ViewJoin: Efficient View-based Evaluation of Tree Pattern Queries

Ding Chen and Chee-Yong Chan

Dept. of Computer Science, National University of Singapore
{chending,chancy}@comp.nus.edu.sg

***Abstract*— There is a lot of recent interest in applying views to optimize the processing of tree pattern queries (TPQs). However, existing work in this area has focused predominantly on logical optimization issues, namely, view selection and query rewriting. With the exception of the recent work on InterJoin (which is primarily focused on path queries and views), there is very little work that has examined the important physical optimization issue of how to efficiently evaluate TPQs using materialized views. In this paper, we present a new storage scheme for materialized TPQ views and a novel evaluation algorithm for processing general TPQ queries using materialized TPQ views. Our experimental results demonstrate that our proposed method outperforms the state-of-the-art approaches.**

I. Introduction

A fundamental problem in XML query processing is tree pattern query (TPQ) matching [1] which computes all data instances in an XML database that match an input TPQ. A wide repertoire of techniques have been developed to address the performance of this important operation including data clustering methods (e.g. [10]), join algorithms (e.g. [1], [4], [6]), and indexing techniques (e.g., [8], [11], [15], [16]). More recently, there is growing attention on applying materialized views, which is an established and effective optimization technique in relational database systems [12], to TPQ matching (e.g., [3], [25], [27]).

The existing work on XML query processing with views has focused predominantly on two *logical optimization* issues, namely, view selection (e.g., [19], [25]), and query rewriting (e.g., [2], [3], [21], [25], [26], [27]). With the exception of the recent work on InterJoin [22], there is very little work that has examined the important *physical optimization* issue of how to efficiently evaluate TPQs using materialized views. In particular, there are two related aspects of the problem that have yet to be adequately investigated: (1) what is an effective way to organize materialized TPQ views? and (2) how to exploit the organization of the materialized views to efficiently evaluate TPQs?

To illustrate these two key issues and motivate our work, let us consider the state-of-the-art approach, InterJoin, for evaluating queries using materialized views. InterJoin is designed for path queries, which is a subclass of TPQs without any branching. Thus, InterJoin only considers materialized path views that correspond to path queries[1]. InterJoin uses a *tuple* scheme to store materialized views: given a path view v with n nodes, v is materialized as a sequence of records, each of which is an n-tuple $(e_1, \cdots, e_n)$ representing a specific path of n element instances in the data that match v. The tuples in v are sorted in ascending order of the composite key $(e_1.start, \cdots, e_n.start)$, where $e_i.start$ represents the document-order rank of element e_i in the data. The *tuple* storage scheme in InterJoin is a generalization of the *element* storage scheme used by the conventional structural join algorithms (e.g., [1]), which essentially partitions all the element instances in the data based on their element types, and stores each subset of instances as a single-element view. The InterJoin algorithm for evaluating path queries using path views is an extension of the structural join algorithm, PathStack [1]. However, the InterJoin algorithm is more intricate than the PathStack algorithm due to interleaving path views (e.g., evaluating the path query $//a//b//c//d$ using the two views $//a//c$ and $//b//d$). The experimental results in [22] show that by exploiting materialized views for evaluating path queries, InterJoin can outperform PathStack by up to a factor of 1.5.

Clearly, as the materialized views already have some structural joins precomputed, using appropriate materialized views (instead of the raw data element views) can help improve query evaluation performance. However, one drawback of InterJoin's tuple scheme is that there could be a lot of data redundancy in a materialized view when the same element instance contributes to many matches in the view. This data redundancy not only increases the I/O cost of accessing the view, but also incurs unnecessary computational overhead when joining multiple views. One obvious approach to eliminate the data redundancy is to "denormalize" the tuple storage scheme into the element storage scheme. Specifically, in the element storage scheme, an n-node view is materialized as a collection of n single-element views without any duplicate element instances within each single-element view. However, the tradeoff for the element scheme is that the structural joins in the original view are no longer explicitly materialized and additional processing cost is required to compute them during query evaluation.

To better understand the performance tradeoffs between InterJoin's tuple scheme and the conventional element scheme, we conducted a performance evaluation (details given in Section VI-A) to compare InterJoin and PathStack, where the materialized views are stored using tuple and element schemes

[1]Following the established line of work on structural joins (e.g., [1], [4]), all the nodes in both path queries and views are output nodes.

respectively[2]. Our experimental results indicate that there is no clear winner between the two approaches. When each data node appears in at most one match in a view, InterJoin outperforms PathStack by up to a factor of 3.5; however, when data nodes can occur in multiple matches of a view, PathStack outperforms InterJoin by up to a factor of 2.5.

Motivated by the unsatisfactory outcome of our experimental comparison between the tuple and element schemes, we seek to probe more deeply the design of storage schemes and evaluation algorithms for view-based TPQ evaluation. Although the design choices for physical storage are not large in general, we believe this area deserves more attention as demonstrated by the recent interest in column-stores vs. row-stores [24] for relational database systems.

In this paper, we address the two aforementioned physical optimization issues and make the following contributions.

1) We propose a new storage scheme for materialized XML views called *linked-element* (`LE`) that combines the complementary strengths of the tuple and element schemes. As the name implies, linked-element is an extension of the element scheme that adds additional element links to preserve the precomputed joins in the tuple-based scheme.
2) We present a novel view-based TPQ evaluation algorithm, `ViewJoin`, that can be used in combination with both the element scheme and the linked-element scheme. `ViewJoin` is a more general approach than InterJoin: it can evaluate general TPQs using materialized TPQ views beyond the path-based queries and views supported by InterJoin.
3) We propose a cost-based model and heuristic to the view selection problem, which takes into account of both the size of the materialized views and the interleaving relationships between the tree patterns of the query and the views.
4) We present the results of a comprehensive experimental evaluation comparing seven combinations of view storage schemes and evaluation algorithms as summarized in Table I where $\texttt{LE}_\text{p}$ refers to a variation of the `LE` scheme. Our experimental results demonstrate the superior performance of `ViewJoin` over both InterJoin and TwigStack by up to a factor of 4.6 and 5.8 respectively (the average gain is 2.2 and 2.5 times respectively).

TABLE I
XML VIEW STORAGE SCHEMES AND EVALUATION ALGORITHMS

Storage scheme	**Evaluation algorithm**		
	InterJoin [22]	TwigStack [4]	ViewJoin (this paper)
Tuple	[22]	-	-
Element	-	[4]	this paper
`LE` (this paper)	-	this paper	this paper
$\texttt{LE}_\text{p}$ (this paper)	-	this paper	this paper

[2]Note that our comparison differs from that in [22] which was comparing InterJoin (with views) against PathStack (without views).

