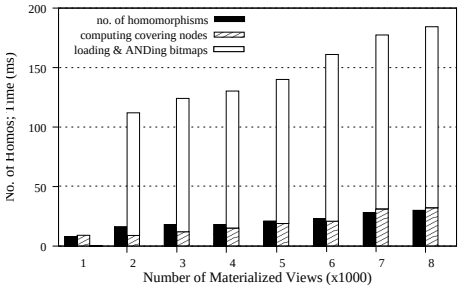
(a) Query processing time

	#INV	#SUB	#ANS
Q1	55854	29963	18977
Q2	77663	26393	4839
Q3	218024	120383	13230
Q4	79076	17084	3237505

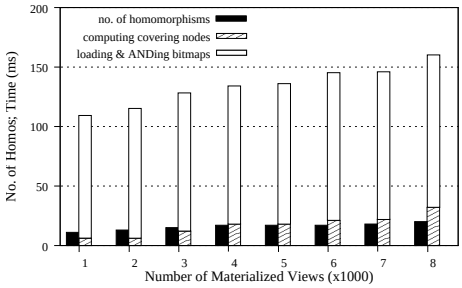
#INV: number of nodes in the inverted lists used for evaluating the query
#SUB: number of nodes in the inverted sublists (materialized views) used for evaluating the query using the views
#ANS: number of tuples in the query answer

(b) Query evaluation statistics

Figure 6: Query processing time and evaluation statistics on the XMark dataset



(a) Q_2



(b) Q_4

Figure 7: Computation overhead with increasing number of materialized views

bitmap scheme for most of the test cases.

Notice that the size of the bitmap materializations can be further reduced using state of the art bitmap compression techniques [30] without compromising the efficiency of bitwise logical operations. Such an implementation is beyond the scope of this paper.

7.4 Query Processing Time

We next show the speedup obtained in query evaluation time with our approach. We assume that the views are materialized in the client side while the base XML data is stored remotely in the server side. Queries are evaluated at the server side without using materialized views, while they are evaluated at the client side using exclusively the view materializations. In both cases the inverted lists evaluation model is adopted and the state of the art holistic algorithm $TwigStack$ [4] is employed. The communications costs are ignored. If these costs are taken into account the savings achieved by our approach are even larger. For the comparison, we used the workload of the 8000 materialized views described above. Among the 8000 views, 6605 have non-empty answers. We also used four test queries on the XMark dataset, which are shown in Figure 8. These queries are randomly generated and they can all be answered using exclusively the materialized views. Figure 6(a) reports on the query processing time per query for two different configurations: *NoViews* refers to evaluating queries on the server XML database without using materialized views. *WithViews* refers to answering queries using exclusively materialized views stored in the client view cache. *Overhead* denotes the computational overhead for using materialized views. It consists of the time needed for finding the covering view nodes of the query nodes and the time needed for loading in memory and bitwise ANDing the bitmaps of the node materializations.

As we can see from Figure 6(a), *WithViews* achieves significant speedup compared to *NoViews*: from 77% for Q_3 up to a factor of 2.3 for Q_4 (our experiments on a highly recursive dataset show a speedup by a factor of 13 for some queries). For each query, the fraction of *Overhead* in the total processing time using *WithViews* is very small, ranging from 0.34% for Q_3 to 1.73% for Q_2.

Q1	//site/people/person[.//interest]/name
Q2	//site/regions/namerica/item[.//quantity] [.//mail//to]/name
Q3	//open_auction[.//description[text//keyword]] [initial][quantity]//bidder/date
Q4	/site[.//person[.//creditcard]/address[country] //zipcode]//africa//incategory

Figure 8: Queries on the XMark dataset

Figure 6(b) shows the evaluation statistics of the four queries of Figure 8 over the XMark dataset. We observe that the query evaluation performance is largely determined by the number of inverted list nodes read from disk during execution, since each disk access triggers I/O whose cost dominates the computation costs of the query. As we can see in Figure 6(b), a query can be computed using substancially smaller inverted lists with our approach (column #SUB) than with the *NoView* approach (column #INV). For instance, the number of nodes accessed using materialized views is reduced by 78% for query Q_4 of Figure 8. This reduction in size, reduces the I/O cost, but it also reduces the CPU cost resulting in a substancial speedup.

7.5 Scalability

Finally, we measured the scalability of our approach as the number of the materialized views in the view pool increases. The scalability is examined in terms of the computation overhead which, as explained in Section 7.4, consists of two parts: (a) the time spent on finding all the query covering nodes in the view pool–this operation is done by the algorithm described in Section 5, and (b) the time spent on loading selected bitmaps from disk to memory and on bitwise ANDing bitmaps.

