Optimizing XML Queries: Bitmapped Materialized Views vs. Indexes

Xiaoying Wu[a], Dimitri Theodoratos[a,*], Wendy Hui Wang[b], Timos Sellis[c]

[a]*New Jersey Institute of Technology, University Heights, Newark New Jersey 07102, USA*
[b]*Stevens Institute of Technology, Castle Point on Hudson, Hoboken New Jersey 07030, USA*
[c]*Institute for the Management of Information Systems, Greece*

Abstract

Optimizing queries using materialized views has not been addressed adequately in the context of XML due to the many limitations associated with the definition and usability of materialized views in traditional XML query evaluation models.

In this paper, we address the XML query optimization problem using materialized views in the framework of the inverted lists evaluation model which has been established as the most prominent one for evaluating queries on large persistent XML data. Under this framework, we propose a novel approach which instead of materializing the answer of a view materializes exactly the sublists of the inverted lists that are necessary for computing the answer of the view. A further originality of our approach is that the view materializations are stored as compressed bitmaps. This technique not only minimizes the materialization space but also reduces CPU and I/O costs by translating view materialization processing into bitwise operations. Our approach departs from the traditional approach which identifies a compensating expression that rewrites the query using the materialized views. Instead, it computes the query answer by executing holistic stack-based algorithms on the view materializations. We experimentally compared our approach with recent outstanding structural summary and B-tree based approaches. In order to make the comparison more competitive we also proposed an extension of a structural index approach to resolve combinatorial explosion problems. Our experimental results show that our compressed bitmapped materialized views approach is the most efficient, robust, and stable one for optimizing XML queries. It obtains significant performance savings at a very small space overhead and has negligible optimization time even for a large number of materialized views in the view pool.

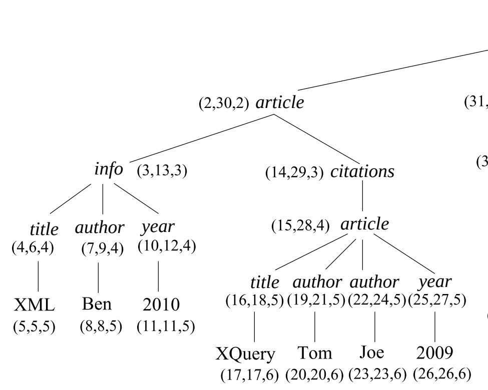
Keywords: XML, XPath query evaluation, optimization of tree-pattern queries using views, bitmap materialized views

1. Introduction

Using materialized views has been along with indexing one of the best known techniques for optimizing the evaluation of queries. The main idea is that precomputing (materializing) a number of views and storing their materializations in a view pool might be beneficial to the evaluation of queries. The expectation is that the effort in computing the views, captured in the view materializations, can be exploited during the evaluation of queries in order to reduce their evaluation cost.

In the context of the Relational model the use of materialized views for answering and optimizing queries has been studied extensively [1, 2] and integrated into commercial DBMSs [3, 4, 5, 6] in past years. In the context of XML, the number of contributions on these issues has been restricted. This is due to the fact that the use of materialized views for answering queries is limited when the traditional approach is used for: defining XML query answers (and consequently view materializations), and for evaluating a query using materialized views. In this paper, we use an original approach for materializing views, and for optimizing queries using materialized views.

*Corresponding author
Email addresses: `xw43@njit.edu` (Xiaoying Wu), `dth@cs.njit.edu` (Dimitri Theodoratos), `hui.wang@stevens.edu` (Wendy Hui Wang), `timos@imis.athena-innovation.gr` (Timos Sellis)



(a) An XML tree

article = {(2,30,2), (15,28,4), (31,40,2), (41,53,2)}

author = {(7,9,4), (19,21,5), (22,24,5), (36,38,4), (46,48,4)}

info = {(3,13,3), (32,39,3), (42,52,3)}

year(2010) = {(10,12,4), (25,27,5), (49,51,4)}

(b) inverted lists

Figure 1: An example XML tree and some of its inverted lists

We compare our optimization technique under the same evaluation model.

Traditionally, e.g. in XPath, the answer of a query expression with a single output node against an XML document tree is the set of *subtrees* rooted at the matches of the query output node in the XML tree. Consider, for instance, the example XML tree with bibliographic information shown in Figure 1(a). The labels of element nodes are shown *emphasized* as opposed to those of value nodes. The triplets annotating the nodes enumerate them according to a regional encoding scheme [7, 8]. Figure 2 shows an XPath query Q and two XPath views V_1 and V_2 (represented as tree patterns). Single line edges represent child relationships while double line edges represent descendant relationships. A node annotated with a star ('*') indicates the output node of a query. Query Q asks for the authors of cited articles published in 2010. View V_1 computes the authors of articles published in 2010 or of articles citing an article published in 2010. View V_2 computes the citing articles. As an example, one can see that view V_2 has one match to the XML tree. Therefore its answer consists of the subtree rooted at node (15, 28, 4) labeled *article*.

In the traditional approach, queries are evaluated using the materialized views by finding a *compensating* query which is a *rewriting* of the original query using materialized views [9, 10, 11, 12, 13, 14]. This compensating query is then evaluated over the materialized views and possibly the input XML document.

Limitations of the traditional approach. The traditional approach has three major drawbacks: (a) The absence of complete structural information outside the subtrees in the view materializations greatly restricts the chances of a query to have a rewriting using views in the pool of views that are materialized for optimizing the query, (b) The view materializations are unindexed fragments of the XML document usually making the computation of a query more expensive compared to computing it against the original XML document, and (c) The size of the answer subtrees can be very large; when multiple views are materialized (and inevitably overlapping portions of the XML document are repeatedly and redundantly stored), view materialization becomes unfeasible due to space limitations. Reducing the materialization space by selectively materializing views [11, 15] further reduces the chances of the query to have a hit in the view pool and consequently its chances to get an optimized evaluation plan using the materialized views.

In the example of Figure 2, one can see that query Q cannot be answered using the materializations of both views V_1 and V_2 but also that it cannot be answered using each one of them. That is, Q cannot be rewritten using V_1 or V_2 and therefore these views cannot be used to optimize the evaluation of Q. The reason is that the structural restrictions of Q cannot be expressed on the subtrees rooted at the *author* nodes and the lower *article* nodes that are returned by the views V_1 and V_2. In contrast, as we show later, query Q can be answered using V_1 or V_2 and can even be answered using exclusively V_1 and V_2 in our novel context of view materialization, this way greatly reducing the evaluation time of Q. In [16], it is shown experimentally that the hit ratio of XML queries in a view pool is much higher for the new type of materialized views we introduce than for traditional materialized views.

Materializing additional information besides the subtrees e.g. ancestor path information, typed val-

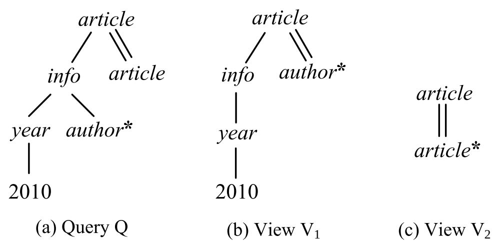


Figure 2: A query and two views

article	info	year(2010)	author
(2,30,2)	(3,13,3)	(10,12,4)	(7,9,4) Ben
(2,30,2)	(3,13,3)	(10,12,4)	(19,21,5) Tom
(2,30,2)	(3,13,3)	(10,12,4)	(22,24,5) Joe
(41,53,2)	(42,52,3)	(49,51,4)	(46,48,4) Jim

(a) Materialization of V_1 as a set of tuples

article = {(2,30,2), (41,53,2)}
info = {(3,13,3), (42,52,3)}
author = {(7,9,4), (19,21,5), (22,24,5), (46,48,4)}
year(2010) = {(10,12,4), (49,51,4)}

(b) Materialization of V_1 as a set of sublists

article = 1001
info = 101
author = 11101
year(2010) = 101

(c) Materialization of V_1 as a set of bitmaps

Figure 3: Materializations of View V_1 of Figure 2(b) on the XML tree of Figure 1(a)

ues and references to XML data [9] only partially addresses the issues mentioned above for the traditional approach while increasing the size of data that need to be stored.

The framework of our optimization techniques. There are two major paradigms of XML query processing techniques: the relational and the native. The relational paradigm stores XML data in relational databases and transforms queries over XML data into SQL queries over relational data [17, 18, 19]. The relational paradigm can leverage existing techniques of relational DBMSs for query processing and optimization. In contrast, the native paradigm uses specialized storage and query processing systems which are tailored for XML data and are developed from scratch. In this paper, we focus on a native XML query processing approach. In the context of the native paradigm, 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 [7, 8]. 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. Figure 1(b) shows the inverted lists for some of the labels in the XML tree of Figure 1(a). As is usually the case with the inverted list evaluation model [7], we assume that there is an index structure that identifies the nodes in an inverted list that satisfy a given predicate. This is the case with the inverted list of label *year* which is restricted to the year of 2010.

All the relevant query evaluation algorithms in this evaluation model are based on stacks. Comparison studies on XML query evaluation techniques [20, 21] show that holistic stack-based algorithms [7, 22, 8, 23, 24, 25, 26, 27] in the inverted lists model are superior to other algorithms and evaluation models (streaming/navigational approaches [28] or sequential/string matching approaches [29]). The framework of our approach is the inverted lists evaluation model along with holistic stack-based evaluation algorithms. Note that in the inverted lists model, the answer of a query Q is not a subtree of the XML tree but a set of tuples having one field for every node in Q. Each tuple contains the (positional representation of) the XML tree nodes that match the query nodes in an embedding of the query to the XML tree. Figure 3(a) shows the answer of view V_1 on the XML tree of Figure 1(a) as a set of tuples. Notice that if the answer of a view is directly materialized as a set of tuples nodes can be redundantly stored multiple times.

In order to compute Q on an XML tree (i.e. set of inverted lists), for every node X in Q labeled by a label a, all the nodes of the inverted list L_a for a are fetched and processed for a possible participation in the answer of Q as values of X. When it can be determined in advance that some nodes in L_a do not contribute to the answer of Q as values for X, fetching and processing these nodes can be avoided thus saving precious time. This is the reason many approaches focus on exploiting indexes (e.g., XB trees [7] or structural indexes [20, 30]) in order to *filter out irrelevant nodes*. The approach we present in this paper uses materialized views in an original way in order to filter out irrelevant nodes.

Problems addressed. We address the problem of optimizing queries using views materialized as compressed bitmaps of inverted sublists. We assume that the base data (inverted lists of the input XML tree) are available locally and therefore the materialized views can be used inclusively (that is, along with base data) or exclusively (that is, without base data) for answering the queries. In this framework, the problems that we face when we try to optimize queries using materialized views are the following:

(a) Given a set of views, can we decide whether a query can be answered using (inclusively or exclusively) the views and how? Can this check be performed efficiently?

(b) If a query can be answered using a set of materialized views, how this computation can be performed efficiently?

(c) Given a query and a pool of materialized views how can we compute efficiently a set of materialized views that can be used for optimally answering the query?

(d) How much time is it worth spending in finding such sets of views?

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

- We suggest a novel approach for view materialization where instead of storing the answer of a view as a set of tuples, we store for every view node exactly the sublist of XML nodes necessary for computing the answer of the view. This way, an exponential number of solutions can be stored in polynomial space. Further, the problem of redundantly storing multiple instances of the same node which is linked to the tuple-based materialization of views is addressed. In order to minimize the storage space, view materializations are represented as compressed bitmaps. Figures 3(b) and (c) depict the materialization of view V_1 of Figure 2(b) as a sets of sublist and as a set of bitmaps, respectively.
- We consider tree-pattern queries and views and we specify necessary and sufficient conditions for answering a query using inclusively or exclusively one or multiple materialized views in terms of homomorphisms from the views to the query. These results are novel since they apply to our new concept of view materialization. For instance, these conditions allow us to deduce in our running example that query Q of Figure 2 can be answered using exclusively the materializations of views V_1 and V_2 of the same figure.
- In order to check the answerability of a query using views, we present 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 enumerating a possibly exponential number of homomorphisms.
- We show how a query can be computed using inclusively or exclusively views in the framework of bitmap materialized views by running state of the art holistic stack-based algorithms over the materializations of the views. View materialization processing (e.g. inverted sublist intersections) is translated into bitwise operations over bitmaps this way reducing not only its CPU cost but also its I/O cost.
- We show that if multiple view sets can be used for answering a query, answering the query using their union minimizes the query execution cost. Further, the cost for finding a maximal set of views that can be used for answering a query (optimization cost) is practically not significant compared to the benefit obtained from using a maximal set of views.
- We compared our approach with other approaches aiming at improving the inverted lists evaluation model by filtering out irrelevant nodes in advance. We considered both: an approach based on XB trees and one on structural indexes. We focused in particular on the existing structural index approach and identified a major limitation of it. To make the comparison more competitive, we proposed an important extension that addresses the problem and we included this extended structural index approach in the comparison.
- We implemented and tuned five approaches for optimizing XML queries and conducted a thorough and extensive experimentation to compare their performance. We investigated and analyzed the behavior of each approach for evaluating simple and complex queries on real, benchmark and synthetic datasets. Our experimental results show that our bitmapped view materialization approach is the most efficient, robust, and stable one for optimizing queries on XML. It obtains significant performance savings at a small space overhead and negligible computation overhead. Further, it scales very smoothly in terms of space and optimization time when the number of materialized views in the view pool increases.