II. Preliminaries

Data labelling. An XML document is represented as a tree. Following the most popular region labelling scheme [18], each node in an XML data tree is assigned a 3-tuple label <*start*, *end*, *level*>, where 'start' and 'end' values are determined by the positions of the start tag and end tag of the node respectively, and the 'level' value is the length of the path from the root node to this node. Using this node labelling scheme, the structural relationship between any two nodes in the same XML tree can be determined efficiently. A node a is an $ancestor$ of node b iff $a.start < b.start$ and $b.end < a.end$; in addition, if $a.level = b.level - 1$, then a is the $parent$ of b. A node a' is a $following\ node$ of node a if $a'.start > a.end$.

Tree pattern queries. Queries and views considered in this paper are tree pattern queries (TPQs) corresponding to the XPath fragment that uses only operators in $\{/, //, []\}$. For simplicity, we assume that the TPQs (including views) have no duplicate element types and that views used to answer a TPQ have no element types in common; details on handling the general case are given elsewhere [5]. TPQs are represented by trees, where the nodes of a TPQ Q are labelled by element types from a finite alphabet. The size of Q, denoted by $|Q|$, is the number of nodes in Q. The nodes in Q are connected by two types of edges: parent-child edges (pc-edges) and ancestor-descendant edges (ad-edges).

Evaluation of TPQs requires matching tree patterns against an XML data tree t. An embedding of a TPQ Q onto an XML data tree t is a mapping β from the nodes of Q to the nodes of t, which satisfies the following conditions.

1) Preserve node types: for each node $u \in Q$, u and $\beta(u)$ are of the same type;
2) Preserve structural relationships: if v is a pc-child (ad-child resp.) of u in Q, then $\beta(v)$ is a child (descendant resp.) of $\beta(u)$ in t.

Each embedding corresponds to a tree pattern instance comprising of a data node $\beta(u)$ for each node $u \in Q$. Following the established line of the work on structural and twig joins [1], [4], [6], the answer to a TPQ Q is the set of the tree pattern instances corresponding to all the embeddings of Q in db (i.e., each node in Q is an output node).

A data node $n \in db$ is a $solution\ node$ of a TPQ Q iff n is in some tree pattern instance matching Q. A solution (data resp.) node of element type t is called a t-type solution (data resp.) node.

A TPQ Q_s is a $subpattern$ of another TPQ Q if there exists a mapping β' from the nodes of Q_s to the nodes of Q, which satisfies the following conditions.

1) Preserve node types: for each node u in Q_s, u and $\beta'(u)$ are of the same type;
2) Preserve structural relationships: if v is a pc-child (ad-child resp.) of u in Q_s, then $\beta'(v)$ is a pc-child (descendant resp.) of $\beta'(u)$ in Q.

Moreover, if Q_s is a connected component of Q, then Q_s is called a $connected\ subpattern$ of Q.

Views. Let V be a set of views, where each view in V is a TPQ. If Q can be answered using views contained in V, we say that V is a covering view set of Q. Specifically, V is a covering view set of Q iff for each query node q in Q, there is a view v in V such that v has a node having the same element type as q and v is a subpattern of Q. A covering view set V of Q is a minimal covering view set of Q if V_s is not a covering set of Q for any $V_s \subset V$. We denote $|V|$ as the number of views in V.

Example 2.1 Consider a TPQ Q in Fig. 1(b), and a set of views $V = \{v_1, v_2, v_3\}$ in Fig. 1(c). Each view in V is a subpattern of Q, but only v_2 and v_3 are connected subpatterns of Q. View v_1 is not a connected subpattern of Q as the ad-edge between nodes a and e in v_1 is not in Q. V is a minimal covering view set of Q. ■

III. Linked-element storage

In this section, we propose a new storage scheme, called linked-element (LE) scheme that combines the complementary strengths of the two existing storage schemes (i.e., tuple scheme and element scheme) for materialized views.

A materialized view stored in the LE scheme is conceptually a directed acyclic graph (DAG). For the ease of presentation, we first present the conceptual DAG structure.

A. Conceptual DAG Structure

For a view pattern v, we use T_v to denote its materialized view w.r.t. an XML data tree T; for a node n in T_v, we use q_n to denote n's corresponding query node in v.

The materialized view T_v can be represented as a DAG structure, where each node n in T_v has the following pointers.

1) a *child pointer* for each pc-child (ad-child) query node q_i of q_n in v: linking n to node n_i in T_v where n_i is the q_i-type child (descendant resp.) of n with the smallest $start$ label.
2) a $descendant\ pointer$: linking n to node n'' in T_v where n'' is the q_n-type descendant node of n with the smallest $start$ label. If node n'' does not exist, the descendant pointer is a null pointer.
3) a $following\ pointer$: linking n to node n' in T_v where n' is the q_n-type following node of n with the smallest $start$ label; if q_n in v has a parent query node α, n and n' are required to have the same lowest α-type ancestor in T_v. If node n' does not exist, the following pointer is a null pointer.

Each node n in T_v has one $descendant$ pointer, one $following$ pointer, and one $child$ pointer for each child query node of q_n in v. A descendant or following pointer points to a node of the same type, while a child pointer points to a node of a different type. The child, descendant and following pointers are represented using downward single arrows, downward double arrows, and horizontal dashed arrows respectively. Through these pointers, the DAG structure captures the structural relationships among the nodes of different types in the view.

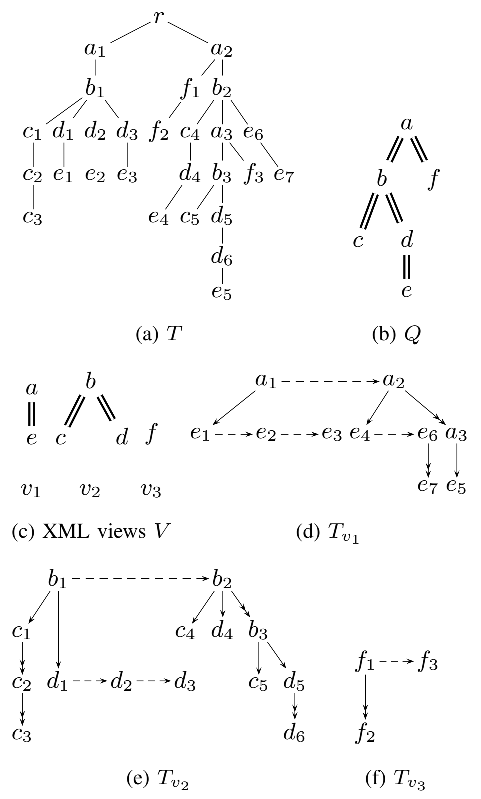


Fig. 1. Conceptual DAG structure for materialized views (null pointers are not shown)