Figure 7 reports on both components of the computation overhead, as well as the number of homomorphisms from the view to

the query when the number of materialized views increases from 1000 to 8000 for two queries Q_2 and Q_4. Notice that the bitmap processing component is 0 for query Q_2 when the view pool contains 1000 views, since Q_2 has no hit on the view cache in this case. As expected, the number of homomorphisms for each query grows as the number of views increases. Both components of the overhead grow very smoothly. For instance, for query Q_4, the covering node computation component and the bitmap processing component for 1000 views are 6ms and 103ms, respectively. They grow to 32ms and 128ms for 8000 views (a ratio of 5.3 and 1.2 respectively). Note that using a bitmap compression technique [30] can further reduce the size of bitmaps and thereby the I/O cost for loading them in memory.

8. CONCLUSION

We have addressed the problem of answering XML queries using exclusively materialized views. We claim that previous approaches to this problem are limited by the way query answers (and view materializations thereof) are defined. To overcome these limitations, we have revised the problem by placing it the setting of the inverted lists model which is currently the prominent model for evaluating queries on large persistent XML data. In this context, we have suggested an original approach for materializing views which stores the inverted lists of only those XML tree nodes that occur in the answer to the view. To the best of our knowledge this is the first time the problem is addressed in this context and such a materialization scheme is adopted. We provided necessary and sufficient conditions for tree-pattern query answerability in terms of view to query homomorphisms. We designed a time and space efficient algorithm for deciding query answerability and we showed how queries can be computed over view materializations using stack-based holistic algorithms. We further developed optimization techniques that minimize the storage space and avoid redundancy by materializing views as bitmaps, and that optimize the evaluation of the queries over the views by applying bitwise operations on view materializations. Our experimental results showed that our approach has largely higher hit rates than previous approaches, significantly speeds up the evaluation of queries without using views, and scales very smoothly in terms of storage space and computational overhead.

Our approach can be directly applied to larger classes of XML queries. Even though the existence of homomorphisms might not be anymore a necessary condition for query answerability using views, it does constitute a sufficient condition. Therefore, homomorphisms can be used to compute covering view nodes for the nodes of these queries as in the case of tree-pattren queries described in the paper. We are currently working on exploiting our approach for optimizing queries using materialized views in centralized environments. In this framework, the focus is on answering partially a query using views and on view selection for materialization. We are also working on techniques for the efficient updating of the view materializations when the XML data is modified.

9. REFERENCES

[1] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

[2] A. Balmin, F. Özcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[3] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *ICDE*, 2003.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[5] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Answering regular path queries using views. In *ICDE*, 2000.

[6] B. Cautis, A. Deutsch, and N. Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.

[7] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, 1995.

[8] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on DAGs. In *VLDB*, 2005.

[9] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, 2002.

[10] Y. Diao and et al. YFilter. http://yfilter.cs.umass.edu/.

[11] A. L. Diaz and D. Lovell. IBM's XML generator. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[12] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, 2001.

[13] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(10):1381–1403, 2007.

[14] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[15] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1), 1982.

[16] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, 2003.

[17] L. V. S. Lakshmanan, H. Wang, and Z. Zhao. Answering tree pattern queries using views. In *VLDB*, 2006.

[18] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen. From region encoding to extended dewey: on efficient processing of XML twig pattern matching. In *VLDB*, 2005.

[19] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[20] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[21] M. M. Moro, Z. Vagena, and V. J. Tsotras. Tree-pattern queries on a lightweight XML processor. In *VLDB*, 2005.

[22] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD Conference*, 2006.

[23] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD*, 2003.

[24] D. Phillips, N. Zhang, I. F. Ilyas, and M. T. Özsu. Interjoin: Exploiting indexes and materialized views in XPath evaluation. In *SSDBM*, 2006.

[25] P. Rao and B. Moon. Prix: Indexing and querying XML using prüfer sequences. In *ICDE*, 2004.

[26] A. Schmidt and et al. XMark: An XML benchmark project. http://monetdb.cwi.nl/xml/.

[27] J. Tang and S. Zhou. A theoretic framework for answering XPath queries using views. In *XSym*, 2005.

[28] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[29] J. Wang and J. X. Yu. XPath rewriting using multiple views. In *DEXA*, 2008.

[30] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 2006.

[31] X. Wu, S. Souldatos, D. Theodoratos, T. Dalamagas, and T. Sellis. Efficient evaluation of generalized path pattern queries on XML data. In *WWW*, 2008.

[32] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[33] B. Yang, M. Fontoura, E. Shekita, S. Rajagopalan, and K. Beyer. Virtual cursors for XML joins. In *CIKM*, 2004.

[34] C. Yu and L. Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD Conference*, 2004.