World Wide Web (2015) 18:607–632
DOI 10.1007/s11280-013-0272-y

Configuring bitmap materialized views for optimizing XML queries

Xiaoying Wu · Dimitri Theodoratos · Anastasios Kementsietsidis

Received: 27 October 2013 / Accepted: 26 November 2013
Published online: 12 January 2014

Abstract In recent years the inverted lists evaluation model along with holistic stack-based algorithms have been established as the most prominent techniques for evaluating XML queries on large persistent XML data. In this framework, we are using materialized views for optimizing XML queries. We consider a novel approach which instead of materializing the answer of a view materializes exactly the inverted sublists that are necessary for computing the answer of the view. This originality allows storing view materializations as compressed bitmaps, a solution that minimizes the materialization space and empowers performing optimization operations as CPU-efficient bitwise operations. To realize the potential of bitmap materialized views in optimizing query performance, we define and address the following problem (view configuration problem): given an XML tree and its schema find a template of tree-pattern views (view configuration) such that: (a) the views of this configuration can answer all the queries that can be issued against the schema, (b) their materialization fits in the space provided, and (c) evaluating the queries using these views minimizes the overall query evaluation cost. We consider an instance of this problem for tree pattern queries. Our intension is to find view configurations whose materializations are small enough to be stored in main memory. We find two candidate solution configurations and we identify cases where views can be excluded from materialization in a configuration without affecting query performance. In order to compare our approach with an approach which also can support the optimization of every query on the schema, we implemented an improvement

The research of this author was supported by the National Natural Science Foundation of China under Grant No. 61202035 and 61272110.

X. Wu
State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China
e-mail: xw43@njit.edu

D. Theodoratos (✉)
New Jersey Institute of Technology, Newark, NJ, USA
e-mail: dth@cs.njit.edu

A. Kementsietsidis
IBM Research, Yorktown Heights, NY, USA
e-mail: akement@us.ibm.com

of a state-of-the-art approach which is based on structural indexes. Our experimental results show that our approach is stable, greatly improves evaluating queries without materialized views, outperforms the structural index approach on all test cases and is very close to the optimal. These results characterize our approach as the best candidate for supporting the optimization of queries in the framework of the inverted lists model.

Keywords XPath query evaluation · XML · Materialized views · View configuration

1 Introduction

A powerful query optimization technique in current database systems consists in materializing (that is, precomputing and storing) views. The main idea is that storing these materializations in a view pool will be beneficial to the evaluation of some incoming queries. In the Relational model, the use of materialized views for answering and optimizing queries has been studied extensively [7, 8] and integrated into commercial DBMSs in past years in addition to indexing techniques [1, 5, 12, 39]. In 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 for evaluating a query using materialized views.

In this paper, we adopt a novel approach for materializing views in the context of XML. Our approach avoids many of the limitations of the traditional approach in view usability. In our new context, view materializations are compressed bitmaps of inverted lists. Our bitmap view approach is extremely space efficient. It is able to materialize thousands of views (and consequently to support the efficient evaluation of thousands of queries) in a space where other approaches that follow previous materialization schemes [3, 18, 29, 38] can only materialize a handful of views (and consequently support a restricted number of queries). Because of their small size thousands of bitmap materialized views (whose size is usually a tiny fragment of the base data) can be kept in main memory. This is one of the several reasons this technique presents a significant advantage over other view materialization techniques. Our ultimate goal is to evaluate the efficiency of our approach. That is, (a) we want to see to what extent we can speed up the evaluation of queries using bitmap materialized views compared to evaluating queries without using views, and (b) how our approach compares in terms of evaluation time to other approaches that aim at using auxiliary structures (materialized views and indexes) for optimizing the evaluation of queries. However, in order to do this evaluation the problem of deciding what views to select for materialization in the database has to be solved. The view selection problem is a well known problem in databases both in the Relational [11, 14, 30] and the XML context [18, 27]: given a query workload and information about the dataset decide what views to materialize in the database that satisfy a number of constraints (e.g., materialization space constraints) and minimize a cost (e.g., the evaluation cost of a given workload). This problem does *not* directly fit our needs because our intention in this paper is to find view sets to materialize that can support the optimization of *all* the queries of the class under consideration that can be issued against the database and not only (or mainly) those that appear in the workload. Therefore, we define here a different version of the view selection problem that can accommodate our goal. In order to introduce the problem we address in this paper and outline our contributions, we overview next with an example our novel framework for optimizing queries on XML data. We provide formal definitions later in Section 2.1.

1.1 Query optimization framework

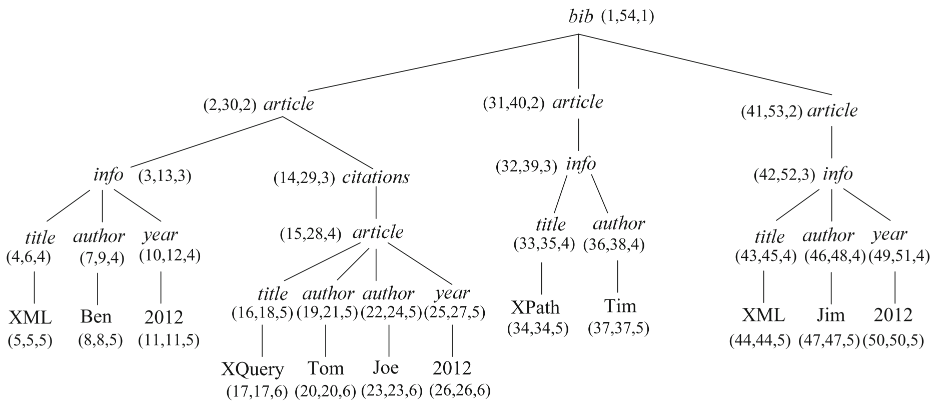
A recent approach for evaluating queries on large persistent XML data assumes that the data is preprocessed and the position of every node in the XML tree is encoded [6, 15]. Figure 1a shows an XML tree and the positional encoding of its nodes in the form of triplets ($\langle begin, end, level \rangle$). The nodes in the XML tree are partitioned by node label, and an index of inverted lists is built on this partition. Figure 1b shows the inverted lists for some of the labels in the XML tree of Figure 1a. As is common with the inverted lists of XML data we assume that there is an index structure that identifies the nodes in an inverted list satisfying a given predicate [6]. For instance, in the example of Figure 1b, the inverted list of label *year*(2012) provides the nodes of the inverted list of *year* which satisfy the predicate $year = 2012$.

Queries are XPath expressions. 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. Subscripts in labels distinguish nodes with the same label. A node label in bold indicates the output node of a query. Query Q asks for the authors of cited articles published in 2012. View V_1 computes the authors of articles published in 2012 or of articles citing an article published in 2012. View V_2 computes the citing articles. As an example, one can see that view V_2 has one match to the XML tree. According to the traditional approach, its answer consists of the subtree rooted at node (15, 28, 4) labeled *article*. In contrast to the traditional approach, the answer of a query Q in our approach is not a subtree of the XML tree but a set of tuples having one field for every node in Q (that is, all query nodes are regarded as output nodes). 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 3a shows the answer of view V_1 on the XML tree of Figure 1a as a set of tuples. Observe that if the answer of a view is directly materialized as a set of tuples, nodes may be redundantly stored multiple times.

Query evaluation model 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. A major advantage of this evaluation model is that in order to evaluate a query, only the inverted lists of the labels that appear in the query need to be fetched from disk and scanned.

View materialization For materializing views, we employ a novel approach where instead of materializing the answer of a view V (set of tuples), we materialize inverted sublists for all view nodes. These inverted sublists comprise exactly the XML tree nodes that appear in the answer of the view for the corresponding node. Figure 3b and d depict the materializations of views V_1 and V_2 of Figure 2 as sets of sublists (one sublist for each view node). This solution greatly reduces the view materialization space since it can store an exponential number of tuples in polynomial space, and also addresses the redundancy issue associated with tupled-based view materializations. In order to minimize the storage space, the inverted sublist of every materialized view node is represented as a bitmap over the corresponding inverted list and is compressed. Figure 3c and e depict the materializations of views V_1 and V_2 of Figure 2 as sets of bitmaps.