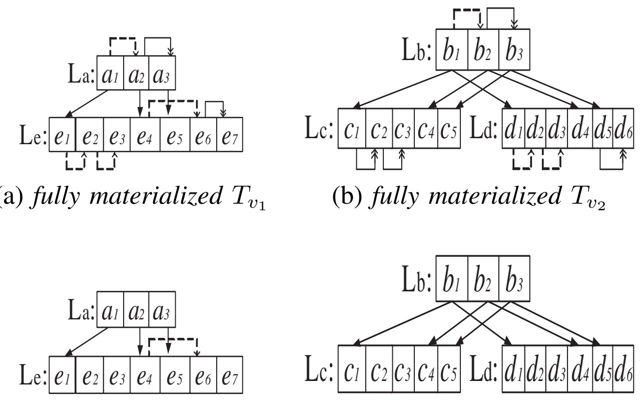
Example 3.1 Consider a TPQ Q (Fig. 1(b)) and an XML document T (Fig. 1(a)). For convenience, subscripts are added to the data nodes in T to distinguish nodes of the same element type. Consider the views v_1, v_2 and v_3 on T as shown in Fig. 1(c). The DAGs corresponding to the materialized views w.r.t. T are shown in Fig. 1(d)-(f) respectively. While the it is straightforward to understand the child and descendant pointers, we give some explanation of the $following$ pointers using T_{v_1} (Fig. 1(d)). There is a following pointer from e_1 to e_2, as e_2 is the $following$ node (e_2.start > e_1.end) of e_1 with the smallest start label, and meanwhile a_1 is the lowest a-type ancestor of e_1 and of e_2 respectively. Similarly, there is a following pointer from e_2 to e_3, and e_4 to e_6. Note that there is not a following pointer from e_4 to e_5, as e_4's lowest a-type ancestor (a_2) and e_5's lowest a-type ancestor (a_3) are not the same node. For similar reason, there is not a following pointer from e_3 to e_4. ■

B. Linked-Element (LE) Scheme

Consider a view v and its materialized result T_v, which is conceptually a DAG structure. We propose a storage scheme called linked-element (LE) to store T_v as multiple lists of nodes: one list L_q for each query node $q \in v$. L_q contains all the q-type nodes in T_v, sorted in document order.

A node n in L_q is stored as a tuple containing its positions and pointers, <$start$, end, $level$, f, d, s_1, ..., s_m>, where f and d are the following pointer and descendant pointer, and

s_1, ..., s_m are the child pointers of n. Each pointer stores the position of the pointed node within the respective list, comprising of a disk page number and a byte offset within the page.


La: a_1 a_2 a_3
Le: e_1 e_2 e_3 e_4 e_5 e_6 e_7
Lb: b_1 b_2 b_3
Lc: c_1 c_2 c_3 c_4 c_5
Ld: d_1 d_2 d_3 d_4 d_5 d_6

(a) *fully materialized* T_{v_1} (b) *fully materialized* T_{v_2}

(c) *partially materialized* T_{v_1} (d) *partially materialized* T_{v_2}

Fig. 2. LE/LE$_p$ lists for DAGs in Fig. 1

Example 3.2 The DAGs in Fig. 1(d)&(e) are stored as multiple linked-element lists as shown in Fig. 2(a)&(b). Nodes in each list are sorted in document order. ■

The LE scheme has the following advantages.

(1) Compact space to capture view result. The linked-element scheme captures the view result in a more compact way than the tuple scheme if a node appears in multiple view matches. Unlike the tuple scheme, there is no need to keep duplicate copies of the node. Similar to the tuple scheme, it has the advantage of the precomputed structural joins of the view patterns.

(2) Skipping non-solution nodes. Evaluation of a query with views stored in LE requires a single scan of the involving lists. The LE scheme allows skipping of non-solution nodes in query evaluation. Consider evaluating query Q in Fig. 1(b) using views v_1, v_2 and v_3 in Fig. 1(c). Node a_1 does not lead to any query match of Q as a_1 has no matching f-type descendant due to a_1.end $<$ f_1.start. We can then advance a_1 to a_2 via the $following$ pointer. Examining the data tree T in Fig. 1(a), we know that all the nodes in the subtree rooted at a_1 are non-solution nodes of Q. With materialized pointers, all the nodes under $a_1 \in T$ can be skipped from processing.

(3) Reduced list accessing cost. With the LE scheme, we can evaluate a query Q using views as follows. First, evaluate a subpattern Q_s of Q. Second, access the matching entries of query nodes not in Q_s by pointers. Consider evaluating a TPQ Q = *//a//b*[*//c/d*]*//e* with views *//a*, *//b*[*//c/d*], and *//e*. We can first evaluate a subpattern Q_s = *//a//b//e* by accessing only the materialized lists corresponding to a, b and e. From the answer of Q_s, we then access the matching entries from the lists corresponding to c and d by the pointers, instead of scanning the entire lists of c and d. When the answer to Q_s is small in size or empty, the reduced cost in accessing c and d lists can be significant.

There are other options to store the materialized view T_v. However, these options do not have some of the advantages described above. For instance, we can store T_v as one integrated list consisting of all the nodes in the DAG structure and the associated pointers, but this would not have the advantage (3). Another alternative is to partition the DAG structure into disk pages with (or without) pointers, but this would not have the advantages (3) (or (2)-(3) resp.).

C. Partial Linked-Element (LE$_p$) Scheme

The materialization of the pointers in our proposed scheme effectively captures the structural matchings of the view pattern. However, not every materialized pointer is beneficial for query evaluation.

Consider again the example of evaluating query Q in Fig. 1(b) using views v_1, v_2 and v_3 in Fig. 1(c). The materialized views of v_1, v_2 and v_3 are shown in Figs. 1(d), 1(e) and 1(f). Node a_1 is not a solution node as the first node f_1 in the linked-element list of f (an ad-child of a) has a $start$ label greater than the end label of a_1. Since none of the descendant nodes of a_1 is a solution node, the a-type node is then advanced to a_1's first *following* node a', which could be potentially a solution node of Q.