- To the best of our knowledge, our paper is the first one that combines materialized views as inverted sublists of XML tree nodes and compressed bitmap techniques for optimizing XML queries, and takes advantage of these concepts for saving storage space and query evaluation time.

Paper outline. The next section reviews related work. Section 3 describes the framework of our optimization approaches, and introduces our concept of bitmapped materialized view. Section 4 presents results on deciding whether a query can be answered using views. Section 5 presents an algorithm that is used for efficiently checking view usability and for efficiently evaluating a query using views. Optimizing queries using bitmapped materialized views is discussed in Section 6, while an improvement of the approach that is based on structural indexes is presented in Section 7. Our comparative experimental results are presented and analyzed in Section 8. Section 9 concludes and suggests future work. Proofs of theorem and propositions are provided in the Appendix.

2. Related Work

In recent years, significant effort has been devoted to developing high-performance XML query processing techniques. These techniques can be categorized into two major classes: the relational approach and the native approach [21]. A number of research projects have considered the relational approach and addressed storing and querying XML data in RDBMSs [17, 18, 19]. In this paper we adopt the inverted lists query evaluation model which accounts as a native approach. Due to lack of space, we therefore provide here a brief review on the state-of-the-art tree-pattern query (TPQ) evaluation and optimization techniques for the native approach. We focus on both: index-based techniques, and view-based techniques. A detailed survey on the query processing techniques for the two approaches is provided in [21]. It is worth noting that many of the techniques developed for the native approach can also be adapted to the relational paradigm.

We provide now a brief review on the state-of-the-art XML tree-pattern query (TPQ) evaluation and optimization techniques. We focus on both: index-based techniques, and view-based techniques. Cost-based query optimization techniques for XML investigating the applicability of relational query optimization techniques to answer TPQs [31, 32] are orthogonal to the optimization techniques studied in this paper.

Index-based Techniques. In [20], the authors compared existing TPQ evaluation techniques (including the holistic twig join algorithm [7, 8], the streaming or navigational technique [31, 28], and the sequential technique [33, 29]) in the framework of the inverted lists evaluation model and quantified their relative advantages and disadvantages. Their performance study showed that the family of holistic evaluation algorithms is the most robust and effective method in this framework.

The approaches that speed up the processing of the original holistic evaluation algorithm TwigStack [7] by skipping unnecessary nodes build indexes on the input inverted lists to define node clusterings and/or orderings. They can be classified into the following two categories.

The first category comprises approaches built upon the conventional B^+-tree technique. It includes the B^+-tree [22], the XB-tree [7], and the XR-tree [34, 8]. A study in [35] compares the performance of the three B^+-tree based techniques on evaluating binary query patterns. This performance comparison is extended in [20] to tree-pattern queries. The experimental results of both papers show that XB-Tree has better performance than the other two techniques. For this reason, in this paper, we choose XB-tree as a representative of the B^+-tree based techniques for evaluating TPQs.

The other category consists of solutions which combine structural indexes with inverted lists to support XML query evaluation [36, 30, 37, 20, 38]. Structural indexes are constructed using the concept of bisimilarity [39, 40]. By partitioning the input XML data nodes according to their structural properties, the size of the resulting structural index is usually smaller than the original XML data. Consequently, the query evaluation conducted directly on the structural index is expected to be more efficient than on the input data itself. The experimental results presented in [20, 30] show that the structural index approach performs better than the index-based techniques PRIX [29] and ViST [33].

The original structural index approach usually starts by computing all the embeddings of the given query to the structural index and then evaluates the query by considering each embedding separately As shown by our experiments in Section 8, this approach exhibits exponential behavior when the input query has a large number of embeddings to the structural index. To address this problem, in this paper, we propose an improved structural index approach which extends the original one in different important ways.

View-based Techniques. A number of papers have recently addressed the important problems of XML query rewriting using views and of XML view selection for materialization [41, 9, 42, 43, 11, 10, 44, 13, 45, 46, 14]. 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 [9, 11, 10, 43, 13]. Balmin et al. [9] 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 depends on the availability of one or more of the four types of materializations. Mandhani and Suciu [11] presented results on equivalent TPQ rewritings when the TPQs are assumed to be minimized. Xu et al. [10] studied the equivalent rewriting existence problem for three subclasses of TPQs. Tang and Zhou [43] considered rewritings for TPQs with multiple output nodes. However, the rewritings are restricted to those obtained through output node dependent homomorphisms from the view to the query. The problem of maximally contained TPQ rewritings was studied in [47] both in the absence and presence of a schema and more recently in [48]. All contributions in [9, 11, 10, 43, 47] 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 XML queries using multiple views has been studied in [13, 45, 46, 14, 49]. Arion et al. [13] considered the problem in the presence of structural summaries and integrity constraints. As in [43], 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 subtrees is also adopted by [45, 46, 49]. All these papers assumed that output node dependent homomorphisms exist among queries and views and they presented rewriting algorithms which use intersection of view answers on node ids.

Tang et al. [14] addressed the multiple view rewriting problem based on the assumptions that structural ids in the form of extended Dewey codes [50] 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 [42, 44] the equivalent rewriting problem has been addresses but for queries and views which are XQuery expressions. Elghandour et al. [51] addressed the problem of selecting relational materialized views for XQuery queries. Techniques for using relational views to rewrite XQuery queries were presented in [52].

Our approach presented in this paper differs from all the previous approaches. Note that structural encodings of XML data nodes are also employed in [13, 14]. However, unlike our approach, [13, 14] store node encodings together with view materializations (XML tree fragments) and use them mainly for combining answers of multiple views to produce query answers.

Phillips et al. [53] consider materializing intermediate query results as sets of tuples in order to allow additional evaluation plans for structural joins.

Materializing views as sets of tuples suffer from the problem of redundantly storing XML tree nodes, an issue we have very successfully addressed in this paper. Further, the proposed technique only works for path queries and path views. Moreover, their context of view usability is very restricted and they do not address query answerability from materialized views issues.

Chen et al. [54] proposed a materialization scheme for XML views that stores inverted sublists for the view nodes. Unlike our approach, that approach stores, in addition, the precomputed structural joins for views in the form of pointers that link nodes in the inverted sublists. A query is computed by traversing the pointers of the materializations. The main drawback of the pointer-based scheme, though, is its space requirement since the pointers consume large amounts of storage space. The problem is exacerbated when the same structural join which is involved in multiple materialized views is redundantly stored in the view cache. Our approach is more general and flexible than the pointer-based scheme in terms of view usability. By materializing views as compressed bitmaps, it minimizes the storage space and avoids redundancy.

A data pre-filtering technique called XML document projection [55, 56] aims at pruning all data that are certain to be irrelevant to the evaluation of a given query. The goal is similar to that of many approaches that adopt the inverted lists evaluation model [7, 22, 8, 23, 24, 25, 26, 27]. However, the techniques of [55, 56] are designed for main-memory XQuery processing. In contrast to approaches that adopt the inverted lists evaluation model and employ for the evaluation of a query only the relevant inverted lists, [55, 56] have to read a large part of the XML tree in order to select the subtree that will be stored in memory for the evaluation of the query. Further, since only subtrees of the input XML tree can be pruned, internal irrelevant nodes that could be filtered out are missed.

The scheme of materializing views as inverted sublists was initially presented in [16]. Our results on view usability in Section 4 and the algorithm that computes covering nodes in Section 5 were first reported there. Our focus in this paper is the optimization of queries using views materialized as compressed bitmaps. Therefore, we assume a centralized environment where the inverted lists are available locally and the materialized views can be used inclusively for answering the queries. Our approach is compared experimentally with other optimization approaches. In contrast, [16] focuses on answering XML queries using materialized views in a distributed environment where the access to the base data (inverted lists) is not possible. Therefore, queries need to be answered using exclusively materialized views. Since this is a paper on optimizing queries using views we do not need to find all the homomorphisms from the views to the queries because the base data (inverted lists) are available locally and all the queries can be answered using the base data. The goal is to see how many of them we need to identify in order to minimize the evaluation cost of a query. This is in contrast to [16] where all the homomorphisms need to be identified since the base data are not available locally and the goal is to answer the queries using exclusively the materialized views. As also shown experimentally in the same section, the time needed for finding homomorphisms is less than the benefit we obtain in the evaluation cost by the additional covering nodes. Therefore, even when some thousands of bitmap views are materialized it is worth spending time trying to find all the covering nodes.

Since this is a paper on optimizing queries using views we do not need to find all the homomorphisms from the views to the queries because the base data (inverted lists) are available locally and all the queries can be answered using the base data. The goal is to see how many of them we need to identify (in other words, how many covering nodes for the query nodes we need to find) in order to minimize the evaluation cost of a query. Note that this is in contrast to our previous CIKM conference paper where all the homomorphisms need to be identified since the base data are not available locally and the goal is to answer the queries using exclusively the materialized views. However, as we explain in Section 8.6, the more covering nodes we compute, the more we reduce the size of the inverted lists used for computing a query (until the minimum-size inverted lists that contain only the nodes necessary for computing the query are obtained). As also shown experimentally in the same section, the time needed for finding covering nodes is less than the benefit we obtain in the evaluation cost by the additional covering nodes. Therefore, even when some thousands of bitmap views are materialized it is worth spending time trying to find all the covering nodes.

3. Data Model, Query and View Language, Evaluation Model and View Materializations

In this section, we outline the data model, and the class of queries and views we consider, and the inverted lists evaluation model we adopt.

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. We denote by $\mathcal{L}$ the set of XML tree node labels. We use lower case letters for XML tree labels and subscripts to distinguish different nodes with the same label.

For XML trees, we adopt the region encoding widely used for XML query processing [7, 8]. 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 $n_2.end < n_1.end$. Node n_1 is the parent of n_2 iff $n_1.begin < n_2.begin$, $n_2.end < n_1.end$, and $n_1.level = n_2.level - 1$.

Query and view language. In order to expose the novel features of our approach without getting lost into the technical details introduced by more complex queries, we consider that queries and views are tree-pattern queries (TPQs). 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. This constitutes an important advantage of our approach compared to previous ones since it allows the exploitation of views when other approaches fail.

A *tree-pattern query* (TPQ) specifies a pattern in the form of a tree. A TPQ Q comprises nodes and child and descendant relationships between nodes. 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. 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 is not restricted. Any kind of query can be a view.

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. During evaluation, the entries in stack S_X correspond to nodes in L_A before C_X. 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 scans the inverted list L_Y we say that node X is *computed on L using the list L_Y*.

Materialized Views as Compressed Bitmaps 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 $V(L)$.

Observe that, 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 [9, 11, 47, 13, 14]. Note that our approach is space efficient since the sublists can encode in linear space an exponential number of solutions for the view.

The materialization L_X of a view node X labeled by a on L can be represented by a bitmap on L_a that has a '1' bit at position x iff L_X comprises the XML tree node at the position x of L_a. Then, the materialization of a view is the set of the bitmaps of its nodes. The bitmaps are stored compressed to even further reduce the materialization space. Clearly, storing the materialization of multiple views as compressed bitmaps results in important space savings. However, as we explain in Section 6, the use of bitmaps also offers CPU and I/O cost savings.

4. Answering Queries Using Materialized Views

In this section, we present necessary and sufficient conditions for answering a query using inclusively or exclusively one or multiple materialized views. Then, we show how queries can be computed efficiently using materialized 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. The materializations of these nodes in $V(L)$ can then be used when 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 using exclusively *V. Otherwise, we say that Q can be answered using* inclusively *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 4 shows a query Q and a view V and four homomorphisms h_1, h_2, h_3 and h_4 from V to Q. For the needs of checking homomorphism existence, a new root r linked through a double edge to the current root should be equivalently added to a query if not already there.

The following theorem relates node coverage to homomorphisms.

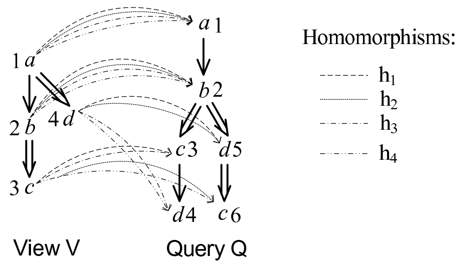


Figure 4: Four homomorphisms from view V to query Q

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.*

The proof follows directly from Definition 4.1. In the example of Figure 4, 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 [11, 10, 47]. This increases the chances for a homomorphism from a view to a query to exist. Based on Theorem 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 using exclusively 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 using exclusively 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.*

The proof is again a direct consequence of Definition 4.1. Based on Corollary 4.2, one can easily see that in the example of Figure 4, query Q can be answered using exclusively view V.