Computing a query using materialized views As we explain later, if a view can be used in answering a query, mappings are established from the view nodes to the query nodes. A view node which is mapped to a query node n is called *covering* node of n in the view. In



(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) }

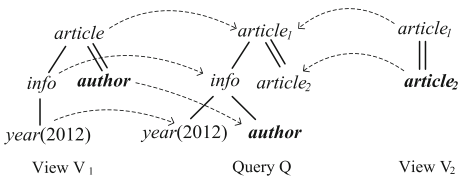
year(2012) = { (10,12,4), (25,27,5), (49,51,4)}

(b) inverted lists

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

order then to evaluate the query using the view, a holistic stack based algorithm is employed which computes the answer by using, for every query node, the inverted sublist of the covering view node (if any) instead of the inverted list of its label. This reduces substantially the evaluation time since the sublists are, in general, much smaller than the corresponding inverted lists. In the example of Figure 2 the dotted lines show the covering view nodes of query nodes. Therefore, a query evaluation algorithm can compute the answer of Q by using for query node $article_1$ the inverted sublist of the materialized view node $article$ shown in Figure 3b instead of the inverted list of label $article$ shown in Figure 1b. Note that our approach allows the computation of a query answer using inclusively or exclusively multiple views materialized in a view pool. When multiple views can be exploited in answering a query, a query node can be computed using the intersection of the inverted sublists of all its covering view nodes in the same and/or different views. This even further reduces the size of the sublists used for computing the query. In our running example of Figure 2 the answer of Q can be computed by using for query node $article_1$ the intersection of the inverted sublists of the materialized view nodes $article$ of V_1 and $article_1$ of V_2 shown in Figure 3b and d, respectively.

Figure 2 A query, two views, and homomorphisms from the views to the query



article	info	year(2012)	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(2012) = {(10,12,4), (49,51,4)}

(b) Materialization of V_1 as a set of sublists

article = 1001
info = 101
author = 11101
year(2012) = 101

(c) Materialization of V_1 as a set of bitmaps

$article_1$ = {(2,30,2)}
$article_2$ = {(15,28,4)}

(d) Materialization of V_2 as a set of sublists

$article_1$ = 1000
$article_2$ = 0100

(e) Materialization of V_2 as a set of bitmaps

Figure 3 Materializations of views V_1 and V_2 of Figure 2 on the XML tree of Figure 1a

Besides storage gains, the bitmap representation of the inverted sublist has also performance advantages during the computation of query answers since: (a) the intersection of the inverted sublists can be implemented as a bitwise operation which incurs less CPU cost, and (b) fetching into memory the operand bitmaps and the resulting inverted sublist has less I/O cost than fetching the operand inverted sublists.

View usability As shown in Section 3.1, in the context of our novel concept of view materialization, a query can be answered using a view if there is a homomorphism from the view to the query. This condition can be generalized to multiple views. In our example of Figure 2, query Q can be answered using view V_1 or view V_2, as the depicted homomorphisms suggest. Note that these homomorphic mappings define also the covering view nodes of a query node. In fact, because all nodes of Q are covered by view nodes, Q can be answered using exclusively the (materializations) of V_1 and V_2. Interestingly, what is needed for exploiting the view materializations in the computation of a query is the identification of the covering view nodes of the query and this, as we explain in Section 3.1, can be done in polynomial time without enumerating all the possible homomorphisms from the views of the view pool to the query.

1.2 Problems addressed and contribution

To realize the potential of bitmap materialized views in optimizing query performance, it is necessary to have a process to determine, for a given input XML database, a proper set of views to materialize. Our goal is to support the optimization of *all queries that may be issued against the XML database rather than a specific set of queries*. To this end, we make the following contributions.

- We formally define and address a problem called view configuration problem. A view configuration represents a set of views defined by a view template. The view configuration problem takes as input an XML tree and its schema and aims at finding a view configuration, which when materialized as bitmap views (a) fits in the space available for view materialization, (b) can answer all the satisfiable queries that can be issued against the schema, and (c) minimizes the overall cost of evaluating the queries using the materialized views. We consider an instance of the problem for tree pattern queries and views. Our intention is to find view configuration whose bitmap materializations are small enough to fit in main memory (Section 2).

- We are looking for a pragmatic not a theoretical solution to the view configuration problem. We find two possible solution configurations (binary paths and binary twigs—the former subsuming the later) of increasing benefit and size. We show how we can refine them and avoid materializing redundant views without compromising their capacity in supporting the optimization of queries (Section 3.2).
- We implemented our approach on both view configurations. In order to compare it with previous ones, we considered and implemented also a state-of-the art approach (denoted SIdx) which combines inverted lists with structural indexes. SIdx has been shown in [21] to be generally faster than other approaches in the context of the inverted lists model. Since the original version of SIdx has limitations (in particular with recursive DTDs) we employed for our comparison an extension of it that addresses these problems (Section 3.3).
- We run extensive experiments on real, benchmark and synthetic datasets. The experimental results show that our approach for computing queries using bitmap views with the solution configurations greatly improves the basic inverted lists approach, in certain cases by orders of magnitude. It also outperforms SIdx on all testing cases and comes very close to the optimal. Further, it is stable and scales smoothly in terms of both space and query executions time when the size of data increases (Section 4).
- Our results suggest that using our approach for appropriately selecting views for materialization as compressed bitmaps is the most promising and practical technique for optimizing XML queries.

Related work is presented in Section 5 along with a further comparison of our approach to the traditional approach. We conclude in Section 6 and suggest future work.

2 Bitmap view configurations

In this section, we present the data and evaluation models we adopt, and the class of queries and views we consider, and we outline the bitmap materialized views approach. We then introduce the concept of view configuration and define formally the view configuration problem.

2.1 The bitmap materialized view approach

Data model An XML database is commonly modeled by a node labeled tree structure. We denote by $\mathcal{L}$ the set of XML tree node labels. Without loss of generality, we assume that only the root node of every XML tree is labeled by $r \in \mathcal{L}$.

As described in the introduction, the XML tree data is preprocessed and the position of every node is encoded. For every label a in the XML tree, an inverted list L_a of the nodes with label a is produced. List L_a contains the positional representation ($\langle begin, end, level\rangle$ triplets) of the nodes labeled by a in T ordered by their *begin* field. Given an XML tree T, L denotes its set of inverted lists.

Query and view language We focus on tree-pattern queries and views. A *tree-pattern query* (TPQ) specifies a pattern in the form of a tree. Every node in a TPQ Q has a label from $\mathcal{L}$. There are two types of edges in Q. A single-line (resp. double-line) edge between two nodes in Q denotes a child (resp. descendant) structural relationship between the two nodes. Figure 4a and b show two queries.

The answer of a TPQ on an XML tree is a set of tuples. Each tuple has one element for every TPQ node. The elements are (positional representations of) XML tree nodes and are named by the corresponding TPQ nodes. More formally, an *embedding* of a TPQ Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) node labels are preserved, and (b) if there is a single-line (resp. double-line) edge between two nodes X and Y in Q, $M(Y)$ is a child (resp. descendant) of $M(X)$ in T. We call *solution* of Q on T a tuple whose elements are the images of the nodes in Q under an embedding of Q to T. The *answer* of Q on T is the set of solutions of Q under all possible embeddings of Q to T.

We assume that a query has at least two nodes since otherwise it can be answered by a simple lookup on the corresponding inverted list. Also, we assume that a query does not include a root-to-leaf path $r//a$ (rooted at an r node and comprising only a single double-line edge). Figure 4c–e show examples of such tree patterns (which are not considered to be legal queries). Note that the tree-pattern of Figure 4b *does not* include such a path and is a legal query. A root-to-leaf path $r//a$ is not allowed in a query Q because it does not provide any additional restriction to the rest of the query: the answer of Q is the Cartesian product of the answer of the rest of Q (that is, the query resulting by removing the a node and its incoming double-line edge) and the inverted list for label a. For instance the answer of query 4d is the Cartesian product of the answer of the query r/a and the inverted list of label b.

A view is a named query. The class of views is not restricted: any kind of legal query can be a view.

Computing queries in the inverted lists evaluation model The inverted lists evaluation model ignores the XML tree T and assumes that the input for the evaluation of queries and views is the *set of inverted lists* L of T. The query evaluation algorithms in this model are based on stacks. Comparison studies on XML query evaluation techniques [13, 21] show that holistic stack-based algorithms [6, 15, 34, 35] in the inverted lists model are superior to other algorithms and evaluation models (streaming/navigational approaches [24] or sequential/string matching approaches [26]).

Let X be a node in query Q labeled by label a. The elements of query node X in the answer of Q are computed by the evaluation algorithm by scanning the inverted list L_a. Therefore, only inverted lists for labels that appear in Q are involved in the computation.

When a query Q is evaluated on L, if the elements of node X in the answer of Q can be computed using a sublist L' of L_a we say that node X can be *computed on L using the sublist L'*.