With the $following$ pointer, the cost of finding a', denoted as c_p, is the sum of c_1 and c_2, where c_1 is the cost of processing a_1's following pointer to get the position of a', and c_2 is the cost of accessing a'. Without the $following$ pointer, the cost of finding a', denoted as c_o, is equal to $d * (c_3 + c_4)$ where c_3 is the cost of accessing the next entry a_k in L_a, c_4 is the cost of comparing a_k.start versus a_1.end, and d is the distance from a_1 to a' counted in the number of entries in L_a.

In the above example, a' is the adjacent entry of a_1, i.e., d = 1. In this case, the advantage of using pointers may not be reflected as the avoided comparison cost may be offset by the cost of processing the pointer. However, the advantage of materializing the following pointer is expected to be more significant when d gets larger.

We propose a variant of the LE scheme, called *partial linked-element* (LE$_p$) scheme, where pointers are partially materialized using the following heuristic rules.

1) If a pointer is a $child$ pointer (which points to a node in a different list), it is materialized;
2) If a pointer is a $following$ or $descendant$ pointer, it is materialized if the pointed node is more than one entry away in the respective list.

The LE$_p$ lists with partial materialization of pointers are shown in Fig. 2(c)&(d).

IV. ViewJoin algorithm

In this section, we examine how to efficiently evaluate a query Q given a minimal covering view set V stored in LE or LE$_p$ scheme. For simplicity of discussion, we first present a solution for LE views, and then we discuss the variation of the solution for LE$_p$ views. In this section, Q refers to a TPQ and V refers to a minimal covering view set of Q where each view in V is stored using the linked-element scheme.

A. *View-Segmented Query*

Let v_s be a connected subpattern of a view $v \in V$. If v_s is also a connected subpattern of Q, then the join operations corresponding to the edges in v_s can be avoided in the evaluation of Q, as the join result of the connected subpattern v_s is precomputed in the materialized views of the linked-element scheme.

Thus it is desirable to decompose the tree pattern of Q into segments such that each segment is a connected subpattern of some view v in V. The join operations corresponding to the edges within a segment are avoided when evaluating Q using V. Evaluating Q can then be transformed into the joining of the segments. This is more efficient as the number of segments is generally smaller than the query nodes in Q. This motivates the concept of *view-segmented query* as follows.

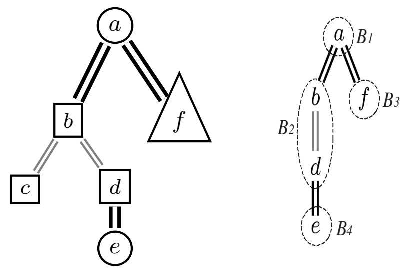
An edge $e = (n_1, n_2) \in Q$ is an *inter-view edge* w.r.t. V if query nodes n_1 and n_2 are covered by two different views in V; otherwise, e is an *intra-view edge*. The view-segmented query Q' can be formed from Q in two steps. First, remove the non-root nodes with no incoming or outgoing inter-view edges. If a non-leaf node $q \in Q$ is removed in this step, then we connect each child node of q to q's parent node with an ad-edge which is treated as an intra-view edge. Second, group together the nodes that are connected by intra-view edges. The set of nodes grouped together is called a *segment*.

Each segment B by itself is a tree pattern and the root of the tree pattern is the root node of the segment, denoted as $r(B)$. If $r(B)$ in Q' has a parent query node which belongs to another segment B', then we call B a *child segment* of B'. Q' has a *root segment*, which is the segment containing the root node of Q. The view-segmented query Q' can be obtained in linear time in the size of Q.

Example 4.1 Consider the same TPQ Q and views in Fig. 1(b)&(c). To depict the interleaving conditions of Q w.r.t. the views, we annotate its query nodes using different shapes as shown in Fig. 3(a): nodes from v_1, v_2, v_3 are shown in circles, boxes, triangles respectively. The inter-view edges are shown in bold: $(a,\ f)$, $(a,\ b)$ and $(d,\ e)$. Node c has no inter-view edges. The view-segmented query Q' of Q is shown in Fig. 3(b): it has four segments, $B_1 = a$, $B_2 = b//d$, $B_3 = f$ and $B_4 = e$; and the root segment is B_1. ■

B. *ViewJoin Evaluation Algorithm*

With the concept of the view-segmented query, we propose an algorithm, *ViewJoin* (Algorithm 1) to efficiently evaluate Q on V in a two-step approach. First, it evaluates the corresponding view-segmented query Q'; this step computes the query result involving only the nodes in Q'. Second, it extends the result of the first step to obtain the matching nodes of the query nodes that are in Q but not in Q' using the materialized pointers corresponding to intra-view edges. Although *ViewJoin* shares the same spirit as the twig join algorithms (e.g., [4], [6]), it has the following non-trivial extensions and unique features.



(a) Query (b) View-segmented query

Fig. 3. View-segmented query for query Q w.r.t. views v_1, v_2, v_3 in Fig. 1

1) It processes the materialized views stored in linked-element scheme instead of element streams;
2) It keeps the intermediate solutions in the view DAG structure (introduced in Section III-A) unlike [4], [6] which use a stack structure. It thus provides a solution for storing the query result as a materialized view;
3) It processes with segments rather than query nodes of the query, which is more efficient as the number of segments is usually smaller than the query nodes;
4) It can skip processing some non-solution nodes in the input view lists, unlike [4], [6] which process every node in the input streams.

Like [4], [6], our *ViewJoin* algorithm may not guarantee space optimality when pc-edges are present for twig queries. However, space optimality is guaranteed for twig queries/views with pc-edges so long as all the inter-view pc-edges in Q w.r.t. V are in the same root-to-leaf path of Q.

Overview of Algorithm ViewJoin. *ViewJoin* computes the solution nodes in document order via the *getNext* function (Function 3), and builds the DAG structure F with the solution nodes via the *addNodes* function (Function 2). Each call of the *getNext* function computes the next solution node with the smallest $start$ label. The *addNodes* function adds nodes to F in a top-down order: descendant nodes are never added to F before its ancestor nodes. If the next node n to be added to F is not under the root node n_r of F, then the solutions in F are outputted via the second step, which computes the query matches embedded in F via a single pass of F, and then F is re-initialized to a new DAG with root n.

When there are inter-view pc-edges in the query Q, algorithm *ViewJoin* builds the DAG structure F by assuming these edges are ad-edges. The pc-relationships are checked when outputting solutions from F by the $level$ labels.

For each node $q \in Q'$, *ViewJoin* maintains a cursor C_q to represent the next node to be processed in the corresponding $\texttt{LE}$ list L_q, and a pointer T_q to point to the last q-type node added to F. For a node n in a materialized view, $q(n)$ denotes the query node in Q that matches n. For a query node $q \in Q$, $st_Q(q)$ denotes the subtree of Q rooted at q; sol_q denotes the q-type solution node which is not yet added to F.