Computing the Answer of a Query Using One View. In the traditional approach to answering a query using a view [9, 11, 10, 13, 14], the query is rewritten using the view. That is, in order to compute the answer of the query, a compensating query is identified which is applied to the materialized view. This compensating query computes the answer of the query by navigating in the view materialization which is a set of subtrees of the original XML tree.

In contrast, in our approach, we do not need a compensating query and a rewriting of the query using the views, but we 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 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

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

For instance, in the example of Figure 4, we consider four homomorphisms from view V to query Q and two of them map view node d to query node d_5. Therefore, the set of the covering nodes for query node d_5 in V is $m(d_5) = \{d\}$.

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

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

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

Figure 5 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 a query using exclusively views we can state the following theorem.

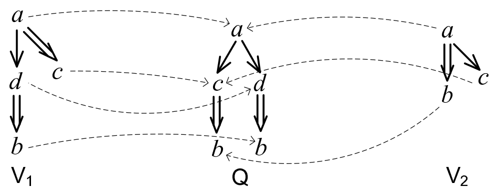


Figure 5: Query Q and views V_1 and V_2 and homomorphisms

Theorem 4.3. *Let Q be a query and $\{V_1, \dots, V_n\}$ be a set of views. Query Q can be answered using exclusively $V_1, \dots, V_n$ if and only if 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 5 can be answered using exclusively 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 what is needed is an association of query nodes with 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 $m_1(X), \dots, m_n(X)$ be the sets of covering nodes of X in $V_1, \dots, V_n$, respectively. Then, the set $m(X)$ of covering nodes of X in $V_1, \dots, V_n$ is

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

For instance, in the example of Figure 5, we consider one homomorphism from each one of the views V_1 and V_2 to query Q. Therefore, the set of the covering nodes for query node d in V_1 and V_2 is $m(d) = \{d[V_1], d[V_2]\}$.

As with the case of a single view, if $\exists X \in Q$, $m(X) \neq \emptyset$, Q can be answered using $V_1, \dots, V_n$. If $\forall X \in Q$, $m(X) \neq \emptyset$, Q can be answered using exclusively $V_1, \dots, V_n$. 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 $m(X)$.

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

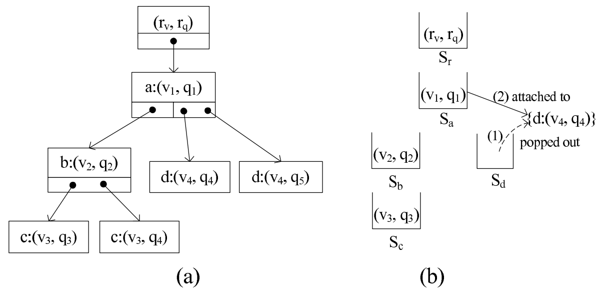


Figure 6: (a) The match structure for the view and the query of Figure 4, (b) The snapshots of stacks after the query leaf node d has been visited during the execution of Algorithm $computeCovering$

are defined in terms of the homomorphisms of 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 Structure. In the algorithm we use a tree structure, called *match structure*, which is similar to the one employed in [57, 58, 59].

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. The match structure $MatchStru(V, Q)$ is a tree whose nodes are pairs of nodes (v, q) such that v matches q and there is a homomorphism from V to Q that maps v to q. The root of $MatchStru(V, Q)$ is (r_V, r_Q), where r_V and r_Q are virtual roots of V and Q respectively, labeled by a distinguished label r. A node (q, v) in $MatchStru(V, Q)$ has a child node (v_i, q_j) if v_i is a child of v in V and q_j is a descendant of q in Q. $MatchStru(v, q)$ denotes the subtree of $MatchStru(V, Q)$ rooted at (v, q). Given a node v, $MatchSet(v_i)$ denotes the set of subtrees of $MatchStru(V, Q)$ rooted at a node (v_i, q_j) which is a child of a node (v, q) of $MatchStru(V, Q)$.

The match structure for the view V and query Q of Figure 4 is graphically shown in Figure 6(a). In order to uniquely identify a node of the view or the query, every node of the view and the query in Figure 4 is associated with a node id.

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 encode all the homomorphisms from V to Q in the form of match structures (line 2). In the second step, the match structures are traversed in a top-down manner to determine the covering view nodes (lines 3-5).

Procedure $constructMS$ traverses the tree pattern Q in preorder, constructing the matching structures as it visits nodes and traverses edges. When $constructMS$ visits a query node for the first time, it creates a match structure for each matching view node. The created match structures are possibly pushed onto stacks. When $constructMS$ returns to a query node after traversing the entire subtree of this node, it determines whether the match structures created for the query node should be inserted into appropriate $MatchSets$ of nodes in selected match structures in the stacks. When $constructMS$ finishes the traversal of Q, $MatchStru(r_V, r_Q)$ encodes all the homomorphisms from V to Q. We describe the process below in more detail.

Initially, a matching structure $MatchStru(r_V, r_Q)$ 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 be the parent node of v and p be the query node of the match structure corresponding to the top entry of stack S_u. $constructMS$ checks whether the structural relationship between p and q in Q satisfies the structural relationship between u and v in V. If this is the case, a match structure $MatchStru(v, q)$ and the $MatchSet$ of every child of v in V is initialized to be empty. The created match structure is 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 $MatchStru(v, q)$ from stack S_v (lines 10-11). In order to determine whether $MatchStru(v, q)$ should be added to stacks, it suffices to check whether all the $MatchSet$s of $MatchStru(v, q)$ are non-empty. If $MatchStru(v, q)$ has to be added to the stacks, for each entry in stack S_u, where u is the parent of v, a pointer to $MatchStru(v, q)$ is created and added to the entry's $MatchSet(v)$ (lines 12-15).

Figure 6(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 structure is popped out from the stack S_d of view node d. Since it has to be added to the stacks, it is attached to the only match structure in the 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 of $prefix(V, v)$ 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, and the total number of match structures constructed during execution is bounded by $|V| \times |Q|$. The number of incoming pointers to each constructed match structure 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 structures). 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 the match structure for v and q should be added to the stacks 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 $O(|V| \times |Q| \times D)$.

Clearly, computing the covering set $m(X)$ of the nodes X in a query Q based on all the views from the view pool that can be used for answering Q minimizes the cost of evaluating Q using views from the view pool. The reason is that the inverted sublists produced for the nodes X of Q by intersecting the materializations of the nodes in $m(X)$ are the minimal possible ones. This answers question (c) raised in the introduction asking for an efficient way for identifying a set of materialized views in the view pool that can be used for optimally answering the query.

Listing 1 Algorithm computeCovering

```
create a stack for each node of $V$
constructMS(root(Q))
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 $false$
for (every $ms \in MatchStru(r_V, r_Q).MatchSet(root(V))$) do
    traverse $ms$ in a top down manner: for each $MatchStru(v, q)$ encountered, if $visited[v, q]$ is $false$, then add $v$ to $m(q)$, set $visited[v, q]$ to $true$, and continue the traversal on each match structure in $MatchSet(v)$
```

Listing 2 Procedure constructMS(q)

```
for (every $v \in nodes(V)$ that matches $q$ considered in post-order) do
    let $u$ be the parent of $v$ in $V$
    if (stack $S_u$ is not empty) then
        let ($u$,$p$) be in the top entry of $S_u$
        if ($v$'s axis is '//' or ($q$ is a child of $p$ and $q$'s axis is '/')) then
            create $MatchStru(v, q)$ and initialize its $MatchSets$ to be empty
            push $MatchStru(v, q)$ to stack $S_v$
for (every child $q'$ of $q$ in $Q$) do
    constructMS(q')
for (every $v \in nodes(V)$ that matches $q$ considered in pre-order) do
    pop out the top entry $e$ from stack $S_v$
    if ($e$ has to be added to the stacks) then
        let $u$ be the parent of $v$ in $V$
        for (every stack entry $e' \in S_u$) do
            add a pointer to $e'$ that points to $e$
```

6. The Bitmapped Materialized Views Approach

The bitmapped materialized views approach for optimizing queries assumes that a set of views are materialized as compressed bitmaps in a view pool. In order to compute the answer of a query, it uses the algorithm of Section 5 to compute, for every query node q, all the covering view nodes in the view pool. Then, it intersects the inverted sublists of these covering view nodes. The resulting sublist is used for the computation of q. If q does not have covering nodes, the corresponding inverted list is used for its computation. This approach takes advantage of some additional optimization improvements:

Bitwise operations. The intersection of the inverted sublists of the covering view nodes can be implemented by a bitwise operation on the corresponding bitmaps: first, the bitmaps of the operand view nodes are fetched into memory and bitwise AND-ed. Then, the target inverted sublist is constructed by fetching into memory the inverted list nodes indicated by the resulting bitmap. Besides space savings, exploiting bitmaps and bitwise operations results in time saving for two reasons. First, bitwise AND-ing bitmaps incurs less CPU cost than intersecting the corresponding inverted sublists. Second, fetching into memory the bitmaps of the operand view nodes and the target inverted sublist nodes indicated by the resulting bitmap incurs less I/O cost than fetching the entirety of the inverted sublist of the operand view nodes as this is required for applying directly an intersection operation.

In the following we assume that the materialized views are sets of inverted sublists represented by bitmaps on the corresponding inverted list.

Empty Covering Nodes Materializations. The materialized views approach evaluates a query by first computing covering view nodes. If the materialization of a covering view node or even the intersection of the materializations of some covering view nodes of a query node is empty the query has an empty answer. Based on this observation, this approach stops the query evaluation as soon as such a condition is satisfied. This way, substantial disk I/Os and CPU cost can be saved.

Avoiding Processing Redundant Query Path Solutions. Holistic evaluation algorithms such as *TwigStack* [7] operate in two steps. In the first step, they generate a list of path solutions for each individual root-to-leaf query path of a query Q. In the second step, they merge join the path solutions to produce the final solutions of Q. In the first step, *TwigStack* sequentially reads nodes of the inverted lists only once. For every node x (in the inverted list of node X in Q) that is pushed on its stack, *TwigStack* [7] ensures the following properties: (a) x has a *descendant* node on each of the inverted lists corresponding to the child nodes of X in Q (regardless of whether the relationship to X is a child or descendant), and (b) each of the descendant nodes recursively satisfies this property. Because of this, when all edges in Q are descendant relationships, each node pushed onto its stack is guaranteed to participate in a final solution of Q and every path solution generated is guaranteed to merge join with other path solutions to generate final solutions for Q. In this case, *TwigStack* is CPU and I/O optimal for computing Q.

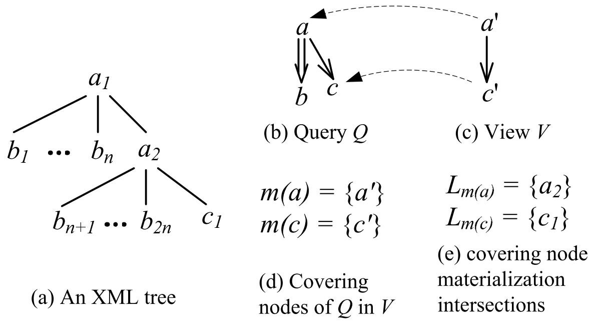


Figure 7: An example for Proposition 6.1

However, as shown in [7], these properties do not hold if Q has at least two leaf nodes and contains a child edge. Choi et al. [60] showed that any version of *TwigStack* that sequentially reads inverted lists only once cannot be optimal for TPQs with arbitrarily mixed child and descendant edges. That is, *TwigStack* may generate redundant path solutions (path solutions that do not contribute to a final solution of Q) when queries contain child edges. For instance, consider evaluating the query of Figure 7(b) on the XML tree of Figure 7(a). *TwigStack* cannot determine whether node a_1 has a child c-node after c_1 in the inverted list of label c. As a result, it will generate $2n$ redundant path solutions $(a_1, b_1), \ldots, (a_1, b_{2n})$ for the query path $a//b$. None of these paths solutions will join with the single solution (a_2, c_1) generated for the query path a/c to produce a solution to Q. Clearly, the existence of redundant path solutions impacts the query performance negatively, since these path solutions are processed and possibly joined with other path solutions without producing a final solution to the query.

Research efforts have focused on avoiding redundant path solutions for some subclasses of TPQs. For example, [23] achieves this for TPQs whose child edges are under non-branching nodes by looking ahead in the inverted lists and caching the nodes. Chen et al. [61] do so for TPQs with child edges only or for TPQs involving one branching node only by considering a partition of the XML tree nodes more refined than label-based inverted lists. As shown in the following proposition, under certain conditions, when *TwigStack* is employed to evaluate queries using materialized views, redundant path solutions are avoided.