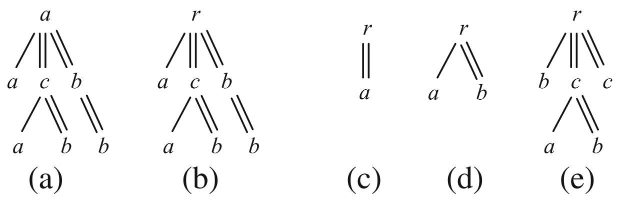


Figure 4 **a–b** queries, **c–e** tree patterns not considered as queries

Materialized views as compressed bitmaps If X is a node in a view V labeled by a, the *materialization,* L_X*, of* X *on* L is a sublist of L_a containing only those nodes that are elements of X in the answer of V on L. The materialization of X is represented by a bitmap on L_a which is stored compressed to even further reduce the materialization space. The *materialization* $V(L)$ *of a view on* L is the set of the materializations of its nodes on L.

Using materialized views in query answering: definitions Let $V(L)$ be the materialization of a view V on L. View V *can be used in answering* a query Q if for some 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 $L_Y \in V(L)$. If this is the case, we say that query node X *is covered* by view node Y or that Y is a *covering node* of X.

When multiple materialized views $V_1, \ldots, V_n$ are present, they can be exploited for answering a query. We say that the views $V_1, \ldots, V_n$ *can be used for answering a query* Q iff some of the V_is can be used for answering Q.

Let's assume that Q can be answered using $V_1, \ldots, V_n$, and let $CN(X)$ denote the set of covering nodes of X in $V_1, \ldots, V_n$. If every node in Q is covered by a node in some V_i, we say that Q can be answered using *exclusively* $V_1, \ldots, V_n$. In this case, $\forall X \in Q$, $CN(X) \neq \emptyset$. Otherwise, we say that Q can be answered using *inclusively* $V_1, \ldots, V_n$.

Computing a query answer using view materializations In order to use the materializations of views $V_1, \ldots, V_n$ in the computation of Q, for every node X of Q, the intersection of the materializations of all the covering nodes in $CN(X)$ is computed. Since these materializations are stored as bitmaps, their intersection can be performed by bitwise AND-ing bitmaps and the cost of this operation is insignificant. The resulting sublists for all the query nodes are fed into a stack-based algorithm which computes the answer of Q. The intersection of the materializations of the nodes in $CN(X)$ is a sublist which is usually much smaller than the inverted list for the label of X. Therefore, when $CN(X)$ for the query nodes X is available, computing the answer of Q using the materialized views non-strictly reduces the cost of computing Q using the inverted lists of the query node labels. We show in the next section that the cost of the computation of the nodes in $CN(X)$ is not significant, even for a large number of views $V_1, \ldots, V_n$.

2.2 The view configuration problem

We assume that an XML tree and the structural part of a DTD (tree or graph) without constraints and cardinalities is given. In the absence of a DTD, a structural summary [20] (e.g. an 1-index), is equally appropriate. We refer to this structure (DTD structure or structural summary) as *schema*.

View configurations A *configuration* of views is a view template. It represents the set of views resulting by instantiating the template in all possible ways. The template is similar to an XPath tree-pattern expression which involves axis variables '$\#_i$' (instead of child '/' and descendant '//' axes) and label variables $X_i, Y_j, Z_k, \ldots$ (instead of XML tree labels). The axis variables '$\#_i$' are instantiated with '/' or '//', and the label variables are instantiated with XML tree labels to yield TPQs. An example of a configuration is the template $X\#Y$ which represents path pattern views with two nodes involving a child or descendant relationship (we call this configuration *binary path* configuration). Figure 5a–d show some possible instantiations of this template. Another example of a configuration is the template $X[.\#_1 Y]\#_2 Z$ representing tree pattern views with a root and two child nodes involving

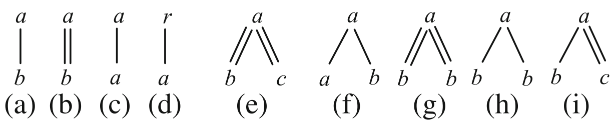


Figure 5 Binary path views (**a–d**), binary twig views (**e–i**)

child and/or descendant relationships (we call this configuration *binary twig* configuration). Figure 5e–i shows some possible instantiations of this template.

A query is *satisfiable* w.r.t. a schema s, if it has a non-empty answer on an XML tree which complies with s. The *instantiation $C(s)$ of a view configuration C on a schema s* is the set of views which can be obtained by instantiating the label variables of template C with node labels from s and the axis variables of C with child and descendant axes such that the resulting view is satisfiable w.r.t. s. The *materialization* $C(L)$ of a configuration C on the set of inverted lists L of an XML tree that complies with s refers to the materialization of the views in $C(s)$ on L.

The problem We now define the *view configuration problem*. Given a schema s, the set L of inverted lists of an XML tree that complies with s, and a threshold t representing the size of the space available for view materialization, compute a view configuration C that satisfies the following conditions:

(a) the space consumed by the materialization of C on L does not exceed t. That is, $size(C(L)) \leq t$.
(b) every query which is satisfiable w.r.t. s can be answered using exclusively the views in the instantiation $C(s)$ of C on s.
(c) the average cost of answering all the satisfiable w.r.t. s queries using the views in $C(s)$ is minimal.

Note that we are interested only in configurations whose materialization on L does not exceed 1 to 2 % of the size of the base data so that they can be also stored and remain in main memory.

Note also that condition (b) guarantees that not only every (satisfiable w.r.t. s) query is supported by a materialized view but also that every part of the query is supported by a materialized view of the configuration instantiation. Indeed, the requirement of a query being answerable using exclusively the views in the instantiation, guarantees that every node of it is covered by a node of a materialized view of the configuration instantiation.

Multiple alternative formulations of the view configuration problem can be envisaged including considering smaller (more restricted) view configurations, materializing multiple view configurations, and materializing a fragment of a view configuration. These considerations are out of the scope of this paper, our purpose being: (a) to provide a practical and quick way for choosing bitmap view sets for materialization that will support efficiently all the queries, and (b) to compare the bitmap materialized view approach on these view selections against other state of the art approaches that also support the evaluation of all the queries.

3 Choosing view configurations

We provide in this section a solution to the view configuration problem and then we show how solution configurations can be refined. In order to do so, we use some previous results on computing queries using bitmap views which are presented next. In the last part of this section, we elaborate on two approaches our approach is compared against.

3.1 Computing covering view nodes

As explained in Section 2.1, in order to compute query answers using bitmap views what is needed is the computation of the set of covering view nodes for the query nodes. To identify covering view nodes we employ the concept of homomorphism from a view to a query. 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. Figure 6 shows a query and three views as well as homomorphisms from the views to the query. A view can have multiple homomorphisms to a query. For instance, View V_1 has two homorphisms to Q. The following theorem relates node coverage to homomorphisms.

Theorem 1 [36, 37] *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.*

A query node can be covered by multiple nodes of the same or different views. For instance, in the example of Figure 6, query node b_1 is covered by nodes b of views V_1 and V_2.

Based on Theorem 1, the set of covering view nodes of a given query node is determined by the homomorphisms from the views to the query. 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 query node X. Then, the set $CN(X)$ of covering nodes for X in V is

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

The set of covering nodes of a given query node in multiple views is determined in terms of the set of covering nodes of the query in a single view: let X be a node in query Q, and $CN_1(X), \ldots, CN_n(X)$ be the sets of covering nodes of X in $V_1, \ldots, V_n$, respectively. Then, the set $CN(X)$ of covering nodes of X in $V_1, \ldots, V_n$ is

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

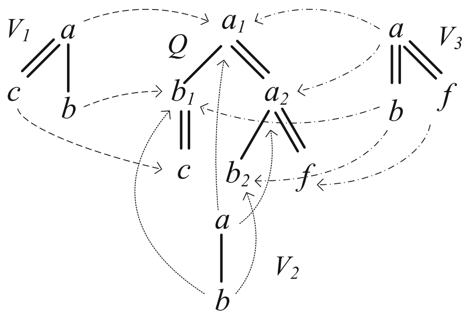


Figure 6 Homomorphisms from the views V_1, V_2 and V_3 to query Q

View usability conditions The following corollary of Theorem 1 specifies necessary and sufficient conditions for using materialized views in answering a query.

Corollary 1 *A materialized view V can be used in answering Q iff there is a homomorphism from V to Q.*

In the example of Figure 6, each one of the views V_1, V_2 and V_3 can be used to answer query Q since there is at least one homomorphism from each one of them to Q. Further, since every node of Q is covered by a node in V_1, V_2 or V_3, Q can be answered using exclusively V_1, V_2 and V_3.