Algorithm 1 ViewJoin (minimal covering view set V, TPQ Q)

```
 1: compute the view-segmented query $Q'$;
 2: initialize $C_q$ to first entry in $L_q$ for each $q \in Q'$;
 3: initialize $T_q$ to null for each $q \in Q'$;
 4: $F$ = new empty DAG;
 5: while ($n$ = getNext(root segment of $Q'$)) $\neq$ null do
 6:    if $q(n) = r(Q')$ then
 7:       if $n_r$ is null then
 8:          $n_r$ = $n$;
 9:       else if $n.start > n_r.end$ then
10:          extend $F$ to cover nodes in $Q$ via pointers;
11:          output matches in $F$;
12:          $n_r$ = $n$;
13:          $F$ = new empty DAG;
14:    $F$.addNodes($n$);
```

function 2 addNodes (node n)

```
 1: for all $q \in st_{Q'}(q(n))$ in top-down order do
 2:    let $p$ = the parent node of $q$ in $Q'$;
 3:    if $q \neq r(Q') \wedge C_q.start > C_p.start$ in $Q'$ then
 4:       set $sol_q$ = $C_q$;
 5:       break; //exit for loop
 6:    else
 7:       add node $C_q$ to $F$;
 8:       set $T_q$ = $C_q$;
 9:       $C_q \rightarrow$ next entry in $L_q$;
```

function 3 getNext (segment B)

```
 1: if isLeaf$_{Q'}(B)$ or $C_{r(B)} = sol_{r(B)}$ then
 2:    return $C_{r(B)}$;
 3: initialize $S$ to an empty set;
 4: for each child segment $B_s$ of $B$ in $Q'$ do
 5:    $n_s$ = getNext($B_s$);
 6:    if $n_s$ is null then
 7:       return null;
 8:    if $q(n_s) \neq r(B_s)$ then
 9:       add $n_s$ to $S$;
10:    else if $n_s.start < C_p.start$ then
11:       let $p$ = the parent node of $q(n_s)$ in $Q'$;
12:       if $n_s$ has a $p$-type ancestor in $F$ then
13:          add $n_s$ to $S$;
14:       else
15:          while $C_{q(n_s)}.start < C_p.start$ do
16:             $C_{q(n_s)} = next\ entry\ in\ L_{q(n_s)}$;
17:          return getNext($B$);
18:    else if $n_s.start > C_p.end$ then
19:       advancePointers($r(B)$, $q(n_s)$);
20:       return getNext($B$);
21:    else
22:       add $n_s$ to $S$;
23: add non-null $C_{q_i}$ to $S$ for each $q_i \in B$;
24: $n_{min}$ = minarg$_{n_s \in S}$ ($n_s.start$);
25: return $n_{min}$;
```

function 4 advancePointers (query node q, q_s)

```
 1: while $C_q.end < C_{q_s}.start$ do
 2:    $C_q \rightarrow C_q$'s linked node via $following\ pointer$;
 3: let $v_q$ = the view containing $q$;
 4: for all node $q_i \in st_q(Q')$ in top-down order do
 5:    let $p_i$ = the parent node of $q_i$ in $Q'$;
 6:    if $q_i \in v_q$ then
 7:       $C_{q_i} \rightarrow C_{p_i}$'s linked node by $q_i$-type $child\ pointer$;
 8:    else
 9:       if $q_i$ is a root node of some segment then
10:          while $C_{q_i}.start < C_q.start$ do
11:             $C_{q_i} \rightarrow$ next entry in $L_{q_i}$;
12:       else
13:          $C_{q_i} \rightarrow C_{p_i}$'s linked node by $q_i$-type $child\ pointer$;
```

We illustrate the *ViewJoin* algorithm with examples with focus on the unique features of the two important functions: the addNodes function and the getNext function.

(i) Segment-level processing. In traditional twig join algorithms (e.g., [4]), the recursive function getNext iterates with the query nodes in a top-down order. The number of the recursive iterations before finding the next solution node is equal to the number of nodes in the query. Our getNext function works in a similar way, but in contrast, it iterates with the segments instead of query nodes. The number of recursive iterations before finding the next solution node is bounded by the number of segments in the view-segmented query Q'. As the number of segments is generally smaller than the number of query nodes, our getNext function is expected to take fewer iterations before finding the next solution node. Note that structural comparisons are performed only for inter-view edges in Q' (between $r(B_s)$ and its parent node p) in lines 10-22.

(ii) Adding multiple solution nodes at a time. For a solution node n returned by the getNext function, C_q for each query node $q \in st_{Q'}(q(n))$ is also a solution node. The reason is the following. As a solution node, n must have an identified matching node n_q for every query node $q \in st_{Q'}(q(n))$; now n_q must not have been added to F due to the top-down order of constructing F. Thus the matching node n_q is C_q. Note that when C_q is a solution node, the cursor is not advanced unless it is already added to F.

The addNodes function adds solution nodes to F as follows. For each query node $q \in st_{Q'}(q(n))$ in top-down order, if q is the root node of Q', C_q is added to F. Otherwise, if $C_q.start$ $< C_p.start$ where p is the parent query node of q in Q', C_q is added to F. If $C_q.start > C_p.start$, C_q is not added to F as doing so might violate the top-down order of constructing F; instead, set sol_q to C_q as a $cached\ solution\ node$ which is not added to F yet. The cursor C_q is not advanced if C_q is cached. The cached solution nodes avoid re-computing them using the getNext function. When the getNext function encounters a query node q for which C_q = sol_q, it then returns C_q without further processing.

(iii) Skipping non-solution nodes. With the LE scheme, the $getNext$ function may skip some non-solution nodes from processing. Consider a segment B with a child-segment B_s. Let p be the parent node of the root node of B_s. Apparently p is in segment B. Let n_s be the node returned by getNext(B_s). C_p matches n_s iff $C_p.start < n_s.start$ and $n_s.end < C_p.end$. If $n_s.start > C_p.end$ (line 18), then C_p has no matching $q(n_s)$-type descendant node and is thus a non-solution node. In this case, pointers are advanced via the *advancePointers* function

(Function 4) to skip nodes that are guaranteed to be non-solution nodes. We illustrate the advancePointers function by the following example.