Proposition 6.1. *Let Q be a query which is com-*

puted on an XML tree T using the materialized views $V_1, \ldots, V_n$. Let also for every child edge X/Y in Q, Y be a leaf node in Q. Algorithm TwigStack will produce no redundant path solutions if for every child edge X/Y in Q, node Y has a covering node in some view V_i that also has a child incoming edge.

Query Q in Figure 7(b) satisfies the condition of Proposition 6.1 for queries. Consider also the view V of Figure 7(c). Query Q and view V satisfy the conditions of Proposition 6.1. Therefore, *TwigStack* will produce no redundant path solutions when evaluating Q using the materialization of V. Indeed, looking at the covering nodes of nodes a and c of Q in V shown in Figure 7(d) and the covering node materialization intersections shown in Figure 7(e), we can see that node a_1 is not in $L_{m(a)}$ and thus no redundant path solutions involving a_1 for the query path a/b will be produced.

7. Improving the Structural Index Approach

Structural Indexes. Given a partitioning of the nodes of an XML tree T based on an equivalence relation on its nodes, a structural index for T is a graph G such that: (a) every node in G is associated with a distinct equivalence class of element nodes in T, and (b) there is an edge in G from the node associated with the equivalence class $\mathcal{A}$ to the node associated with the equivalence class $\mathcal{B}$, iff there is an edge in T from a node in $\mathcal{A}$ to a node in $\mathcal{B}$. The equivalence class of nodes in T associated with each node in G is called *extent* of this node. Structural indexes have been referred to with different names in the literature and they differ in the equivalence relations they employ to partition the nodes of the XML tree (e.g., they are called structural summaries in [20, 38], path indexes or path summaries in [30, 37]). Structural indexes have been used as a back-end for XML query processing (i.e., queries are evaluated on the structural indexes alone). The majority of recent works on exploiting structural indexes for evaluating queries [36, 30, 20, 38] considers an approach that combines structural indexes with inverted lists to support XML query evaluation.

An often-used structural index is called *1-index* [39, 40]. A 1-index considers as equivalent nodes in T that have the same incoming path from the root of T. A 1-index is a tree representing a summarization of the paths that actually occur in a XML tree[1]. It can be constructed by traversing the given XML tree once, and it is usually much smaller than the corresponding XML data. Based on measurements on XML documents from different repositories , a 1-index is three to five orders of magnitude smaller than the corresponding XML data [37]. Also, the experimental results reported in [38] show that 1-indexes outperform approximate structural indexes (e.g., the A(k) indexes [63] which record only an approximation of the paths—up to depth k) in terms of both space and query evaluation time. For these reasons, in this paper, we choose 1-indexes as the representative structural index for query processing.

The Structural Index Approach. The structural index approach usually processes a given query Q in two steps. In the first step, in most existing realizations [30, 20], it computes the embeddings of Q to G. Because the size of a 1-index is usually small, the cost of this step is not important and even a naive approach would be satisfactory for finding the embeddings.

In the second step, for every embedding e of Q to G, Q is evaluated against the extents of the images of its nodes under e. Usually, a holistic twig join algorithm such as *TwigStack* [7] is employed for performing this evaluation. The solutions obtained from each evaluation are unioned to compute the answer of Q. Algorithm *TwigStack* can be used with structural indexes because the nodes in the extents are represented by their positional encoding and are ordered on their *begin* value (Section 3).

When Q has a very small number of embeddings to G, and Q is very selective, the structural index approach can greatly reduce the CPU and disk-read cost compared to the inverted lists approach. The reason is that the structural index makes it possible to skip a large number of nodes that do not participate in the query solutions. Recall that the inverted lists approach partitions nodes of the XML tree T based on their labels so that nodes in an inverted list have the same label. In contrast, in the structural index approach, each structural index node is associated with an extent which contains all the nodes that have the same incoming (label) path from the root in T. That is, the structural index approach refines the label-based partitioning of the nodes of T. Because

[1]1-indexes are similar to *strong DataGuides* [62] when the data is a tree.

the partitioning is usually much more refined, the size of the extents is much smaller than that of inverted lists. This results in significant savings when evaluating a single TPQ using the structural index approach compared to the inverted lists approach.

A Problem with the Structural Index Approach. The structural index approach has a drawback: Q can have many embeddings to G. In fact, the number of embeddings can even be exponential on the number of nodes in Q. Clearly, it is expensive to evaluate multiple TPQs and union their solutions in order to compute the answer of Q. Further, a big part of Q may be mapped to the same part of G under different embeddings. This overlapping of the images of Q results in redundant computations since extents associated with the overlapping parts have to be scanned multiple times in the second step of the query evaluation process described above. These repeated computations can significantly increase both the CPU and the disk-read cost. Our experiments in Section 8 confirm this observation.

The Improved Structural Index Approach. To avoid redundant computations, we extend the original structural index approach in the following two important ways.

First, we design an algorithm which, without enumerating all the embeddings of Q to G, computes for each query node in Q its image nodes on G under all the possible embeddings of Q to G. Similar to the covering algorithm described in Section 5, this algorithm works by performing a single streaming (preorder) traversal of G, and encodes the embeddings in a space-efficient way, thus avoiding a combinatorial explosion of the image nodes.

Second, in order to scan the extents of the image nodes in G of a node in Q only once, for each node in Q, we logically union these extents and make these unions the input to algorithm $TwigStack$. The logical union operation is implemented by a priority queue with each item in the queue being a cursor to one of the extents. During the execution of $TwigStack$, each priority queue returns the node with the minimal *begin* value among the nodes referenced by the cursors at that time and subsequently the queue is updated.

Note that our approach of computing query nodes images to a structural index bears similarity to the approach presented in [37] for computing "relevant path sets" of query nodes to a structural index. However, the computed relevant path sets there are not used directly for evaluating the input query. But rather, they are used for constructing query plans which consist of binary structural joins [64] for the query.

Analysis of the Improved Structural Index Approach. Let M denote the average number of image nodes in G per node in Q, and $|Q|$ denotes the number of query nodes in Q. The number of embeddings of Q to G is $O(M^{|Q|})$. For each embedding, all the nodes in the extents of the images of the query nodes are accessed. The number of extents scanned is $O(M^{|Q|} \times |Q|)$. This is the reason that the original structural index approach is expensive. On the other hand, in the improved structural index approach, the extents of the images of each query node are accessed only once, and for each node accessed in the extents, the updating cost of the corresponding priority queue is $O(log(M))$. The number of extents scanned is $O(M \times |Q|)$. Clearly, the updating overhead of the improved structural index approach is very small compared to the cost of scanning the same node $O(M^{|Q|})$ times as is the case with the original structural index approach. Our experimental evaluation in Section 8 confirms this analytical result.

8. Experimental Evaluation

We compare our bitmapped materialized views approach with our improved structural index approach and with other previous approaches. Even though all these approaches employ different techniques, they have a common denominator which is that they aim at filtering out, in advance, nodes of the inverted lists that do not participate in the answer of the query. This way, the processing of these nodes is avoided. For the comparison, we implemented the following approaches: (1) the inverted lists approach with the $TwigStack$ algorithm [7] (denoted INV), (2) the $TwigStack$ algorithm with the XBTree index [7] (denoted XB), (3) the structural index approach which evaluates the embeddings of the query to the structural index separately [30, 20] (denoted $SIemb$), (4) our improved structural index approach which logically unions structural index node extents (Section 7) (denoted $SIlu$), (5) our approach which materializes for every view the inverted sublists for its nodes that are needed for computing the view answer (denoted $MVlist$), and (6) our bitmapped materialized views approach (Section 6) (denoted $MVbit$).

The *INV*, *SIemb*, *SIlu*, *MVlist*, and *MVbit* approaches are independent of the algorithm used to evaluate the queries. We have chosen algorithm *TwigStack* [7] for implementation convenience. Other stack-based holistic algorithms (e.g. $Twig^2Stack$ [65]) can be used as well.

The performance of all the approaches is measured in terms of three metrics: (a) the total evaluation time required for computing the query, (b) the total number of input nodes accessed, and (c) the number of page I/Os used.

We also provide a comparison of our approach with an approach which is implemented over a commercial RDBMS and can use indexes and materialized views.

8.1. Experimental Setup

Our implementation was coded in Java. All the experiments reported here were performed on an Intel Core Duo CPU 3.16 GHz processor with 4GB memory running JVM 1.6.0 in Windows 7 Professional. Each displayed time value in the plots is averaged over the values obtained from five runs.

Datasets. To analyze the behavior of each approach under different scenarios, we ran experiments with three datasets which have different structural properties. The statistics of the datasets are shown in Figure 8. The first one is a real dataset, the DBLP dataset[2] collected in September 2009. The DBLP dataset is flat, shallow and bushy. It contains a few fairly regular structural patterns. The second one is a benchmark dataset using *XMark*[3] with $factor = 5$. It is deep and has many regular structural patterns. Both DBLP and XMark datasets include very few recursive elements. The third one is a synthetic dataset generated by IBM's XML Generator[4] based on the DTD shown in Fig. 9. The generator was configured with $NumberLevels = 8$ and $MaxRepeats = 7$, where *NumberLevels* is the maximum number of levels in the resulting XML tree and *MaxRepeats* is the maximum number of times a child element characterized with the * (zero or more occurrences) or the + (at least one occurrence) option will be repeated at a node. By construction, this dataset comprises highly recursive and irregular structures. The statistics for the structural indexes (1-indexes) of the three datasets are also shown in Figure 8. The XB-tree index [7] was built by bulk loading XML data. The sizes shown in Figure 8 reflect the size of the XML documents. The inverted lists used in the experiments contain only element nodes and their sizes are shown in Figure 12.

	DBLP	*XMark*	*Synthetic*
XML document size	632MB	568MB	582MB
#nodes	15397K	8157K	16519K
#labels	35	74	27
Max/Avg depth	6/3	12/5.6	9/8.9
#1-index nodes	144	514	3075

Figure 8: Dataset statistics

```
<!ELEMENT R (a, b, c)+>
<!ELEMENT a (b*, c*, d*, e*, f*, m*, n*, s*, t*)>
<!ELEMENT b (a*, c*, g*, h*, i*, o*, p*, u*, v*, z*)>
<!ELEMENT c (a*, b*, j*, k*, l*, q*, r*, w*, x*, y*)>
<!ELEMENT d (#PCDATA)>
                    ...
<!ELEMENT z (#PCDATA)>
```

Figure 9: The DTD for the synthetic dataset

8.2. View and Query Generation

View Generation. We used the XPath generator *YFilter* [66] to produce views. *YFilter* uses the DTD of the XML document to generate views 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 so that it is not only a child axis (/).

The DBLP and XMark datasets are essentially non-recursive, and for almost every element in their DTD graph, all the paths from the root to this element have the same length. Under these conditions, for almost every pair of elements A and B, if A/B is satisfiable w.r.t the DTD, a query of the form A/B has the same answer as the query $A//B$ on an XML tree complying to the DTD. Since all the views generated by *YFilter* are satisfiable w.r.t the input DTD, we can restrict the views generated to those that comprise only descendant axes, without missing almost no useful view. We did not generate either views with only one non-root node since in almost all cases

[2]http://dblp.uni-trier.de/xml/
[3]http://monetdb.cwi.nl/xml/
[4]http://www.alphaworks.ibm.com/tech/xmlgenerator

	DBLP	XMark	Synthetic
Depth	3	5	3
Prob. of desc. edges	1	1	0.8
Prob. of branches	0.8	0.8	0.8
# Materialized views	2024 (773 have solutions)	3369	407

Figure 10: Parameters for view generation on the three datasets

the materialization of the non-root node is identical to the corresponding inverted list which is available in the database anyway.

The settings for the workload generation and the number of materialized views used for each dataset are shown in Figure 10. The number of unique XPath views to be generated for each of the three datasets was set to 10000. For the DBLP and the synthetic datasets, only 3872 and 785 distinct views respectively could be generated. The XMark dataset has the largest DTD in terms of elements, and was able to generate 10000 views.

Note that many XPath queries generated by $YFilter$, although represented by different XPath expressions, correspond to the same TPQ if all the nodes are viewed as output nodes. For this reason, in the experiments, we post-processed the XPath views generated by $YFilter$ by identifying and eliminating duplicate views. After this processing, for the three datasets, DBLP, XMark and synthetic, we obtained 2024, 7676, and 407 views, respectively. Note also that among the 2024 views used for the DBLP dataset, only 773 have non-empty answers, while the views for the other two datasets all have non-empty answers.

In order to further restrict the number of the XMark views while retaining those that are more likely to be useful for answering queries, we imposed size restrictions. We retained only XMark views with the following two properties: (1) the height of the corresponding TPQ is less than 4, and (2) the total number of root-to-leaf paths is less than 4. After this pruning, the final number of views used on the XMark dataset was 3369.