Avoiding homomorphism enumeration The covering nodes for a node of a query Q in a view V are defined in terms of the homomorphisms of V to Q. The number of these homomorphisms can be exponential in the number of view nodes. However, the number of covering nodes in $CN(X)$ is bounded by the number of nodes in V. For the computation of covering nodes we use a stack-based algorithm [36, 37] 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.

3.2 A solution to the view configuration problem

It is not difficult to see that the only configurations that satisfy condition (b) of the definition of the view configuration problem in Section 2.2 are those corresponding to the view templates $C_k = X[.\#_1 Y_1] \dots [.\#_{k-1} Y_{k-1}]\#_k Y_k$. Figure 7a shows an instantiation of this template for $k = 5$. If $k = 1$ the template corresponds to a binary path configuration ($X\#Y_1$). For $k = 2$ it corresponds to a binary twig configuration ($X[.\#_1 Y_1]\#_2 Y_2$). Intuitively, the views in the instantiation of any one of these configurations can be used to assemble any arbitrarily complex query or view.

Condition (a) of the view configuration problem constraints the size of the materialization of the selected configuration, while condition (c) requires the minimization of the query evaluation cost.

We can observe that if $n > m$, a configuration C_n subsumes a configuration C_m in the following sense: for every view V_m in the instantiation $C_m(s)$ of C_m on a schema s, there is a view V_n in $C_n(s)$ such that there is a homomorphism from V_n to V_m that maps descendant (resp. child) edges to descendant (resp. child) edges and all the nodes in V_m are images of nodes in V_n under the homomorphism. Clearly, V_m has also a homomorphism to V_n that maps descendant (resp. child) edges to descendant (resp. child) edges. Nodes in the two views V_n and V_m that are associated through these homomorphisms have the same materialization on the set of inverted lists L of any XML tree that complies with s. For instance, in the example of Figure 7b, nodes b of V_2 and b_1 and b_2 of V_4 have the same materialization on L. The reason is that for every embedding of V_m to an XML tree, there is an embedding of V_n that maps nodes associated through a homomorphism in the two views to the same XML tree node, and vice versa.

Also, if there is a homomorphism from V_m to a query Q, there is also a homomorphism from V_n to Q that maps view nodes associated through a homomorphism in V_m and V_n to the same query node.

These observations suggest that:

(a) The number of views in $C_n(s)$ is non-strictly larger than that in $C_m(s)$.

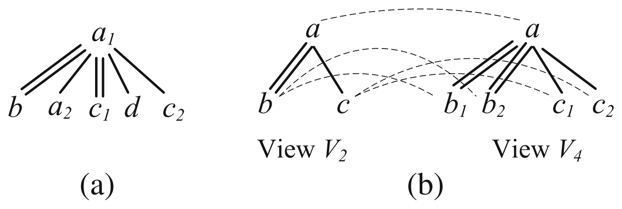


Figure 7 **a** An instantiation of configuration C_5, **b** Views V_2 and V_4 and their node correspondences

(b) The size of the materialization $C_n(L)$ of C_n on L is larger than that of $C_m(L)$. That is, the size of $C_k(L)$ increases monotonically with k.

(c) For every query node X, the intersection of the materializations (inverted sublists) of the covering view nodes of X in the instantiation $C_n(s)$ of C_n on s is a subset of the intersection of the covering view nodes of X in $C_m(s)$. As a consequence, the number of nodes in each inverted sublist employed for evaluating a query using materialized views from $C_n(L)$ is non-strictly smaller than what we can obtain using materialized views from $C_m(L)$.

The cost of evaluating a query using bitmap materialized views is composed of two parts: the query optimization cost and the query execution cost (the cost of running a holistic stack-based algorithm on the inverted sublists of the query nodes).

The optimization cost consists in turn of two components: the cost for computing the covering view nodes for the nodes of the query and the cost for decompressing and bitwise ANDing the bitmaps of the view node materializations. As mentioned in Section 3.1, our algorithm computes covering view nodes efficiently. The time for decompressing and bitwise ANDing the bitmaps of the view node materializations is insignificant.

The query execution cost which is the bulk of the query evaluation cost depends on the size of the inverted sublists used to compute the query. As mentioned in remark (c) above, the size of these inverted sublists, non-strictly monotonically decreases with k.

The query optimization cost is a very small percentage of the query execution cost (at least for values of k which lead to materializations of C_k of manageable size). This is due to the efficiency of our algorithm that computes the covering view nodes and is also confirmed experimentally in the next section. Since the optimization time is insignificant compared to the query execution time, choosing a configuration C_k with a larger k non-strictly reduces the query evaluation cost.

The previous remarks show that for values of k of practical interest, the solution to the view configuration problem is the configuration C_k with the greatest k whose materialization $C_k(L)$ fits in the available space (i.e. its size does not exceed the given threshold t). In fact, because of the low threshold we have imposed on the space available for materialization, we have to restrict the configurations we consider to binary twigs and often even to binary paths. The materializations of higher degree configurations ($k > 2$) exceed the threshold. Therefore, in practice, the solution to the view configuration problem is the binary twig configuration if its size on the input dataset allows its materialization or otherwise, the binary path configuration (whose materialization usually fits in the available space).

The next theorem shows that for non-recursive schemas (that is, schemas that do not have cycles) the inverted sublists reach their minimum size at $k = 2$ and therefore their size stabilizes beyond $k = 2$. As a consequence, for non-recursive schemas, the configurations

C_k with $k > 2$ need not be considered even if they fit in the space available for materialization: these configurations do not reduce the query evaluation cost and they increase the space consumption.

Theorem 2 *Let X be a node of a query Q on an XML tree T with non-recursive schema s. Let also $CN_2(X)$ (resp. $CN_k(X)$, $k > 2$) be the set of covering view nodes of X in the instantiation $C_2(s)$ of the configuration C_2 (resp. C_k) on s. Let finally L_2 (resp. L_k) be the intersection of the materializations (inverted sublists) of the nodes in $CN_2(X)$ (resp. $CN_k(X)$) on T. Then, $L_2 = L_k$.*

To prove the above theorem recall that from the previous discussion in this section, $L_k \subseteq L_2$. In order to show that $L_2 \subseteq L_k$, let X' be a covering node of X in a view $V_k \in C_k(s)$. Observe that if a node l in the inverted list of the label of X is not in the materialization of X' of V_k on T (because there is no embedding of V_k to T that maps X' to l) then there is a view $V_2 \in C_2(s)$ which is a subtree of V_k and comprises X', and l is not in the materialization of X' of V_2 on T. As an example of this observation, consider the XML tree T and the query Q and views V_3, V_2 and V_1 shown in Figure 8. Integers are used as node identifiers in T and the materialization of the query and view nodes on T are shown by the nodes in the figure (the materializations are empty for all the nodes of Q, V_3 and V_2 as this query and views do not have an embedding to T). Node 3 of T labeled by b is not in the materialization of node b of V_3. As a consequence, there is a view in $C_2(s)$ which is a subtree of V_3 and comprises node b and node 3 is not in the materialization of b of this view on T. This is view V_2. We conclude that $L_2 \subseteq L_k$.

Notice that this is not the case with a covering node of X in a view $V_2 \in C_2(s)$ (a binary twig): there is not necessarily a view $V_1 \in C_1(s)$ (a binary path) which is a subtree of V_2 that comprises X', such that l is not in the materialization of X' of V_1 on T. Looking again at the example of Figure 8, one can see that there is no view in $C_1(s)$ which is a subtree of V_2 that comprises node b such that node 3 is not in the materialization of b of V_1 on T. Indeed, view V_1 is the only binary path subtree of V_2 that comprises b and the materialization of b of V_1 contains 3.

In the rest of the paper, we focus on binary path and binary twig configurations. We discuss next how we can refine these two configurations by eliminating redundant views and by replacing views with simpler ones.

Refining the binary path configuration We start by providing a definition for redundant views.