Example 4.2 Consider the same view-segmented query Q' in Fig. 3(b) with the linked-element lists in Fig. 2(a)&(b). We show how getNext(B_1) skips non-solution nodes where B_1 is the root segment of Q'. Initially, we have $C_a = a_1$, $C_b = b_1$, $C_d = d_1$, $C_e = e_1$, and $C_f = f_1$. Recursive call of getNext(B_3) returns $C_f = f_1$. As f_1.start $> a_1$.end implies that a_1 has no matching f node, a_1 is thus not a solution node. The advancePointers function advances C_a to a_2 via its following pointer (lines 1-2 of advancePointers). Node a in Q' has four descendant query nodes: b, d, e and f. For node b which is the root node of segment $B2$, C_b is advanced to the next entry $b_2 \in L_b$; for $d \in B_2$, C_d is set to d_4 via the child pointer of b_2 skipping all the nodes in L_d between d_1 and d_4 in L_d; for $e \in B_1$, C_e is set to e_4 via child pointer of a_2 skipping all the nodes between e_1 and e_4 in L_e. ■

We now use the following example to illustrate how algorithm *ViewJoin* evaluates query Q in Fig. 3(a) with the views in Fig. 1.

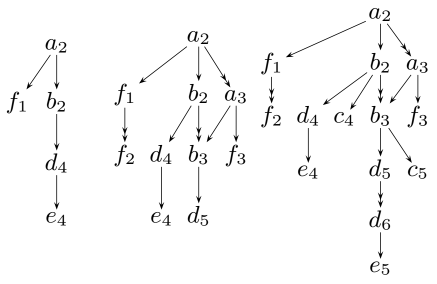
Example 4.3 Cont'd with Example 4.2. Now we have $C_a = a_2$, $C_b = b_2$, $C_d = d_4$, $C_e = e_4$, $C_f = f_1$. In this iteration of getNext(B_1), recursive calls getNext(B_2) and getNext(B_3) return b_2 and f_1 respectively, which are both descendants of $C_a = a_2$. Both b_2 and f_1 are added to the set S. Finally we have $S = \{a_2, b_2, f_1\}$. Node a_2 with the smallest start label is then returned as the first solution node.

The addNodes function tries to add to F the current cursors b_2, d_4, e_4 and f_1 for the query node below a in Q', which are also solution nodes. We initialize F and add these solution nodes to F in top-down order, and update these cursors to the next entry in the respective lists. Fig. 4(a) shows the current status of F.

Node cursors are now as follows, $C_a = a_3$, $C_b = b_3$, $C_d = d_5$, C_e=e_5 and $C_f = f_2$. The solution node returned by the *getNext* function is f_2. The *addNodes* function adds only f_2 to F as f is a leaf node in Q', and updates C_f to f_3. The next solution node returned is a_3. Similarly, b_3, d_5, e_5 and f_3 are also solution nodes. All these solution nodes are added to F except e_5, as e_5 is a descendant of the updated node cursor $C_d = d_6$, which is potentially a solution node. Fig. 4(b) shows the current status of F.

The next solution node returned is d_6. Now d_6 and e_5 are added to F. There are no more solution nodes returned by getNext. There is a node $c \in Q$ which is not in Q'. For each of the b-type node $n_b \in F$, add the linked c-type nodes via *child* pointers. The final F for Q is shown in Fig. 4(c). ■

Variations of the ViewJoin algorithm. So far we have presented the ViewJoin algorithm by assuming that the entire solution for the query can be kept in the memory, which we call the *memory-based* approach. In the case that the memory space is insufficient to keep the solution, we propose an variation of the ViewJoin algorithm, which we call the *disk-based*



(a) F: status 1 (b) F: status 2 (c) F: status 3

Fig. 4. Running example of ViewJoin

approach. The disk-based approach outputs the intermediate query solutions and then read them into the memory again to compute the query matches. More specifically, the disk-based approach outputs the portion of the DAG F that is comprised of nodes with their *end* labels smaller than the *start* label of the current solution node returned by the getNext function. The disk-based approach uses less memory space than the memory-based approach.

Complexity results. The number of comparisons that a node n in a materialized view T_v participates in is bounded by the number of the inter-view edges connected to the corresponding query node in Q'. We then have the time-complexity: $O(\sum_{q \in Q} |L_q| * e_q + |output|)$ where e_q is the number of inter-view edges of q in Q.

The space complexity depends on the variations of the ViewJoin algorithm. The space needed for the memory-based approach is $O(|F_{max}|)$ where F_{max} is the largest F computed. The size of F_{max} can be as large as the entire output. In this approach, the space needed for the disk-based approach is $O(|Q| * |l|)$, where l is the longest root-to-leaf path in the data. ■

For algorithm *ViewJoin*, we have the following result. Note that the space complexity in Lemma 4.1 is for the disk-based approach.

Lemma 4.1: For a given TPQ Q and a minimal covering view set V in linked-element scheme, algorithm *ViewJoin* computes all the matches of Q on V in $O(\sum_{q \in Q} |L_q| * e_q + |output|)$ time where L_q is the element list of q and e_q is the number of inter-view edges of q in Q. The I/O complexity is $O(\sum_{q \in Q} |L_q| + |output|)$. The space complexity is $O(|Q| * |l|)$ where l is the longest root-to-leaf path in the data.

Lemma 4.1 shows that the size of the views as well as the complexity of the interleaving conditions between Q and the views in V are important determinants of the evaluation cost.

Evaluation algorithm for views in $\texttt{LE}_\texttt{p}$ **scheme.** Algorithm *ViewJoin* can evaluate a query using views in $\texttt{LE}_\texttt{p}$ scheme via some minor modifications of the advancePointers function as follows. Before Step 2 of the advancePointers function (Function 4) which intends to advance a cursor C_q via its following pointer p, add an additional step to check whether p is materialized. If p is not materialized, C_q is advanced to the next entry in L_q. The modified *ViewJoin* algorithm evaluates a given query using views in $\texttt{LE}_\texttt{p}$ scheme with the same time and space complexity results.

V. VIEW SELECTION

In this section, we examine the view selection problem which is defined as follows. Given a set of materialized views V and a TPQ Q, find a subset V_s of V such that (1) V_s answers Q and (2) there does not exist a proper subset V_s' of V that is more efficient than V_s for answering Q w.r.t. a cost model. The view selection problem is a NP-complete problem.

In the following, we present an evaluation cost model for the `ViewJoin` algorithm that takes account of both the size of the materialized views as well as the interleaving conditions between the view patterns and the query pattern.