Queries. As with views, we used $YFilter$ [66] to generate random queries but with different parameters than those used for the generation of views. For each of the three datasets, five queries out of 100 randomly generated queries were used for the experiments. They are shown in Figure 11. The following comments can be made:

First, most of the queries generated for the DBLP dataset are highly selective. In particular, queries

		Query
DBLP	DQ_1	//article[.//month]/title
	DQ_2	//phdthesis[chapter]//number
	DQ_3	//book[.//author][.//journal]/ee
	DQ_4	//inproceedings[.//author][.//crossref]//month
	DQ_5	//proceedings[.//address][.//year]//editor
XMark	XQ_1	//description[./text]//parlist//listitem
	XQ_2	//namerica//item[description]//quantity
	XQ_3	//europe//item[.//incategory][.//location]//name
	XQ_4	//closed_auctions/closed_auction[.//type]//seller
	XQ_5	/site[.//description[.//text//keyword]]//person[.//name]//homepage
Synthetic	SQ_1	//c[.//b][y]/x
	SQ_2	//b[.//a[t]//c]//u
	SQ_3	//a[.//b[z]/o]//s
	SQ_4	//a[.//b//h][e]//f
	SQ_5	//b[.//a//s][.//c//j]/i

Figure 11: Queries used in the experiments

DQ_3, DQ_4, and DQ_5 have empty answers. This is due to the fact that most of the cardinality constraints on the DTD elements are optional. In contrast, queries generated for both the XMark and the synthetic datasets are non-selective having a large number of solutions.

Second, all the queries for the DBLP dataset have at most one embedding to the corresponding 1-index tree. With the exception of queries XQ_1 and XQ_5, the rest of XMark queries have only one embedding to the XMark's 1-index tree. Each of the queries on the synthetic dataset, however, have a large number of embeddings. This is due to the fact that the synthetic dataset contains highly complex and recursive structures, where a query node can have multiple images occurring both in the same and in different tree paths.

8.3. Space Usage

We compare the disk space usage of the six approaches INV, XB, $SIemb$, $SIlu$, $MVlist$ and $MVbit$. Figure 12 reports on the space consumed by each one of them on the three datasets considered. The basic approach INV consumes the least space since no additional structures are used. Both $SIemb$ and $SIlu$ use slightly more space than INV because of the small space overhead incurred by the use of a more refined node partitioning. Among the six approaches, $MVlist$ requires the largest amount of space for storing the inverted sublist materializations for views. XB requires the second largest amount of space consumed for storing the XB-tree index.

The space used by $MVbit$ consists of two parts: (1) the space for storing the inverted lists of the corresponding dataset, which is the same as that of INV,

	DBLP	XMark	Synthetic
INV	242	128.3	260
XB	373.3	197.8	400.5
SIemb & SIlu	242.2	128.7	262.7
MVlist	2291.8	12029.8	8477.3
MVbit	244.7(2.7)	151.1(22.8)	296(36)

Figure 12: Space usage (in MB) of each approach. The size of bitmapped view materializations is shown in parentheses.

	uncompressed (MB)	compressed (MB)	compression ratio
DBLP	170	2.7	1.6%
XMark	187.6	22.8	12.2%
Synthetic	122.5	36	29.4%

Figure 13: The compression ratio of bitmapped view materializations on the three datasets

and (2) the space for storing view materializations. Recall that the view materializations are stored as bitmaps, one per each view node. In order to reduce the space used for view materializations, we compressed the bitmapped materializations using the Java zip package. Figure 13 shows the sizes of view materializations before and after the compression and the compression ratios of the bitmapped materializations for the three datasets.

The space usage of view materializations is shown within parentheses next to the total space usage of $MVbit$ in Figure 12. The ratios of the view materialization space over the total space used by $MVbit$ on the DBLP, XMark, and the synthetic datasets are 1.1%, 15.1%, and 12.2%, respectively. As we will show later, the small extra space usage of $MVbit$ compared to that of INV, $SIemb$, and $SIlu$, is compensated by the speedup on the query evaluation.

8.4. Query Performance

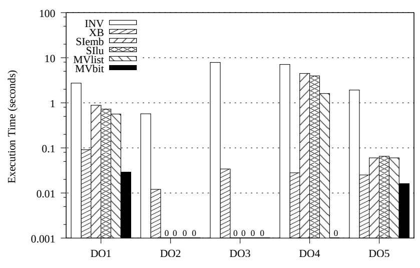
We compare the query performance of the six approaches INV, XB, $SIemb$, $SIlu$, $MVlist$ and $MVbit$. The performance is expressed by the query evaluation time, which is the total time required by each algorithm to compute the query answer. The total number of nodes accessed by each algorithm for computing a query is the principal determining factor of the query evaluation time. It also determines the total number of I/Os performed by each algorithm. Figure 14 reports on the query performance results for the three datasets (notice the logarithmic scale used for the Y-axis in Figure 14(a)). The number of nodes accessed by each algorithm as a percentage of the number of nodes accessed by INV are shown in Figures 15(a) and 15(b), respectively.

8.4.1. The Bitmapped Materialized Views Approach

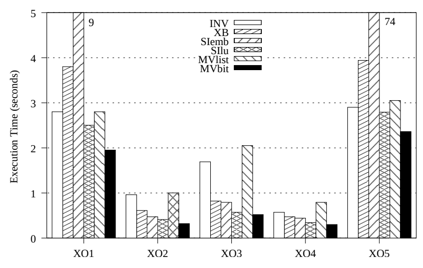
As it can be observed in Figure 14, $MVbit$ performs better than the other approaches on all the testing cases. In particular, on the DBLP dataset, it outperforms INV by orders of magnitude. Notice also that $MVbit$ is able to determine that queries DQ_2 and DQ_3 have empty answers almost immediately after they are issued because of covering views (views having homomorphisms to the query) having empty answers or because of empty covering view node materializations intersections (Section 6). These significant performance savings are obtained at a very small space overhead which is due to the view materializations. As Figure 12 shows, on the DBLP dataset, the space used by $MVbit$ exceeds that of INV by only about 1.1%. Further, the performance of $MVbit$ is stable, and does not degrade with more complex queries and on data with highly recursive structures (Figure 14(c)).

Note that the query evaluation time of $MVbit$ consists of the *optimization time* and the query *execution time*. The query execution time is the time needed for computing the query using the view materializations. The optimization time consists of the time needed for finding the covering view nodes of the query nodes and the time needed for disk loading, decompressing and bitwise ANDing the bitmaps of the node materializations. In [67], a bitmap compression technique is developed which allows bitwise logical operations to be performed directly on compressed bitmaps. We have not pursued this direction further in this paper as our experimental results show that the optimization time is already very small: the average optimization time over all three datasets is less than 8% of the query evaluation time.

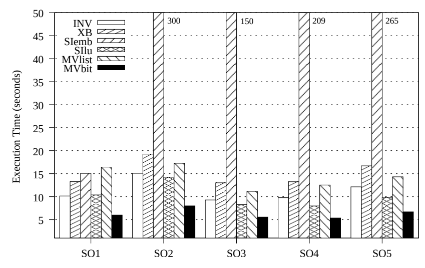
Quick identification of candidate views. The views in the view pool that contain a label not occurring in a query Q cannot be used for answering Q. Our experimental results show that the average numbers of non-relevant views per query for the DBLP, XMark and the synthetic datasets are 67%, 49%, and 47%, respectively. In order to speed up the computation of the covering view nodes of Q, we maintain in memory the definition of every materialized view. Even so, it is time consuming to apply the covering algorithm to all views if their number is very large. For this reason, we construct an in-memory view index which allows us to quickly filter views that cannot



(a) DBLP dataset



(b) XMark dataset



(c) Synthetic dataset

Figure 14: Query evaluation time

	INV	*XB*			*SIemb*	*SIlu*	*MVlist*	*MVbit*
		internal	leaves	total				
DQ_1	100	0.05	0.58	0.63	39.06	39.06	39.43	0.3
DQ_2	100	0.04	0.02	0.07	0	0	0.00	0
DQ_3	100	0.03	0.02	0.05	0	0	0.00	0
DQ_4	100	0.03	0.02	0.06	63.91	63.91	70.49	0
DQ_5	100	0.05	0.06	0.11	3.94	3.94	0.75	0
XQ_1	100	1.59	100	101.59	434.35	90.99	134.39	64
XQ_2	100	0.81	29.66	30.47	27.33	27.33	214.35	27.33
XQ_3	100	0.85	26.16	27.01	23.41	23.41	244.77	23.41
XQ_4	100	0.91	54.95	55.87	54.93	54.93	292.96	54.93
XQ_5	100	1.47	92.59	94.06	1915.77	81.13	102.49	77.52
SQ_1	100	1.59	99.87	101.45	126.266	70.31	389.02	60.57
SQ_2	100	1.59	99.86	101.45	1920.3	51.27	325.41	52.84
SQ_3	100	1.59	99.69	101.28	1378.68	46.48	223.21	45.99
SQ_4	100	1.59	99.79	101.38	1904.95	34.42	284.66	46.25
SQ_5	100	1.59	99.83	101.41	1995.6	29.9	275.55	38.52

(a) Percentage of nodes accessed per approach

	INV	*XB*	*SIemb*	*SIlu*	*MVlist*	*MVbit*
DQ_1	19569	489	7644	7644	7800	116
DQ_2	3499	7	0	0	0	0
DQ_3	41924	78	0	0	0	0
DQ_4	41954	94	26815	26815	31594	0
DQ_5	10220	50	404	404	1632	5
XQ_1	9059	36796	39498	8276	17467	9059
XQ_2	4290	5226	1174	1174	9189	1177
XQ_3	6802	7345	1595	1595	16623	1595
XQ_4	2082	4646	1144	1144	6090	1146
XQ_5	11992	45090	230040	9788	14587	11104
SQ_1	27230	110492	35566	19404	191152	27227
SQ_2	36606	148533	720173	19036	200869	36562
SQ_3	31936	129359	448994	15074	136624	31883
SQ_4	32046	129939	630893	11239	164317	31939
SQ_5	41050	166502	846118	12528	196024	40672

(b) Number of I/Os per approach

Figure 15: Percentage of nodes accessed and number of I/Os per approach

be used to compute Q.

Given a set of n views, for every distinct node label in the views, the view index maintains a list of views having a node with that label. The view index is implemented as a bitmap on the n views. The lookup of Q in the index consists of two steps. The first step computes the bitwise OR of the bitmaps of all the labels that do not occur in Q. The resulting bitmap denotes those views that cannot be used to compute Q. Then, the second step takes the negation of the resulting bitmap which denotes the views that are potentially useful for computing Q. The lookup time of a query in the view index is negligible, and this helps making the cost for computing covering nodes very small.

8.4.2. The Inverted Sublists Materialized Views Approach

Given a query Q, $MVlist$ uses the same procedure as $MVbit$ to compute the covering nodes of Q in the views. The inverted sublists of the covering nodes are then used to compute Q. When a node X in Q has more than one covering nodes, the inverted sublist used to computed X is the intersection of the inverted sublists of X's covering nodes. Since nodes of the inverted sublists are stored on disk, the CPU time and I/Os for performing the intersection are proportional to the sum of the length of each individual inverted sublist. This sum is likely even larger than the length of the inverted list of X itself. In this case, $MVlist$ needs to read more nodes than INV for computing Q. This is confirmed by the experimental results. As we can see in Figure 14, INV outperforms $MVlist$ for queries on both the XMark dataset and the synthetic dataset. In contrast, using the bitmapped materialization of views, the $MVbit$ approach can dramatically reduce space but also save substantially disk I/Os and CPU cost.

8.4.3. The Index-Based Approaches

In contrast to $MVbit$, the performance of index-based approaches, including XB, $SIemb$, and $SIlu$, varies greatly across queries and datasets used. We analyze the performance of each index-based approach below.

The XB-tree Approach. XB largely outperforms INV for the queries on the DBLP dataset, and in fact it achieves the second best result for most queries on this dataset. The reason is that the DBLP publications are clustered by their types, such as article, book, inproceedings, etc. This clustering property allows XB to prune significant portions of the input data for queries on specific type of publications, especially when such queries are selective. For instance, on evaluating query DQ_3, the percentages of leaf and internal XB-tree nodes visited by XB over the total input inverted list nodes are 0.02% and 0.03%, respectively (Figure 15(a)).

The performance advantage of XB over INV on the XMark dataset is less prominent. XB is even outperformed by INV for queries on the synthetic dataset, where it virtually loses the node-pruning capability and actually needs to visit more nodes for the query evaluation than INV (Figure 15(a)). The reason is that for both datasets, query solutions are dispersed throughout the entire dataset. This is especially the case for the synthetic dataset, which is randomly generated and has highly recursive structures. In those situations, XB has to go deep in the XB tree and down to the leaves in many cases. In the worst case, XB needs to traverse the entire XB tree to compute the query solutions.