Definition 1 Given a configuration C and a schema s, a view V in the instantiation $C(s)$ of C on s is *redundant* in the presence of another view V' in $C(s)$ if for every query Q and for

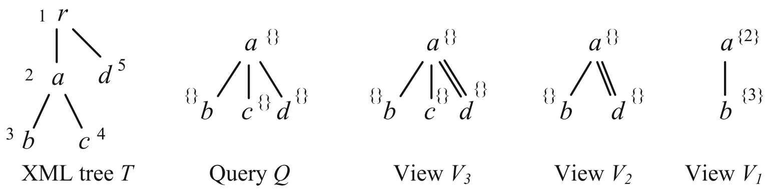


Figure 8 An XML tree T and a query and three views along with the materialization of their nodes on T shown by the nodes between { and }

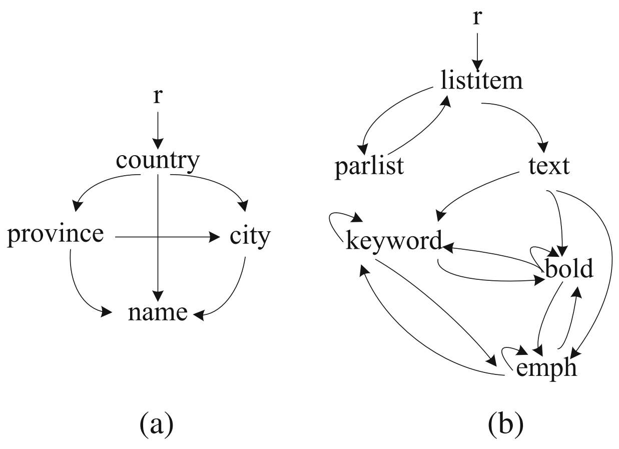


Figure 9 **a** A fragment of the MONDIAL schema, **b** A fragment of the XMark schema

any materialization of C on an XML tree that complies with s: if a node X_Q in Q is covered by a node X of V, there is a node X' of V' which also covers X_Q and the materialization of X' is a subset of the materialization of X.

Clearly, a redundant materialized view V does not need to be included in the materialization of a view configuration when the view that renders V redundant is materialized.

The next proposition characterizes redundant views in the instantiation of a binary path configuration on a schema. We first provide a definition: an edge (x, y) (x not necessarily different than y) in a schema s is *transitive* if: (a) node x or y are on a cycle in s that does not include (x, y), or (b) there is a path from node x to node y in s that does not include edge (x, y). As an example, in the schema of Figure 9a (a fragment of the MONDIAL[1] schema), edge (country, name) is transitive while (province, city) is not. In the schema of Figure 9b (a fragment of the XMark[2] schema), the edge (listitem, parlist) is the only non-transtive edge, All the other edges, e.g., (keyword, keyword) and (keyword, emph) are transitive.

Proposition 1 *Let C_1 be the binary path configuration and s be a schema. The view x/y is redundant in $C_1(s)$ in the presence of the view $x//y$ if the edge (x, y) is not transitive in s.*

The proof can be obtained by observing that in the materialization of C_1 on an XML tree that complies with s, the materialization the view x/y is the same as the materialization of the view $x//y$.

Refining the binary twig configuration In a binary twig configuration, *symmetric binary twig* queries are twigs whose leaf nodes have the same label and their edges are both child or both descendant edges. Figure 5g and h show examples. As discussed in a more general context in the beginning of Section 3.2, a symmetric twig query can be replaced in the configuration materialization by the binary path view obtained by merging the leaf nodes

[1]http://www.dbis.informatik.uni-goettingen.de/Mondial/#XML

[2]http://monetdb.cwi.nl/xml/

of the twig. For instance, the binary twig of Figure 5g can be replaced by the binary path of Figure 5b and the one of 5h by 5a.

Clearly, the size of the materialized binary path is smaller than that of the corresponding binary twig. Therefore, replacing symmetric binary twigs by the corresponding binary paths in a configuration materialization: (a) does not affect the size of the inverted sublists used for evaluating the query using the materialized views (that is, it does not affect the query execution cost), (b) decreases the optimization time of the queries since the number of view nodes to be considered when computing the covering view nodes of query nodes is lower, and (c) reduces the materialization space (therefore increasing the chances for a binary twig configuration to fit in the space available for materialization).

The next proposition characterizes redundant views in the instantiation of a binary twig configuration on a schema.

Proposition 2 *Let C_2 be the binary twig configuration and s be a schema. The binary twig view $x[./y]//y$ is redundant in $C_2(s)$ in the presence of the binary twig view $x[.//y]//y$ if the edge (x, y) is not transitive in s.*

The proof in this case results from the fact that in the materialization of C_2 on an XML tree that complies with s, the materialization of the view $x[./y]//y$ is the same as the materialization of the view $x[.//y]//y$ (all x nodes and all y nodes in the two views have the same materialization, respectively).

The view configuration selection process can be summarized as shown in Figure 10.

3.3 Comparison approaches

Our approach cannot be compared to previous view selection approaches since they all aim at supporting a predefined workload of queries. In contrast, our approach aims at optimizing all the queries. It is appropriate to compare it with index-based approaches in the inverted lists model since it shares with them the goal of improving query performance by filtering out from the inverted lists nodes that do not contribute to the answer of the query (*irrelevant nodes*). For this reason, we choose an approach which is based on structural indexes [16]. Our choice is based on the fact that this approach was shown in [21] to perform better than other approaches in the inverted lists model. For the comparison, we extend this approach to further improve its performance.

```
Input: a schema $s$, the set $L$ of inverted lists of an XML tree that complies with $s$,
    a materialization space threshold $t$.
Output: the index $i$ of a view configuration $C_i$ to be materialized in the available
    space (i = 0 denotes no solution configuration).
k := 1; size = 0;
while $size < t$
    Compute the instantiation $C_k(s)$ of $C_k$ on $L$;
    Remove from $C_k(s)$ redundant and symmetric (if applicable) views;
    Compute the materialization $C_k(L)$ of the refined $C_k(s)$ on $L$;
    size := $size(C_k(L))$;
    k := k + 1
endwhile;
return $k - 1$.
```

Figure 10 An outline of the view configuration selection process

A structural index based approach 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 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 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 [4, 16, 21, 22] considers an approach that combines structural indexes with inverted lists to support XML query evaluation.

An often-used structural index, also adopted here, is called *1-index* [20]. A 1-index considers as equivalent nodes in an XML tree T that have the same incoming path from the root of T. It is a tree representing a summarization of the paths that actually occur in T. It can be constructed by traversing the given XML tree once, and it is usually much smaller than the corresponding XML data.

The structural index approach usually processes a given query Q in two steps. In the first step, in most existing realizations [4, 21], it computes the embeddings of Q to the 1-index G. Because the size of G is usually small, the cost of this step is not important.

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 $TwigStack$ [6] is employed for performing this evaluation. The solutions obtained from each evaluation are unioned to compute the answer of Q.

When Q has a very small number of embeddings to G, and is very selective, the structural index approach can greatly reduce the CPU and disk-read cost of the original inverted lists approach. The structural index approach refines the label-based partitioning of the nodes of the XML tree T. Because the partitioning is usually much more refined, the size of the extents is much smaller than that of inverted lists. Therefore, the structural index makes it possible to skip a large number of nodes that do not participate in the query solutions.

However, the original structural index 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 use an improved version of the approach which modifies the original one in two ways.

First, for computing the images of the nodes of Q to G, we employ an algorithm which, similarly to the algorithm that computes covering nodes, performs the computation without enumerating all the embeddings of Q to G.

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 $TwigStack$.

An optimal approach For a given a query, minimum size inverted sublists for evaluation can be obtained when the query itself is stored as a materialized view. In this case, the sublists of the view nodes comprise exactly the XML tree nodes that appear in the answer of the query for the corresponding query nodes. Therefore, the number of their nodes represents a lower bound in the number of nodes of the sublists produced by any approach that aims at improving the inverted lists approach by filtering out irrelevant nodes. Reducing the number of nodes of the sublists given as input to an evaluation algorithm in the inverted lists model reduces also the query evaluation cost. Clearly, not all possible queries can be stored as materialized views in the database. Thus, this is not a view materialization approach that can be implemented. However, the number of sublist nodes used by this approach for the evaluation of a query can be compared with that of other approaches as a measure of closeness to the optimal.

4 Experimental evaluation

We compare our bitmap materialized view approach on the two materialized view configurations with the improved structural index approach. Even though the two approaches employ different techniques, they have a common denominator which is that both 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 [6] (denoted INV), (2) the improved structural index approach (denoted $SIdx$), and (3) our compressed bitmap materialized view approach (denoted *MVbit*) on the configuration with binary paths (denoted $MVPath$) and on the configuration with binary twigs (denoted $MVTwig$). We refer to the evaluation of queries using *MVbit* on their own materialization as *OPT*. The performance of the approaches is measured in terms of three metrics: (a) the total time required for evaluating the query, (b) the total number of inverted list nodes accessed (a metric that reflects the ability of each approach to skip nodes that do not belong to the final query answer), and (c) the number of page I/Os used.