Cost model. Consider a TPQ Q and a view v which is a subpattern of Q. Let $c(v, Q)$ denote the cost of evaluating Q using v, which is the sum of two components: the I/O cost of accessing the view (denoted by c_1) and the CPU cost of the join (denoted by c_2). The I/O cost c_1 is measured by the size of the materialized view of v, i.e., $c_1 = \sum_{q\in v} |L_q|$ where $|L_q|$ denotes the size of the q-type list. The join cost c_2 is measured as follows. For each query node $q \in v$, we measure its join cost as $|L_q| * e_q$ where e_q denotes the number of the edges of q in Q that are not present in v. We then define c_2 as the sum of the join cost of all the nodes in v, i.e., $c_2 = \sum_{q\in v} |L_q| * e_q$. For an edge (p, q) in Q but not present in v, the sizes of both L_p and L_q are taken into account in measuring the join cost c_2. The total cost $c(v, Q)$ is a weighted sum of the two components given by $c(v, Q) = (1\text{-}\lambda)\sum_{q\in v} |L_q| + \lambda\sum_{q\in v} |L_q| * e_q$, where λ is a weight parameter, $\lambda \in [0, 1]$.

TABLE II
VIEW SELECTION FOR Q = //DATASET//TABLEHEAD[//TABLELINK//TITLE]//FIELD//DEFINITION//PARA

	View	Size (MB)	$c(v, Q)$
v_1	//dataset//definition	0.88	1.76
v_2	//dataset//tableHead	0.14	0.17
v_3	//field//para	0.73	1.01
v_4	//definition	0.83	1.66
v_5	//tableLink//title	0.37	0.20
v_6	//field//definition//para	0.97	0.27

Our experimental results show that query evaluation is CPU bound. Thus we set λ = 1 to obtain a good approximation.

Heuristic for view selection. Given the above cost model, we can apply the greedy heuristic in [13] to solve the view selection problem. Note that if v is not a subpattern of Q, v is removed from consideration as it cannot be used in answering Q.

We denote V_m as the set of selected views, which is initially empty. Each unselected view $v \in V - V_m$ has a benefit given by $\frac{N_v}{c(v,Q)}$ where N_v is the set of query nodes in Q that are not covered by any view in V_m and covered by v.

The greedy heuristic iteratively selects the most beneficial unselected view from $V - V_m$ and adds to V_m until (1) all query nodes in Q are covered by the views in V_m or (2) V becomes empty.

If the heuristic terminates under condition (1), then V_m is the minimal view set for Q; otherwise, Q cannot be answered by V. The time complexity of the heuristic is $O(|Q||V|)$.

Example 5.1 Consider a query Q = //dataset//tableHead [//tableLink//title]//field//definition//para and the set of views shown in Table II. Query Q and the views are defined on the Nasa dataset from [20]. The size of the materialized views and the $c(v, Q)$ cost for each view v are also specified in Table II. Note that all the views given are subpatterns of Q. Our proposed heuristic approach will select the view set $\{v_2, v_5\ , v_6\}$. If a heuristic based purely on the size of views is used, the set of views selected would be $\{v_2, v_3, v_4, v_5\}$. Our experimental result in Section VI-B shows that evaluating Q using $\{v_2, v_5\ , v_6\}$ outperforms $\{v_2, v_3, v_4, v_5\}$ by a factor of 1.93. ■

VI. EXPERIMENTAL RESULTS

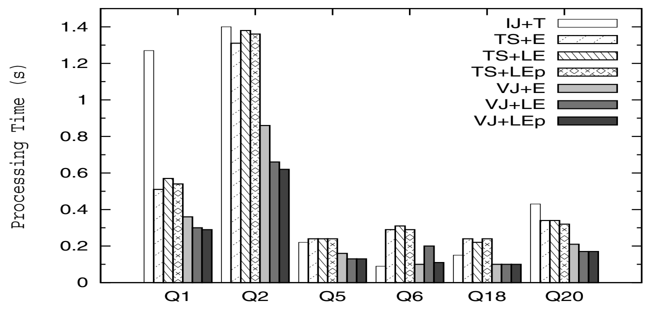
In this section we present our experimental results on the performance of different combinations of view storage schemes and evaluation algorithms. For simplicity, we use `IJ`, `TS`, `VJ` to refer to `InterJoin`, `TwigStack`, `ViewJoin`; and `T`, `E`, `LE`/$\texttt{LE}_\texttt{p}$ to refer to tuple, element, linked-element scheme with full/partial materialization of pointers respectively. For $a \in \{\texttt{IJ}, \texttt{TS}, \texttt{VJ}\}$ and $s \in \{\texttt{T}, \texttt{E}, \texttt{LE}, \texttt{LE}_\texttt{p}\}$, we use $a + s$ to refer to a combination of the algorithm and the view storage scheme.

Table I in Section I summarizes the seven combinations of algorithms and view storage schemes in our experiments. Note that `IJ` works only with tuple views, while `TS` and `VJ` work only with element views and linked-element views. The original `TS` algorithm can only process element views, which we extend to process views in linked-element schemes.

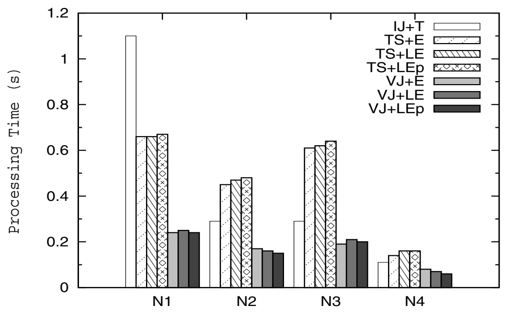
We implemented these algorithms using Java 1.5.0 and performed experiments on a Pentium 4 PC with 3.0Ghz processor and 1GB main memory, and a 30GB hard disk. We ran each experiment five times with the average of the results reported. As introduced in Section IV, both `TS` and `VJ` may have two processing approaches. The experimental results are based the memory-based approach of both `TS` and `VJ` unless otherwise specified.

Datasets and test queries. Both synthetic and real datasets are used for the experimental evaluation. The synthetic datasets are the XMark datasets [23] generated with different scaling factors: the size of the XMark datasets is from 100MB to 700MB. The standard 113MB XMark dataset is used for the experiments unless otherwise specified. The real dataset used is the Nasa dataset from [20] consisting of 23MB data, which was also used in [22] for experimental study.