The structural index approaches. Both the structural index approaches $SIemb$ and $SIlu$ use a 1-index (a structural index) for query evaluation. As mentioned in Section 7, a 1-index makes it possible for the structural index approaches to skip a large number of nodes that do not participate in the query solutions. Also, they help to detect queries with empty answers thereby stopping their evaluation at an early stage. For instance, consider the queries DQ_2 and DQ_3. Both queries have no embeddings to the 1-index of the DBLP dataset and hence have empty answers. The structural index approaches do not further process them. However, not every query with an empty answer can be detected by the structural index approaches without evaluation. For instance, the query DQ_4 has an embedding to the 1-index of the DBLP dataset, and each of its individual root-to-leaf paths has solutions on the data. Nevertheless, the query itself does not have solutions on the dataset but this cannot be detected by the structural index approaches. In contrast, the $MVbit$ approach is able to determine that DQ_4 has empty answer after the optimization phase without executing the query.

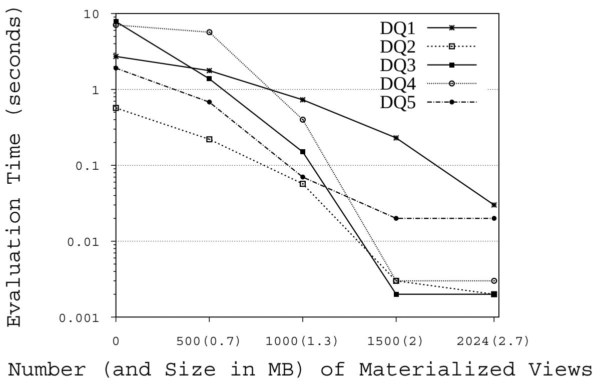
The query performance of the traditional structural index approach $SIemb$ depends largely on the queries and the datasets used. More specifically, the performance of $SIemb$ is greatly affected by the number of the embeddings of the input query to the 1-index. When the query has at most one embedding, as is usually the case with the DBLP dataset, $SIemb$ largely outperforms INV (Figure 14(a)). However, when the number of embeddings is large, as is usually the case with the synthetic dataset, $SIemb$ is largely outperformed by INV, in many cases, by more than one order of magnitude (Figure 14(c)). For instance, $SIemb$ has a poor performance with query XQ_5 on the XMark dataset because XQ_5 has 81 embeddings to the 1-index. This poor performance of $SIemb$ is even more noticeable with queries SQ_2, SQ_4 and SQ_5 on the synthetic dataset which have 5640, 5927 and 6725 embeddings, respectively, to the 1-index.

The improved structural index approach $SIlu$ addresses the problem of $SIemb$. In many cases, it outperforms the INV approach and has the second best performance following $MVbit$. However, when the number of image nodes of the input query on the 1-index is very large, the cost of performing a logical union of the extents of the image nodes (Section 7) can offset the savings obtained by the filtering of irrelevant data nodes. For instance, consider query SQ_2 on the synthetic dataset. Although $SIlu$ skips about 50% of the input nodes compared to INV (Figure 15(a)), with an average number of 106 images per query node, the query processing time of $SIlu$ is only slightly better than that of INV (Figure 14(c)).

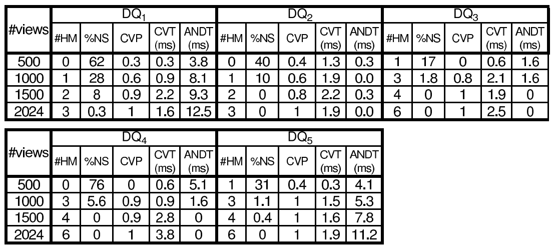
8.5. Query Performance vs. Materialization Space

We also measured how the number of materialized views in the view pool (space allocated for view materialization) affects the query execution time. We gradually increased the number of materialized views and measured the execution time, and the number of inverted list nodes accessed as a percentage of nodes accessed by the INV approach.

In order to randomize the distribution of the views we produced ten different random orderings of all the generated views and incrementally selected the same number of views from each one of them averaging the measured evaluation times for the queries. We also collected evaluation statistics, including the number of computed homomorphisms of the views



(a) Query evaluation time



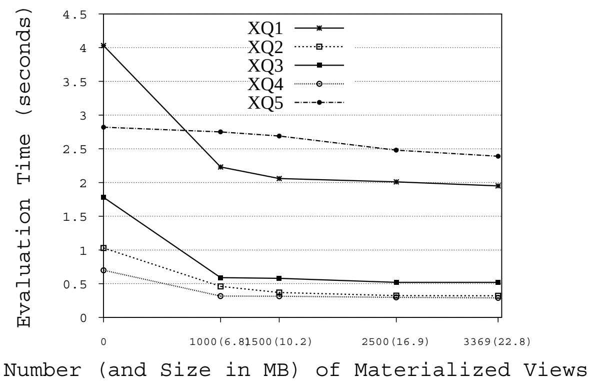
#views	DQ$_1$					DQ$_2$					DQ$_3$				
	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)
500	0	62	0.3	0.3	3.8	0	40	0.4	1.3	0.3	1	17	0	0.6	1.6
1000	1	28	0.6	0.9	8.1	1	10	0.6	1.9	0.0	3	1.8	0.8	2.1	1.6
1500	2	8	0.9	2.2	9.3	2	0	0.8	2.2	0.3	4	0	1	1.9	0
2024	3	0.3	1	1.6	12.5	3	0	1	1.9	0.0	6	0	1	2.5	0

#views	DQ$_4$					DQ$_5$				
	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)
500	0	76	0	0.6	5.1	1	31	0.4	0.3	4.1
1000	3	5.6	0.9	0.9	1.6	3	1.1	1	1.5	5.3
1500	4	0	0.9	2.8	0	4	0.4	1	1.6	7.8
2024	6	0	1	3.8	0	6	0	1	1.9	11.2

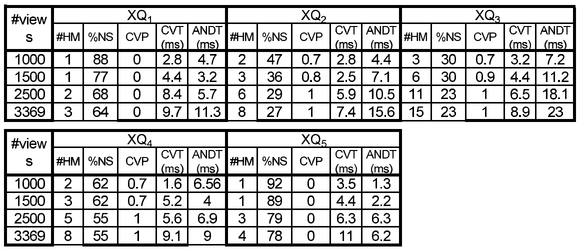
#views: no. of views in the view pool
#HM: avg. number of computed homomorphisms from the views to the query
%NS: avg. percentage of inverted lists nodes scannded
CVP: avg. probability for the query to be completely covered by the views in the view pool
CVT: avg. time for computing covering view nodes
ANDT: avg. time for disk loading, decompressing and bitwise ANDing bitmap materializations

(b) Query evaluation statistics

Figure 16: Query evaluation time and statistics by incrementally loading view materializations for the DBLP dataset



(a) Query evaluation time



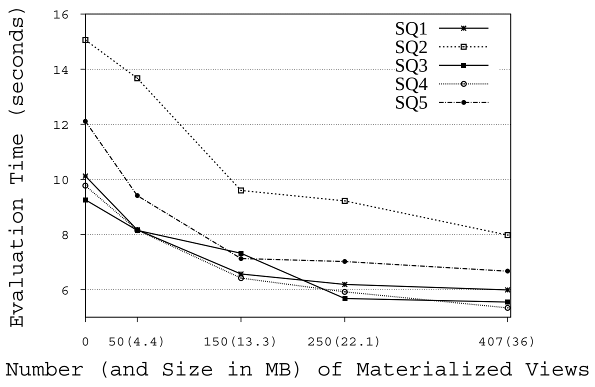
#views	XQ$_1$					XQ$_2$					XQ$_3$				
	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)
1000	1	88	0	2.8	4.7	2	47	0.7	2.8	4.4	3	30	0.7	3.2	7.2
1500	1	77	0	4.4	3.2	3	36	0.8	2.5	7.1	6	30	0.9	4.4	11.2
2500	2	68	0	8.4	5.7	6	29	1	5.9	10.5	11	23	1	6.5	18.1
3369	3	64	0	9.7	11.3	8	27	1	7.4	15.6	15	23	1	8.9	23

#views	XQ$_4$					XQ$_5$				
	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)
1000	2	62	0.7	1.6	6.56	1	92	0	3.5	1.3
1500	3	62	0.7	5.2	4	1	89	0	4.4	2.2
2500	5	55	1	5.6	6.9	3	79	0	6.3	6.3
3369	8	55	1	9.1	9	4	78	0	11	6.2

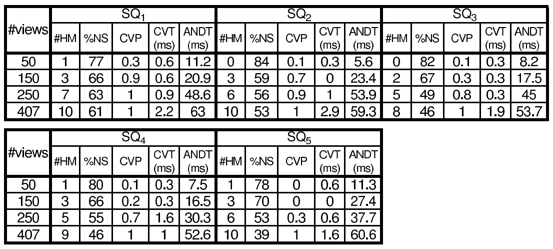
#views: no. of views in the view pool
#HM: avg. number of computed homomorphisms from the views to the query
%NS: avg. percentage of inverted lists nodes scannded
CVP: avg. probability for the query to be completely covered by the views in the view pool
CVT: avg. time for computing covering view nodes
ANDT: avg. time for disk loading , decompressing and bitwise ANDing bitmap materializations

(b) Query evaluation statistics

Figure 17: Query evaluation time and statistics by incrementally loading view materializations for the XMark dataset



(a) Query evaluation time



#views	SQ$_1$					SQ$_2$					SQ$_3$				
	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)
50	1	77	0.3	0.6	11.2	0	84	0.1	0.3	5.6	0	82	0.1	0.3	8.2
150	3	66	0.9	0.6	20.9	3	59	0.7	0	23.4	2	67	0.3	0.3	17.5
250	7	63	1	0.9	48.6	6	56	0.9	1	53.9	5	49	0.8	0.3	45
407	10	61	1	2.2	63	10	53	1	2.9	59.3	8	46	1	1.9	53.7

#views	SQ$_4$					SQ$_5$				
	#HM	%NS	CVP	CVT (ms)	ANDT (ms)	#HM	%NS	CVP	CVT (ms)	ANDT (ms)
50	1	80	0.1	0.3	7.5	1	78	0	0.6	11.3
150	3	66	0.2	0.3	16.5	3	70	0	0	27.4
250	5	55	0.7	1.6	30.3	6	53	0.3	0.6	37.7
407	9	46	1	1	52.6	10	39	1	1.6	60.6

#views: no. of views in the view pool
#HM: avg. number of computed homomorphisms from the views to the query
%NS: avg. percentage of inverted lists nodes scannded
CVP: avg. probability for the query to be completely covered by the views in the view pool
CVT: avg. time for computing covering view nodes
ANDT: avg. time for disk loading, decompressing and bitwise ANDing bitmap materializations

(b) Query evaluation statistics

Figure 18: Query evaluation time and statistics by incrementally loading view materializations for the synthetic dataset

to the query, the number of input nodes accessed, whether the query is completely covered by the views, and the query optimization time consisting of the time needed for finding the covering view nodes of the query nodes and the time needed for disk loading, decompressing and bitwise ANDing the bitmaps of the node materializations. The reported numbers are the averaged measurements over the ten repeats of the incremental view selection. Figures 16, 17 and 18 report on the query evaluation time and the evaluation statistics for the five queries on the DBLP, XMark and the synthetic datasets. Recall that the evaluation time consists of the optimization time and the query execution time.

First, the query execution time improves as the number of views in the view pool increases. This is expected as an increase in the number of the materialized views reduces the number of accessed nodes (Figures 16(b), 17(b) and 18(b)), which in turn reduces the CPU and I/O times.

Second, the benefit on the query execution time obtained by the addition of new views decreases when the number of views in the view pool goes up. On the DBLP dataset, with 1000 materialized views (whose size is 0.54% of the input data) we already obtained about 93% of the average query execution time improvement obtained using all the views. On the XMark dataset, with 1500 materialized views (7.95% the size of the input data) we obtained 93% of the average execution time improvement, and on the synthetic dataset, with 150 materialized views (5.1% the size of the input data), we obtained 82% of the average execution time improvement. This is so because, as we can see in the columns %NS of Figures 16(b), 17(b) and 18(b), the views initially used for computing a query reduce dramatically the number of input inverted list nodes accessed.

8.6. Optimization Time vs. Benefit on Query Execution Time

If we keep increasing the number of views we consider for coverage during the optimization of a query, we will reach a point beyond which additional views will not contribute anymore significantly to the reduction of the execution cost if at all. The reason is that the views found that far to be usable for answering the query produce (almost) the minimum size inverted sublists that can be possibly obtained from the views in the view pool for the computation of the query. On the other hand, the consideration of these additional views will increase the optimization time and the benefit obtained on the query execution cost by the use of the additional views will be offset by the additional optimization time spent due to the consideration of these additional views. Therefore, this point yields the minimum query evaluation time.