4.1 Experimental setup

Our implementation was coded in Java. All the experiments reported here were performed on an Intel Core 2 CPU 2.13 GHz processor with 2GB memory running JVM 1.6.0 in Windows XP Professional. Each displayed time value in the plots is averaged over five runs.

Datasets To analyze the behavior of each approach we ran experiments on three datasets which have different structural properties. The statistics of the datasets are shown in Table 1. The first one is a real dataset, the DBLP dataset.[3] The DBLP dataset is flat, shallow and bushy. It contains a few fairly regular structural patterns. The second one is a benchmark dataset using $XMark$[4] 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.[5] The generator was configured with $NumberLevels = 8$ and $MaxRepeats = 7$. 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 Table 1.

4.2 View and query generation

View generation For the two materialization configurations $MVPath$ and $MVTwig$, we use the DTD of the XML document to generate all the possible satisfiable binary path and binary twig views.

[3] http://dblp.uni-trier.de/xml/

[4] http://monetdb.cwi.nl/xml/

[5] http://www.alphaworks.ibm.com/tech/xmlgenerator

Table 1 Dataset statistics

	DBLP	*XMark*	*Synthetic*
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

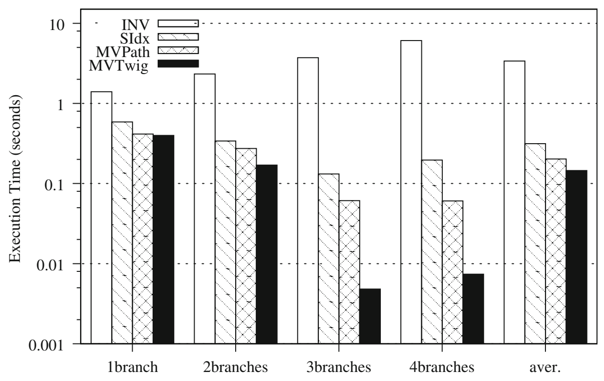
Most of the edges in the schemas of the DBLP and XMark datasets are non-transitive (see Section 3.2). Therefore, we can take advantage of Propositions 1 and 2 to reduce the number of views that need to be materialized by excluding some of them. Table 2 shows the number of the views generated for each one of the three datasets. The XMark dataset has the largest DTD in terms of elements, and generated the largest number of views. Notice also that a large percentage of the generated views for the DBLP dataset have empty answers. The reason is that most of the cardinality constraints on the DBLP DTD elements are optional and also, unlike the XMark and the synthetic datasets, the DBLP dataset is not randomly generated according to a DTD. Therefore, even if a view is satisfiable w.r.t. the DTD, it may not necessarily have an answer in an instance conforming to that DTD.

Query generation We used the XPath generator $YFilter$ http://yfilter.cs.umass.edu/ to generate random queries. $YFilter$ 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 (/).

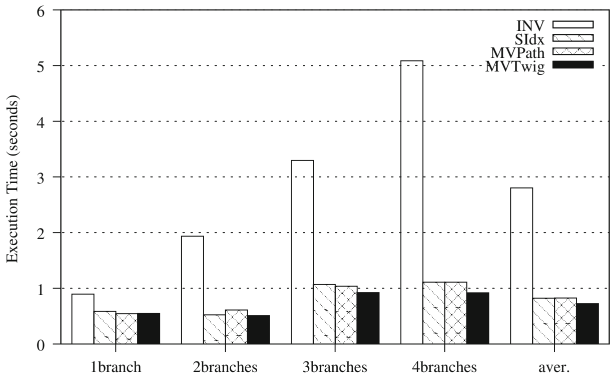
For each of the three datasets, we generated a broad spectrum of queries. It comprises one-, two-, three-, and four-branch queries, which are queries whose nodes have at most one, two, three, and four child nodes, respectively. The number of unique XPath queries to be generated for each query type was set to 100. The probability of descendant edges was set to 0.8 for all the generated queries. The maximum query depth was set to 4, 8, and 5 for the queries on the DBLP, XMark and synthetic datasets, respectively.

Table 2 Space usage (in MB) and number of materialized views for each one of the approaches

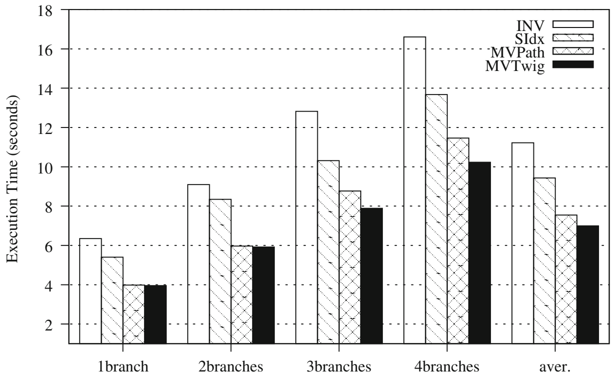
Datasets		*INV*	*SIdx*	*MVPath*	*MVTwig*
DBLP	# views (# non-empty)	242	242.2	241 (127)	3099 (1010)
	data size (size of bitmaps)			242.27 (0.27)	245.17 (3.17)
Xmark	# views (# non-empty)	128.3	128.7	372 (369)	3405 (3369)
	data size (size of bitmaps)			130.03 (1.73)	146.6 (18.3)
Synthetic	# views (# non-empty)	260	262.7	91 (91)	1614 (1433)
	data size (size of bitmaps)			265 (5)	344.3 (84.3)



(a) DBLP dataset



(b) XMark dataset



(c) Synthetic dataset

Figure 11 Query evaluation time

4.3 Space usage

We compare the disk space usage of the three approaches. Table 2 reports on the space consumed by each one of them on the three datasets considered. The baseline approach INV consumes the least space since no additional structures are employed. $SIdx$ uses slightly more space than INV because of the small space overhead incurred by the use of a more refined node partitioning.

The space used by the bitmap materialized view approach $MVbit$ consists of two parts: (1) the space for storing the inverted lists of the corresponding dataset, which is the same as that of INV, and (2) the space for storing view materializations. Recall that the view materializations are stored as compressed bitmaps, one per each view node. We compressed the bitmap materializations using the Java zip package, and we stored them in a commercial RDBMS as Binary Large Objects.

The space consumed by the view materializations for the two materialization configurations $MVPath$ and $MVTwig$ is shown in Table 2 within parentheses next to the total space used. The amount of space used by $MVPath$ is close to that of $SIdx$. $MVTwig$ uses significantly more space than $MVPath$. The ratios of the size of the materialized views used by $MVTwig$ ($MVPath$) over the size of the base data on the DBLP, XMark, and the synthetic datasets are 1.31 % (0.11 %), 14.3 % (1.35 %), and 32.4 % (1.92 %), respectively. Therefore, only on the DBLP dataset $MVTwig$ does not exceed the threshold of 2 % and can be exploited. On the XMark and the synthetic datasets, only the binary path configuration can be materialized.

4.4 Query performance

We compare the query performance of the three approaches. The performance is expressed by the query evaluation time, which is the total time required by each algorithm to compute the query answer. The query performance of an algorithm is determined by the total number of nodes accessed and the total number of I/Os performed during evaluation. Figure 11 reports on the query performance of evaluating four types of queries on the three datasets. Notice the logarithmic scale of the Y-axis in Figure 11a. Each reported value is the average evaluation time of 100 queries. The average number of nodes accessed as a percentage of those accessed by INV and the average number of I/Os performed by each algorithm are shown in Tables 3(a) and 3(b), respectively.

Table 3 Percentage of nodes accessed and number of I/Os per approach

	INV	*SIdx*	*MVPath*	*MVTwig*
(a) Average percentage of inverted list nodes accessed per query				
DBLP	100	9.50	4.03	2.79
XMark	100	20.75	17.61	14.48
synthetic	100	40.88	47.90	42.13
(b) Average number of I/Os per query				
DBLP	12993	1234	597	423
XMark	9464	1969	2286	2285
Synthetic	26783	11126	25892	25684

Table 4 shows the closeness of the query performance of each approach to the performance of OPT (the optimal one) on each dataset.

The bitmap materialized views approach As it can be observed in Figure 11, the bitmap view materialization approach $MVbit$ on the two materialization configurations greatly improves the baseline approach INV. It also outperforms $SIdx$ on all testing cases. Further, the performance of $MVbit$ is stable, and does not degrade with more complex queries and on data with highly recursive structures.

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 decompressing and bitwise ANDing the bitmaps of the node materializations. In [33], 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 of both configurations is already very small: the average optimization time of $MVPath$ and $MVTwig$ over all three datasets is only about 0.5 % and 1.63 % of the query evaluation time, respectively.

On the DBLP dataset, both $MVTwig$ and $MVPath$ outperform INV by orders of magnitude for the two-path queries and above. The reason is that both configurations are able to determine queries having empty answers almost immediately after they are issued because of empty covering view node materialization intersections (Section 2.1). $MVTwig$ outperforms of course $MVPath$ since its view set contains that of $MVPath$ and the availability of larger query subpatterns (binary twigs vs. binary paths) allows a more aggressive filtering of inverted lists nodes that do not participate in the answer of the query. This in turn reduces the CPU and I/O costs (Figure 11a, b and c).

$MVbit$ obtains significant performance savings at a small space overhead thanks to the compressed bitmap view materializations. This is especially the case with the $MVPath$ configuration. As Table 2 shows, the space used by $MVPath$ exceeds that of INV only by about 0.11 %, 1.35 %, 1.92 %, on the DBLP, XMark, and synthetic datasets, respectively. With such small space overhead, $MVPath$ achieves query performance which is 71 %, 83 %, and 87 % of the optimal for the above three datasets, respectively (Table 4).

The structural index approach The structural index approach $SIdx$ uses a 1-index (a structural index) for the evaluation of the queries. As mentioned in Section 3.3, a 1-index makes it possible for the structural index approach to skip a large number of nodes that do not participate in the query solutions. It also helps $SIdx$ to cluster XML tree nodes participating in query answers in a smaller number of pages than $MVbit$. For this reason, $SIdx$ is able to achieve less page I/Os on the XMark and the synthetic datasets compared to the other approaches (Figure 2b).

Table 4 Closeness to the optimal performance

	INV	*SIdx*	*MVPath*	*MVTwig*
DBLP	4.22	45.52	70.57	98.49
XMark	24.25	82.77	82.52	93.82
Synthetic	55.23	69.29	86.66	94.68

A 1-index is also able to detect queries with empty answers thereby stopping their evaluation at an early stage of the computation. These are queries without embeddings to the 1-index of the dataset. However, not every query with an empty answer can be detected by the structural index approach. In contrast, $MVPath$ and $MVTwig$ are able to detect more cases of queries with empty answers during the optimization phase without executing the query.

In all the testing cases, $SIdx$ outperforms INV by filtering out a large percentage of irrelevant nodes. On the XMark dataset, $SIdx$ performs slightly better than $MVPath$ for 2-paths queries (Figure 11b).

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 3.3) can offset the savings obtained by the filtering of irrelevant data nodes and the reduction of the page I/Os. For instance, on the synthetic dataset, $SIdx$ is able to filter out the largest number of irrelevant nodes in the inverted lists for evaluating the queries (Table 3). With an average number of 88 images per query node, $SIdx$ is outperformed on the average by $MVPath$ and $MVTwig$ by a factor of 1.25 and 1.35, respectively (Figure 11c). For some queries, the query processing time of $SIdx$ even exceeds that of INV.

4.5 Scalability

Finally, we measured the scalability of our $MVbit$ approach on evaluating the four types of queries over XMark datasets with size increasing from 200MB to 800MB.

Figure 12a shows the space usage of $MVbit$ on the two materialization configurations. As a comparison, we also show the space usage of INV. As we can see, the size of materializations of both configurations grows very slowly as the document size increases, with the percentage on the corresponding space of INV being less than 1.5 % and 16 % for $MVPath$ and $MVTwig$, respectively.

Figure 12b reports on the query performance of the three approaches. We used the same set of queries which comprise all four query types. $MVTwig$ has the best time performance in all the cases. It significantly improves the query performance of INV by a factor of over five. Also, its performance grows very smoothly as the size of XMark datasets increases. The performance of $MVPath$ closely follows $MVTwig$.

In Figure 12c, we show the percentage of query optimization time over the total query evaluation time for $MVTwig$ and $MVPath$. As we can see, in all cases, the optimization time of both configurations is very small, with the percentage over the total query evaluation time being less than 5 % and 2 % for $MVTwig$ and $MVPath$, respectively. Also, the percentage decreases when the size of XMark datasets increases from 200MB to 800MB.

5 Related work

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: view-based techniques and index-based techniques. Results on the complexity of the query containment problem in the presence and in the absence of DTDs were presented in [19, 23].

View-based techniques Traditionally, the answer of an XML query (and likewise the materialization of a view) is a set of *subtrees* of the XML tree against which the query is evaluated. In order to evaluate a query using materialized views, a *compensating* query is computed

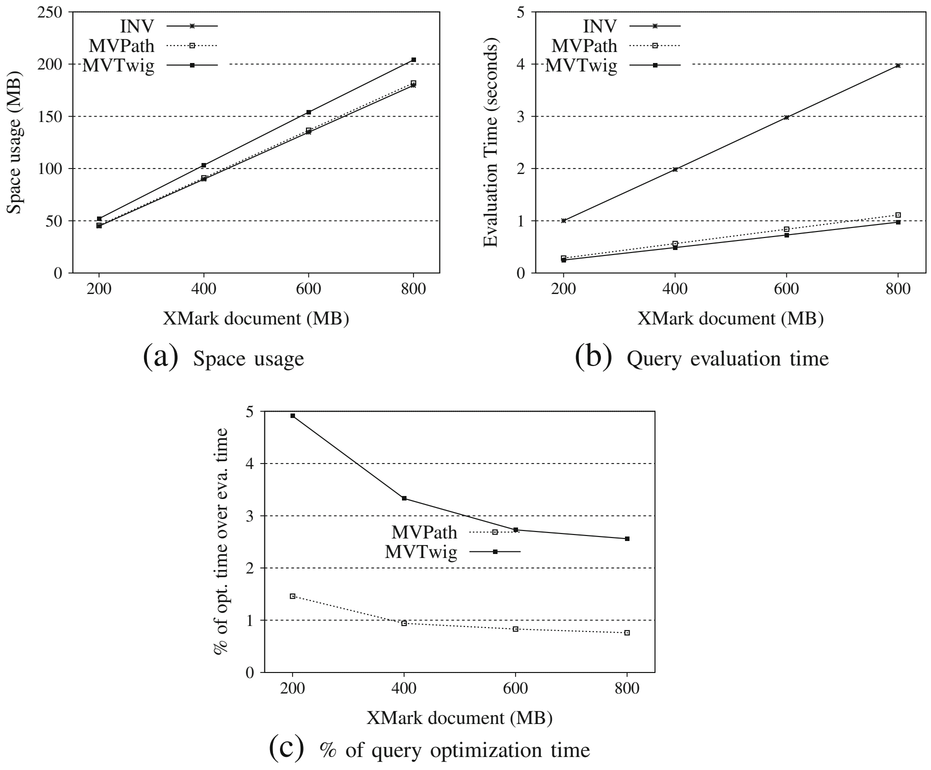


Figure 12 Scalability of $MVPath$ and $MVTwig$

which is a *rewriting* of the original query using materialized views [3, 17, 18, 27, 32, 38]. This compensating query is then evaluated over the materialized views and possibly the input XML document.

Many contributions [3, 17, 18, 38] are restricted to query rewritings using a *single* materialized view. A common constraining requirement for view usability in this context 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. Papers studying the problem of answering XML queries using multiple views [2, 27, 32] assume that output-preserving homomorphisms exist among views and they present rewriting algorithms which use intersection of view answers on node ids.

The traditional approach has a number of drawbacks: (a) view usability is low, (b) the benefit from using materialized views (when this is possible) is restricted, (c) the size of the view materializations can be very large, and (d) the query rewriting algorithms using views are complex. Materializing additional information besides the subtrees, e.g., ancestor path information, typed values and references to XML data [3], only partially addresses the issues mentioned above while increasing the materialization space.

In the example of Figure 2, according to the traditional approach, query Q cannot be answered using the materializations of both views V_1 and V_2 but also that it cannot be answered using any 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 showed in Section , 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.

A couple of papers study answering XML queries using materialized views in the inverted lists evaluation model [9, 25]. Phillips et al. [25] 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. Chen et al. [9] 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.

View selection In the context of XML databases, the materialized view selection problem is discussed in [18, 28]. The proposed approaches are all workload-driven. They syntactically analyze the workload to enumerate the relevant candidate views, and they greedily build a configuration of the most pertinent views. Both approaches adopt the traditional approach for answering XML queries using materialized views and suffer from its limitations and they consider only answering queries using a single view. The goal of this paper is to non-trivially support the optimization of *all* the queries that can be issued against the XML database.