We want to examine how much time we should spend on optimizing a query (equivalently, how many views we need to examine from the pool of materialized views) in order to minimize the query evaluation cost. The plots of figures 16(a), 17(a), and 18(a) show the evaluation time for all queries and datasets when sets of materialized views considered for optimization increase gradually (the measurements are averaged over 10 different orderings of the views in the view pool). As we can see, with an imperceptible exception in the curves of queries XQ_3 and XQ_4 after 1000 views, all curves decrease monotonically. In fact, the optimization time is very small compared to the benefit in the query execution time obtained from using the materialized views. For instance, across all three datasets, the query XQ_4 on XMark has the least benefit of 405ms (Figure 17(a)) if all the views are used. The corresponding optimization time is 18.1ms (Figure 17(b)) representing 4.5% of the benefit. The query SQ_1 on the synthetic dataset has the largest optimization time of 64ms (Figure 18(b)). The corresponding query evaluation benefit is 4599ms (Figure 18(a)). That is, the optimization time represents 1.4% of the benefit. For all the other queries the ratio is smaller than 8%. Therefore, we conclude that even for view pools that are large enough to guarantee prevalence over the other approaches, in practice we do not need to restrict the time spent on optimization (equivalently, the number of views checked for usability).

This answers question (d) raised in the introduction on the time worth spending in finding what views can be used for optimally answering the queries.

8.7. Summary

In summary, our experiments on evaluating different approaches for optimizing XML queries illustrate the following three points.

- When the given dataset contains irregular and/or recursive structures, the *MVbit* approach is the most robust and stable solution. It obtains significant performance savings for both simple and complex queries. In addition, our experiments reveal the following two points about *MVbit*: (1) The performance savings can be obtained with a restricted number of views, that is, at the expense of a small space overhead compared to the size of the input data. (2) In order to minimize the query evaluation time, there is no need to restrict the optimization time as it generally represents a very small fraction of the benefit in the query execution time.
- When the structures of the given dataset are regular and non-recursive, the structural index ap-

proaches *SIemb* and *SIlu* can generally achieve the best performance benefit to space cost ratio. This happens because in this case: (a) the input query has usually a very small number of embeddings to the structural index, and therefore a large number of nodes that do not participate in the query solutions can be skipped, and (b) since the structural index (without the extents) is usually much smaller than the corresponding XML data, the space usage is insignificant. However, when the dataset contains highly recursive structures, the traditional structural index approach *SIemb* usually exhibits an exponential behavior. The improved structural index approach *SIlu* resolves the combinatorial explosion problems but does not succeed in outperforming *MVbit*.

- Our experiments also showed that the *XB* approach usually incurs large space cost. When the query solutions are clustered in certain areas of the input data, the *XB* approach is capable of skipping large portions of the input data, and therefore, can achieve very good query performance. However, when the query solutions are dispersed throughout the entire input data its node pruning power is largely reduced, and it fails to compete with the *SIlu* and the *MVbit* approaches.

8.8. Comparison with a Relational Approach

We also compare our bitmapped materialized views approach with an approach which is implemented on top of a state-of-the-art commercial RDBMS which can use materializes views to optimize queries. We compare the two approaches in terms of disk space usage and query performance. As in our approach, we store in the relational database the positional representation of the XML tree nodes. We adopt the *Node* approach developed in [68] for storing XML data in an RDBMS: the nodes of an XML tree are stored in a table `Node` with schema `(label, start, end, level)`, where `label` records the label and the triplet `(start, end, level)` records the positional representation (see Section 3) of a node. In this context, TPQs can be expressed by specifying child and descendant relationships in the Where clause of the SQL queries. An example is shown in Figure 19. Further, view usability conditions can be expressed in terms of implications of conjunctions of arithmetic comparisons (mainly inequalities). The evaluation of the SQL version of a TPQ involves two steps: node

```
Select *
From   Node article, Node month, Node title
Where  article.label = 'article'    and
       month.label = 'month'        and
       title.label = 'title'        and
       article.start < month.start  and
       month.start < article.end    and
       article.start < title.start  and
       title.start < article.end    and
       article.level + 1 = title.level;
```

Figure 19: SQL statement for the TPQ `//article[.//month]/title`

selection and node joining [21]. The node joining step joins the data nodes retrieved by the node selection step on their `(start, end, level)` values. Because the relational system was unable to compute queries in a reasonable amount of time without indexes, we built indexes on all the attributes of table `Nodes` despite the fact that our approach does not use indexes.

For the experiments, we used the three datasets, the view sets and the query sets used in the previous experiments (Figures 8, 10 and 11 respectively). For each dataset, the five random queries and the views which can be used to answer these queries (called *relevant* views in the sequel) were translated into SQL statements. In addition, the SQL views were materialized and query rewriting was enabled.

The table of Figure 20 shows the space usage of *MVBit* and the relational approach. Columns "Inv. List size" and "MVBit size" show the size of the inverted lists and the size of the bitmap views, respectively, in *MVBit*. Columns "Table size", "Indexes size" and "SQL MV size" show the size of table `Node`, the size of indexes and the size of the materialized views, respectively, in the Relational system. As mentioned above, only relevant views were materialized for the relational approach. The materialization size of all the views was estimated based on the materialization size of the relevant views. We were unable to materialize any view on the synthetic dataset since the view materialization and query evaluation times were prohibitive despite the use of indexes (no results were returned after many hours for a single query). As we can see the relational approach uses 17.5 times more space on the DBLP dataset and 75 times more space on the XMark dataset compared to *MVBit*. The materialized views in the relational approach consume space several times larger than that of the data: more than 3 and more than 44 times larger for the DBLP and XMark datasets, respec-

		MVBit		*Relational*		
Dataset	#Views (non-empty)	Inv.lists size (MB)	*MVBit* size (MB)	Table size (MB)	Indexes size (MB)	SQL MV size (MB)
DBLP	2024 (773)	242	2.7	456	1136	1947.6
XMark	3369	128.3	22.8	256	624	11628.5
SD	407	260	36	408	1136	*n/a*

Figure 20: Comparison of space consumption of *MVBit* and the relational approach on the three datasets.

Query		#Solutions	*MVBit* (sec.)	SQL w.o. MVs (sec.)	SQL with MVs (sec.)	Comments
DBLP	DQ_1	2474	0.029	613.55	0.76	
	DQ_2	0	0.001	0.06	0	
	DQ_3	0	0.001	5.51	0.29	
	DQ_4	0	0.001	281.5	0.05	
	DQ_5	8	0.006	3.25	0.02	
XMark	XQ_1	2268326	1.95	3886.19	10510.13	SQL: the plan without views is cheaper
	XQ_2	50000	0.32	3686.07	1593.8	SQL: 5.41 sec. with another MV
	XQ_3	113722	0.52	5337.4	1109.26	
	XQ_4	48750	0.3	345.47	171.79	SQL: 4.96 sec. with another MV
	XQ_5	27095027760	2.36	22263.07	650.2	SQL: 2 non-overlapping MVs are used

Figure 21: Comparison of query evaluation times with *MVBit* and the relational approach on DBLP and XMark.

tively. In contrast, the ratio of the materializated views space over the data space in *MVbit* for the DBLP, XMark, and the synthetic datasets are 1.1%, 15.1%, and 12.2%, respectively.

The table of Figure 21 shows the query performance of *MVBit* and the relational approach. As with the previous experiments, the performance is measured by the query evaluation time, which is the time required to compute the query answer (excluding the time for returning solutions). Since the relational approach was unable to evaluate any given query on the synthetic data within a reasonable amount of time, we report only on results on the DBLP and XMark datasets. As we can see, *MVBit* largely outperforms the relational approach in all cases (empty queries might take zero time with both approaches). In particular, for queries on the XMark dataset having a large number of solutions, *MVBit* outperforms the relational approach by two to four orders of magnitude.

The following observations can be made. The relational approach can make good use of the materialized views on the DBLP dataset which is shallow and the queries are empty or do not have a lot of solutions. On the XMark dataset which is deeper and the queries have more solutions, the relational approach was unable, in many cases, to find the materialized view with the highest benefit for a given query, despite the fact that only the relevant views were materialized. For instance, as shown in Figure 21, the relational approach fails to identify the most beneficial view for answering queries XQ_2 and XQ_4. When forced to use some other materialized view its performance improves dramatically. For query XQ_1, it fails to recognize that a plan without views is cheaper and it selects a more expensive plan which uses a materialized view. Further, the relational approach was unable (or was not considering beneficial) to exploit simultaneously multiple overlapping materialized views. In our experiment, every query can be answered using at least two views. However, only on XQ_5 did the relational system use two materialized views, and even in this case the views were non-overlapping. In contrast to the relational system, which rewrites the queries using the views, *MVBit* does not need a query rewriting process and can exploit simultaneously all the materialized views that can be used for answering a query without spending significant time on optimizing the queries.

In summary, *MVBit* outperforms the relational approach in terms of space and time without using indexes. This result is not surprising. As pointed out in [21] and demonstrated in [68], relational systems typically do not support efficiently joins involving multiple inequality-comparison predicates. Efficient XML query processing by RDBMS systems can benefit by the inclusion in their strategies of holistic twig-join algorithms [7].

9. Conclusion

We have addressed the problem of optimizing XML queries using materialized views in the framework of the inverted lists evaluation model which has been established as the most prominent one for evaluating queries on large persistent XML data. Under this context, we have suggested a novel approach for materializing views which stores the inverted lists of only those XML tree nodes that occur in the answer to the view. A further originality of our approach is that view materializations are stored as compressed bitmaps, which not only minimizes the storage space but also reduces CPU and I/O costs by translating view materialization processing into bitwise operations. Unlike the traditional approach which evaluates queries by identifying a compensating expression that rewrites the query using the materialized views, our approach computes the query answer by executing holistic stack-based algorithms on the view materializations.

We have conducted extensive experiments to compare our approach with recent outstanding structural summary and B-tree index based approaches. In order to make the comparison more competitive we also proposed an extension of a structural index approach to resolve combinatorial explosion problems. Our experimental results show that our compressed bitmapped materialized views approach is the most efficient, robust, and stable one for optimizing XML queries. It obtains significant performance savings at a very small space overhead and has negligible optimization time even for a large number of materialized views in the view pool.

Our results are obtained without using any systematic technique for selecting views for materialization. An interesting research direction consists in developing elaborated techniques for selecting views when frequent input queries are known in advance but also in the absence of such information. These techniques are expected to further improve the performance of the compressed bitmapped materialized views approach making it even more competitive against its rivals.

Materialized views need to be maintained when the data over which they are defined change. In this paper, we consider a static environment. Another interesting research direction involves studying techniques for maintaining materialized views in response the changes to the underlying data.

Appendix A. Proofs

We provide in this Appendix proofs for the propositions and theorems presented earlier in the paper.

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.

Proof. If there is a homomorphism from V to Q that maps Y to X, the materialization of X is a subset of that of Y on any XML tree T since for every embedding of Q to T there is an embedding of V to T that map X and Y to the same node in T. Therefore, X is covered by Y.

If X is covered by Y and there is no homomorphism from V to Q that maps Y to X, let T_Q be the XML tree constructed by replacing double (i.e., descendant) edges in Q (if any) by single ones and by adding a new root r (if r is not already the root). Since there is no homomorphism from V to Q that maps Y to X, V does not have an embedding to T_Q that maps Y to X. Thus, there is a tree node in the materialization of node X of Q on T_Q that does not appear in the materialization of node Y of V on T_Q. Therefore, X cannot be computed using the materialization of Y on T_Q which contradicts that X is covered by Y. We conclude that if X is covered by Y, there is a homomorphism from V to Q. $\square$

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.

Proof. Clearly, if Q can be answered using V_i, the materialization of a node X in Q is a subset of the materialization of some node Y_i in V_i on any set of inverted lists L. Therefore, there are nodes $Y_1, \ldots, Y_k$ in $V_1, \ldots, V_n$, such that, X can be computed using $L_{Y_1} \cup \ldots \cup L_{Y_k}$ for every L, that is, Q can be answered using $V_1, \ldots, V_n$.

If Q can be answered using $V_1, \ldots, V_n$ but for no V_i, $i \in [1, n]$, Q can be answered using V_i, let T_Q be the XML tree constructed by replacing double (i.e., descendant) edges in Q (if any) by single ones and by adding a new root r (if r is not already the root). Since Q cannot be answered using any V_i, $i \in [1, n]$, no V_i has a homomorphism to Q and thus, no V_i has an embedding to T_Q. Therefore, the materialization of the V_is on T_Q is empty. In contrast, Q has an embedding to T_Q and therefore its materialization is non-empty. But then, for no node X in Q, there are nodes $Y_1, \ldots, Y_k$ in $V_1, \ldots, V_n$, such that, X can be computed on T_Q using the materializations of $Y_1, \ldots, Y_k$ on T_Q, which contradicts our assumption that Q can be answered using $V_1, \ldots, V_n$. We conclude that Q can be answered using $V_1, \ldots, V_n$ only if for some $V_i,\ i \in [1, n]$, Q can be answered using V_i.□

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

Proof. The *if* part is straightforward.