The scheme of materializing views as bitmapped inverted sublists was first presented in [36]. Conditions for usability of multiple views are also reported there. The focus of that paper is on the problem of answering XML queries using exclusively materialized views in a distributed environment where the access to the base data is not possible. An elaboration on using (inclusively or exclusively) bitmap views for optimizing queries in a centralized environment is provided in [37]. Nevertheless, in both papers [36, 37] the selection of views is performed randomly. In contrast, in this paper we focus on providing a technique for selecting view configurations of compressed bitmapped materialized views in order to optimize queries on XML data that comply with a schema.

Our work compares better with indexing techniques for the inverted lists model because they also aim at filtering out irrelevant nodes in inverted lists.

Index-based techniques The approaches that speed up the processing of the original holistic evaluation algorithm TwigStack [6] 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 [10], the XB-tree [6], and the XR-tree [15]. A study in [21] compares the performance of the three B$^+$-tree based techniques on evaluating TPQs.

The other category consists of solutions which combine structural indexes with inverted lists to support XML query evaluation [4, 16, 21, 22]. 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 [4, 21] show that the structural index approach performs better than the index-based techniques PRIX [26] and ViST [31]. Also, in [21], it is shown that the structural index approach is generally faster than the XB-tree [6] while consuming much less space. For this reason, in this paper, we choose the structural index approach as a representative of the index-based techniques for comparing with our approach.

6 Conclusion

We studied a novel technique for optimizing XML queries in the framework of the inverted lists model which is currently the most promising technique for evaluating queries on large persistent data. Our technique is based on materializing views as compressed bitmaps of inverted sublists which can reside in main memory. In order to evaluate the potential of our approach and compare it with previous ones we defined and addressed the problem of choosing configurations of views for materialization which can support all the queries that can be issues against a schema. Our extensive experimental analysis showed that our approach outperforms a state-of-the-art approach which is based on structural indexes, is very close to the optimal and scales smoothly in terms of both space consumption and query performance. These results characterize our approach as the best candidate for supporting the optimization of queries in the framework of the inverted lists model.

Future work includes examining additional cases where the properties of the schema can be exploited to further refine the configurations. A different but relevant problem of equal interest in the context of the bitmap materialized view approach is the selection of views for materialization when a workload of queries of interest is given as input. The results presented in this paper can be leveraged for addressing that problem too.

References

1. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated Selection of Materialized Views and Indexes in SQL Databases. In: VLDB, pp. 496–505 (2000)
2. Arion, A., Benzaken, V., Manolescu, I., Papakonstantinou, Y.: Structured materialized views for XML queries. In: VLDB, pp. 87–98 (2007)
3. Balmin, A., Özcan, F., Beyer, K.S., Cochrane, R.J., Pirahesh, H.: A framework for using materialized XPath views in XML query processing. In: VLDB, pp. 60–71 (2004)
4. Barta, A., Consens, M.P., Mendelzon, A.O.: Benefits of path summaries in an XML query optimizer supporting multiple access methods. In: VLDB, pp. 133–144 (2005)
5. Bello, R.G., Dias, K., Downing, A., Feenan, Jr., J.J., Finnerty, J.L., Norcott, W.D., Sun, H., Witkowski, A., Ziauddin, M.: Materialized views in oracle. In: VLDB, pp. 659–664 (1998)
6. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD Conference, pp. 310–321 (2002)
7. Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim, K.: Optimizing queries with materialized views. In: CDE, pp. 190–200 (1995)
8. Chaudhuri, S., Shim, K.: Optimizing queries with aggregate views. In: EDBT, pp. 167–182 (1996)
9. Chen, D., Chan, C.-Y.: View join: efficient view-based evaluation of tree pattern queries. In: ICDE, pp. 816–827 (2010)

10. Chien, S.-Y., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C.: Efficient structural joins on indexed XML documents. In: VLDB, pp. 263–274 (2002)
11. Chirkova, R., Halevy, A.Y., Suciu, D.: A formal perspective on the view selection problem. In: VLDB, pp. 59–68 (2001)
12. Goldstein, J., Larson, P.-Å.: Optimizing queries using materialized views: a practical, scalable solution. In: SIGMOD, pp. 331–342 (2001)
13. Gou, G., Chirkova, R.: Efficiently querying large XML data repositories: a survey. IEEE Trans. Knowl. Data Eng. **19**(10), 1381–1403 (2007)
14. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. In: SIGMOD Conference, pp. 205–216 (1996)
15. Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic twig joins on indexed XML documents. In: VLDB, pp. 273–284 (2003)
16. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the integration of structure indexes and inverted lists. In: SIGMOD, pp. 779–790 (2004)
17. Lakshmanan, L.V.S., Wang, H., Zhao, Z.: Answering tree pattern queries using views. In: VLDB, pp. 571–582 (2006)
18. Mandhani, B., Suciu, D.: Query caching and view selection for XML databases. In: VLDB, pp. 469–480 (2005)
19. Miklau, G., Suciu, D.: Containment and equivalence for a fragment of xpath. J. ACM **51**(1), 2–45 (2004)
20. Milo, T., Suciu, D.: Index structures for path expressions. In: ICDT, pp. 277–295 (1999)
21. Moro, M.M., Vagena, Z., Tsotras, V.J.: Tree-pattern queries on a lightweight XML processor. In: VLDB, pp. 205–216 (2005)
22. Moro, M.M., Vagena, Z., Tsotras, V.J.: Evaluating structural summaries as access methods for XML. In: WWW, pp. 1079–1080 (2006)
23. Neven, F., Schwentick, T.: Xpath containment in the presence of disjunction, dtds, and variables. In: ICDT, pp. 312–326 (2003)
24. Peng, F., Chawathe, S.S.: XPath queries on streaming data. In: SIGMOD, pp. 431–442 (2003)
25. Phillips, D., Zhang, N., Ilyas, I.F., Özsu, M.T.: InterJoin: exploiting indexes and materialized views in XPath evaluation. In: SSDBM, pp. 13–22 (2006)
26. Rao, P., Prix, B.Moon.: Indexing and querying XML using prüfer sequences. In: ICDE, pp. 288–300 (2004)
27. Tang, N., Yu, J.X., Özsu, M.T., Choi, B., Wong, K.-F.: Multiple materialized view selection for XPath query rewriting. In: ICDE, pp. 873–882 (2008)
28. Tang, N., Yu, J.X., Tang, H., Özsu, M.T., Boncz, P.A.: Materialized view selection in XML databases. In: DASFAA, pp. 616–630 (2009)
29. Tang, J., Zhou, S.: A theoretic framework for answering XPath queries using views. In: XSym, pp. 18–33 (2005)
30. Theodoratos, D., Sellis, T.K.: Data warehouse configuration. In: VLDB, pp. 126–135 (1997)
31. Wang, H., Park, S., Fan, W., Yu, P.S.: ViST: a dynamic index method for querying XML data by tree structures. In: SIGMOD, pp. 110–121 (2003)
32. Wang, J., Yu, J.X.: XPath rewriting using multiple views. In: DEXA, pp. 493–507 (2008)
33. Wu, K., Otoo, E.J., Shoshani, A.: Optimizing bitmap indices with efficient compression. ACM Trans. Database Syst. **31**(1), 1–38 (2006)
34. Wu, X., Souldatos, S., Theodoratos, D., Dalamagas, T., Sellis, T.K.: Efficient evaluation of generalized path pattern queries on XML data. In: WWW, pp. 835–844 (2008)
35. Wu, X., Souldatos, S., Theodoratos, D., Dalamagas, T., Vassiliou, Y., Sellis, T.K.: Processing and evaluating partial tree pattern queries on XML data. IEEE Trans. Knowl. Data Eng. **24**(12), 2244–2259 (2012)
36. Wu, X., Theodoratos, D., Wang, W.H.: Answering XML queries using materialized views revisited. In: CIKM, pp. 475–484 (2009)
37. Wu, X., Theodoratos, D., Wang, W.H., Sellis, T.: Optimizing XML queries: bitmapped materialized views vs. indexes. Inf. Syst. **38**(6), 863–884 (2013)
38. Xu, W., Özsoyoglu, Z.M.: Rewriting XPath queries using materialized views. In: VLDB, pp. 121–132 (2005)
39. Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., Urata, M.: Answering complex SQL queries using automatic summary tables. In: SIGMOD, pp. 105–116 (2000)