If Q can be answered using exclusively $V_1, \ldots, V_n$ but for some node X in Q, there is no covering node in a $V_i,\ i \in [1, n]$, let T_Q be the XML tree constructed as in the proof of Theorem 4.2. Query Q has a unique homomorphism h to T_Q, and let x be the image of X under h. Since no V_i has a homomorphism to Q that maps a node to X (because of Theorem 4.3), no V_i has an embedding to T_Q that maps a node to x, and thus, x does not appear in the materialization of any V_i on T_Q. Therefore, Q cannot be answered on T_Q using exclusively the materializations of $V_1, \ldots, V_n$ on T_Q, and as a consequence, Q cannot be answered using exclusively $V_1, \ldots, V_n$ which contradicts our assumption. We conclude that Q can be answered using exclusively $V_1, \ldots, V_n$ only if for every node in Q, there is a covering node in some $V_i,\ i \in [1, n]$. □

Proposition 6.1 Let Q be a query which is computed on an XML tree T using the materialized views $V_1, \ldots, V_n$. Let also for every child edge X/Y in Q, Y be a leaf node in Q. Algorithm $TwigStack$ will produce no redundant path solutions if for every child edge X/Y in Q, node Y has a covering node in some view V_i that also has a child incoming edge.

Proof. A redundant path solution can be produced by $TwigStack$ for a root-to-leaf path p of a query Q when there is an embedding M' for query Q' (the query obtained by replacing child edges in Q by descendant ones) such that: (a) the image of p' (the counterpart root-to-leaf path of p in Q) under M' satisfies the child relationships of p, and (b) some child relationship in Q but not in p is violated by the image of Q' under M'.

Let Q be a query that satisfies the condition of Proposition 6.1, p be a root-to-leaf path in Q, and X/Y be a child edge in Q but not in p. Assume that for every child edge Z/W in Q, node W has a covering node in some view V_i and this covering node has a child incoming edge. Let $L_{m(Z)}$ denote the intersection of the materializations of the covering nodes of query node Z in $V_1, \ldots, V_n$. Let's also assume that $TwigStack$ is used to evaluate Q using $V_1, \ldots, V_n$ as described in this section (that is, if a node Z in Q has a covering node in $V_1, \ldots, V_n$, $TwigStack$ uses $L_{m(Z)}$ for this node; otherwise, it uses the inverted list of the label of Z). We show below that if a path solution for p is produced by $TwigStack$, it is not redundant.

Assume that a path solution s is computed by $TwigStack$ for p based on an embedding M' of Q'. Let x be the image of X under M'. Since X/Y is a child edge in Q, X and Y have covering nodes X_i and Y_i, respectively, in some view V_i such that X_i/Y_i is in V_i. Therefore, $TwigStack$ uses $L_{m(X)}$ for X and $L_{m(Y)}$ for Y, and $x \in L_{m(X)}$. Since X_i/Y_i is in V_i, x has a child node y in T which belongs to the materialization of Y_i. Clearly, y belongs to the materialization of every covering node of Y and consequently $y \in L_{m(Y)}$. Therefore, we can construct an embedding for Q based on M' which coincides with M' on the nodes of p. We conclude that the path solution s for p is not redundant. □

References

[1] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, K. Shim, Optimizing queries with materialized views, in: ICDE, 1995, pp. 190–200.

[2] S. Chaudhuri, K. Shim, Optimizing Queries with Aggregate Views, in: EDBT, 1996, pp. 167–182.

[3] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, M. Ziauddin, Materialized views in oracle, in: VLDB, 1998, pp. 659–664.

[4] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata, Answering complex SQL queries using automatic summary tables, in: SIGMOD, 2000, pp. 105–116.

[5] S. Agrawal, S. Chaudhuri, V. R. Narasayya, Automated Selection of Materialized Views and Indexes in SQL Databases, in: VLDB, 2000, pp. 496–505.

[6] J. Goldstein, P.-Å. Larson, Optimizing queries using materialized views: A practical, scalable solution, in: SIGMOD, 2001, pp. 331–342.

[7] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: SIGMOD Conference, 2002, pp. 310–321.

[8] H. Jiang, W. Wang, H. Lu, J. X. Yu, Holistic twig joins on indexed XML documents, in: VLDB, 2003, pp. 273–284.

[9] A. Balmin, F. Özcan, K. S. Beyer, R. J. Cochrane, H. Pirahesh, A framework for using materialized XPath views in XML query processing, in: VLDB, 2004, pp. 60–71.

[10] W. Xu, Z. M. Özsoyoglu, Rewriting XPath queries using materialized views, in: VLDB, 2005, pp. 121–132.

[11] B. Mandhani, D. Suciu, Query caching and view selection for XML databases, in: VLDB, 2005, pp. 469–480.

[12] L. V. Lakshmanan, H. W. Wang, Z. J. Zhao, Answering Tree Pattern Queries Using Views, in: VLDB, 2006, pp. 571–582.

[13] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, Structured materialized views for XML queries, in: VLDB, 2007, pp. 87–98.

[14] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, K.-F. Wong, Multiple materialized view selection for XPath query rewriting, in: ICDE, 2008, pp. 873–882.

[15] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, P. A. Boncz, Materialized view selection in XML databases, in: DASFAA, 2009, pp. 616–630.

[16] X. Wu, D. Theodoratos, W. H. Wang, Answering XML queries using materialized views revisited, in: CIKM, 2009, pp. 475–484.

[17] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, J. Teubner, Pathfinder: XQuery - the relational way, in: VLDB, 2005, pp. 1322–1325.

[18] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, J. Teubner, MonetDB/XQuery: a fast XQuery processor powered by a relational engine, in: SIGMOD Conference, 2006, pp. 479–490.

[19] H. Georgiadis, V. Vassalos, XPath on steroids: exploiting relational engines for XPath performance, in: SIGMOD Conference, 2007, pp. 317–328.

[20] M. M. Moro, Z. Vagena, V. J. Tsotras, Tree-pattern queries on a lightweight XML processor, in: VLDB, 2005, pp. 205–216.

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

[22] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, C. Zaniolo, Efficient structural joins on indexed XML documents, in: VLDB, 2002, pp. 263–274.

[23] J. Lu, T. Chen, T. W. Ling, Efficient processing of XML twig patterns with parent child edges: a look-ahead approach, in: CIKM, 2004, pp. 533–542.

[24] B. Yang, M. Fontoura, E. J. Shekita, S. Rajagopalan, K. S. Beyer, Virtual cursors for XML joins, in: CIKM, 2004, pp. 523–532.

[25] L. Chen, A. Gupta, M. E. Kurul, Stack-based algorithms for pattern matching on dags, in: VLDB, 2005, pp. 493–504.

[26] X. Wu, S. Souldatos, D. Theodoratos, T. Dalamagas, T. K. Sellis, Efficient evaluation of generalized path pattern queries on XML data, in: WWW, 2008, pp. 835–844.

[27] X. Wu, D. Theodoratos, S. Souldatos, T. Dalamagas, T. K. Sellis, Efficient evaluation of generalized tree-pattern queries with same-path constraints, in: SSDBM, 2009, pp. 361–379.

[28] F. Peng, S. S. Chawathe, XPath queries on streaming data, in: SIGMOD, 2003, pp. 431–442.

[29] P. Rao, B. Moon, Prix: Indexing and querying XML using prüfer sequences, in: ICDE, 2004, pp. 288–300.

[30] A. Barta, M. P. Consens, A. O. Mendelzon, Benefits of path summaries in an XML query optimizer supporting multiple access methods, in: VLDB, 2005, pp. 133–144.

[31] J. McHugh, J. Widom, Query optimization for XML, in: VLDB, 1999, pp. 315–326.

[32] Y. Wu, J. M. Patel, H. V. Jagadish, Structural join order selection for XML query optimization, in: ICDE, 2003.

[33] H. Wang, S. Park, W. Fan, P. S. Yu, ViST: A dynamic index method for querying XML data by tree structures, in: SIGMOD, 2003, pp. 110–121.

[34] H. Jiang, H. Lu, W. Wang, B. C. Ooi, XR-Tree: Indexing XML data for efficient structural joins., in: ICDE, 2003.

[35] H. Li, M.-L. Lee, W. Hsu, C. Chen, An evaluation of XML indexes for structural join, SIGMOD Record 33 (3) (2004) 28–33.

[36] R. Kaushik, R. Krishnamurthy, J. F. Naughton, R. Ramakrishnan, On the integration of structure indexes and inverted lists, in: SIGMOD, 2004, pp. 779–790.

[37] A. Arion, A. Bonifati, I. Manolescu, A. Pugliese, Path summaries and path partitioning in modern XML databases, World Wide Web 11 (1) (2008) 117–151.

[38] M. M. Moro, Z. Vagena, V. J. Tsotras, Evaluating structural summaries as access methods for XML, in: WWW, 2006, pp. 1079–1080.

[39] T. Milo, D. Suciu, Index structures for path expressions, in: ICDT, 1999, pp. 277–295.

[40] R. Kaushik, P. Bohannon, J. F. Naughton, H. F. Korth, Covering indexes for branching path queries, in: SIGMOD Conference, 2002, pp. 133–144.

[41] D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Y. Vardi, Answering regular path queries using views, in: ICDE, 2000, pp. 389–398.

[42] C. Yu, L. Popa, Constraint-based XML query rewriting for data integration, in: SIGMOD, 2004, pp. 371–382.

[43] J. Tang, S. Zhou, A theoretic framework for answering XPath queries using views, in: XSym, 2005, pp. 18–33.

[44] N. Onose, A. Deutsch, Y. Papakonstantinou, E. Curtmola, Rewriting nested XML queries using nested views, in: SIGMOD, 2006, pp. 443–454.

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

[46] J. Wang, J. X. Yu, XPath rewriting using multiple views, in: DEXA, 2008, pp. 493–507.

[47] L. V. S. Lakshmanan, H. Wang, Z. Zhao, Answering Tree Pattern Queries Using Views, in: VLDB, 2006, pp. 571–582.

[48] J. Wang, J. Li, J. X. Wu, Answering tree pattern queries using views: a revisit, in: EDBT, 2011.

[49] B. Cautis, A. Deutsch, N. Onose, V. Vassalos, Efficient rewriting of XPath queries using query set specifications, PVLDB 2 (1) (2009) 301–312.

[50] J. Lu, T. W. Ling, C. Y. Chan, T. Chen, From region encoding to extended dewey: On efficient processing of XML twig pattern matching, in: VLDB, 2005, pp. 193–204.

[51] I. Elghandour, A. Aboulnaga, D. C. Zilio, C. Zuzarte, Recommending XMLTable views for XQuery workloads, in: XSym, 2009, pp. 129–144.

[52] P. Godfrey, J. Gryz, A. Hoppe, W. Ma, C. Zuzarte, Query rewrites with views for XML in db2, in: ICDE, 2009, pp. 1339–1350.

[53] D. Phillips, N. Zhang, I. F. Ilyas, M. T. Özsu, InterJoin: Exploiting indexes and materialized views in XPath evaluation, in: SSDBM, 2006, pp. 13–22.

[54] D. Chen, C.-Y. Chan, ViewJoin: Efficient view-based evaluation of tree pattern queries, in: ICDE, 2010, pp. 816–827.

[55] A. Marian, J. Siméon, Projecting XML documents, in: VLDB, 2003, pp. 213–224.

[56] V. Benzaken, G. Castagna, D. Colazzo, K. Nguyen, Type-based XML projection, in: VLDB, 2006, pp. 271–282.

[57] C. M. Hoffmann, M. J. O'Donnell, Pattern matching in trees, J. ACM 29 (1) (1982) 68–95.

[58] G. Miklau, D. Suciu, Containment and equivalence for an XPath fragment, in: PODS, 2002, pp. 65–76.

[59] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, V. Josifovski, Streaming XPath processing with forward and backward axes., in: ICDE, 2003, pp. 455–466.

[60] B. Choi, M. Mahoui, D. Wood, On the optimality of holistic algorithms for twig queries., in: DEXA, 2003.

[61] T. Chen, J. Lu, T. W. Ling, On boosting holism in XML twig pattern matching using structural indexing techniques, in: SIGMOD, 2005.

[62] R. Goldman, J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, in: VLDB, 1997.

[63] R. Kaushik, P. Shenoy, P. Bohannon, E. Gudes, Exploiting local similarity for indexing paths in graph-structured data, in: ICDE, 2002, pp. 129–140.

[64] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, D. Srivastava, Structural joins: A primitive for efficient XML query pattern matching, in: ICDE, 2002, pp. 141–152.

[65] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, K. S. Candan, Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents, in: VLDB, 2006.

[66] Y. Diao, et al., YFilter, http://yfilter.cs.umass.edu/.

[67] K. Wu, E. J. Otoo, A. Shoshani, Optimizing bitmap indices with efficient compression, ACM Trans. Database Syst. 31 (1) (2006) 1–38.

[68] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman, On supporting containment queries in relational database management systems, in: SIGMOD, 2001.