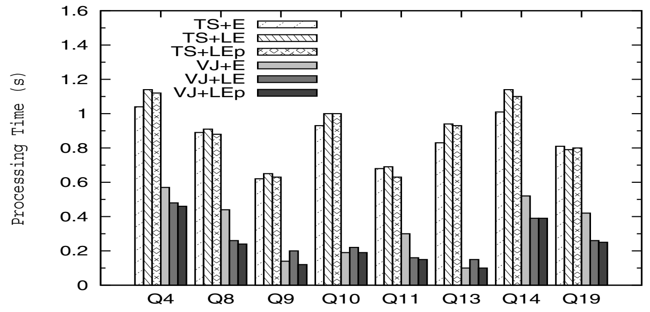
For the XMark datasets, our test queries are derived from XMark's benchmark XQuery queries by removing the features not supported in XPath (e.g., groupings, user-defined functions, re-constructs) and discarding the value-based predicates. We use the 14 derived XPath queries (Q_1-Q_2, Q_4-Q_6, Q_8-Q_{11}, Q_{13}-Q_{14}, Q_{18}-Q_{20}) which do not have OR/NOT-predicates. Out of the 14 benchmark queries on XMark datasets, 6 queries



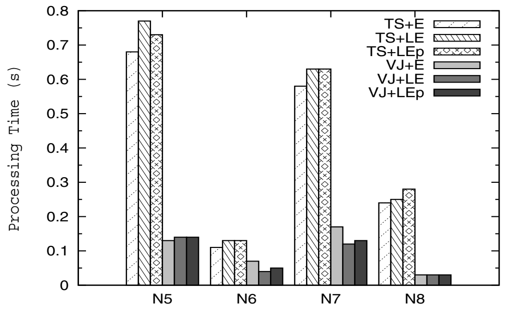
(a) XMark benchmark path queries



(b) Path queries on Nasa

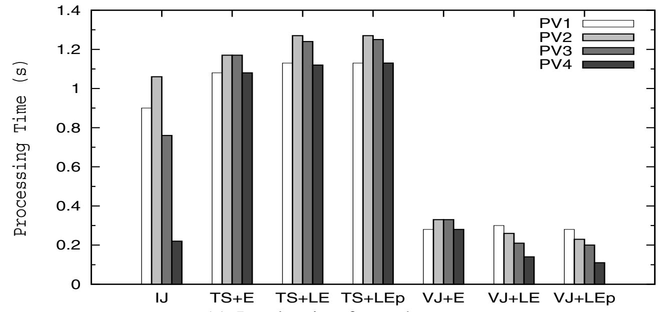


(c) XMark benchmark twig queries

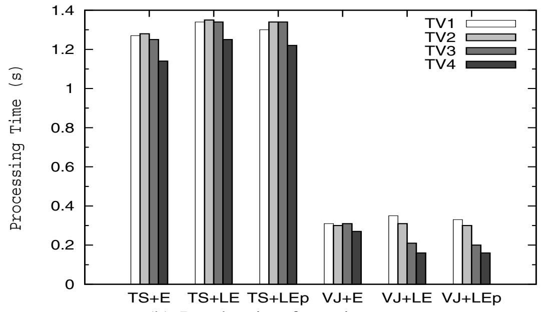


(d) Twig queries on Nasa

Fig. 5. Twig/path queries with twig/path views



(a) Interleaving for path query



(b) Interleaving for twig query

Fig. 6. Impact of interleaving conditions

are path queries and the other 8 queries are twig queries. The detailed queries are given elsewhere [5].

For the Nasa dataset, as there are no benchmark queries available, we generated four path ($N1$-$N4$) and four twig ($N5$-$N8$) queries based on various selectivity.

$N1$ = //field//footnote//para
$N2$ = //dataset//definition//footnote
$N3$ = //revision/creator/lastname
$N4$ = //reference//journal//date//year
$N5$ = //dataset[//definition/footnote]//history//revision//para
$N6$ = //journal[//suffix][title]/date/year
$N7$ = //dataset[//field//footnote]//journal[//bibcode]//lastname
$N8$ = //descriptions[//observatory]description//para

The main observations and findings of our experimental results are summarized as follows.

1) VJ outperforms IJ and TS by a factor up to 4.6 and 5.8, respectively, in terms of the total query processing time.
2) There is no clear winner between IJ and TS. In particular, IJ outperforms TS by a factor of 3.5 if data nodes do not occur in multiple tuples for the tuple scheme, and TS outperforms IJ by a factor of 2.5 if otherwise.
3) The view storage schemes have significant impact on the performance of VJ. Specifically, $\mathrm{VJ+LE_p}$ outperforms VJ+LE by a factor up to 1.8, which in turn outperforms VJ+E by a factor up to 1.9.
4) Our experimental results on XMark datasets show that VJ is scalable in terms of processing time, I/O cost, as well as the memory space usage.

A. Processing Time for Path/Twig Queries

We examine both path queries as well as twig queries on the XMark and Nasa datasets. The experimental results of the path and twig queries are shown in Fig. 5, which compare the total processing time (I/O time + CPU time) of answering a query using different algorithms with materialized views in different storage schemes. Note that the CPU time is the dominant part of the total processing time. For path queries, we compared the performance of all the seven combinations of algorithms and view storage schemes as shown in Figs. 5(a)&(b). Note that `TS` for path queries is the equivalent to the PathStack algorithm [4]. For twig queries, we investigated the performances without `IJ` (Figs. 5(c)&(d)), as `IJ` handles only path-based queries and views. The main observations from Fig. 5 are as follows.

First, `VJ` outperforms `TS` and `IJ` regardless of the view storage scheme. More specifically, `VJ` outperforms `TS` for all the queries on both the XMark and Nasa datasets by a factor of 1.4 to 5.8; for the Nasa dataset with highly skewed element distribution, a higher performance gain over `TS` is observed than the XMark datasets due to a higher benefit in skipping non-solution nodes using pointers. For the path queries, `VJ` outperforms `IJ` (except for query Q6) by a factor of 1.1 to 4.6. For query Q1 and N1, `IJ` performs significantly worse than `VJ` and `TS` due to the high data redundancy in the tuple views. However, for Q6, `IJ` slightly outperforms `VJ` as Q6 is very simple (with only three steps) and the tuple views have no recurring nodes, i.e., each node appears in exactly one tuple.

Second, there is no clear winner between `IJ` and `TS`. As shown in Figs. 5(a)&(b), `TS` outperforms `IJ` for queries Q1, Q2, Q20, and N1 by a factor up to 2.5. The tuple views for these queries have high data redundancy. For the other path queries, `IJ` outperforms `TS` by a factor up to 3.5.

Third, for `VJ`, the best performance is observed with the $\texttt{LE}_\text{p}$ storage scheme. More specifically, for all the queries, $\texttt{VJ+LE}_\text{p}$ outperforms `VJ+LE`: the performance gain is up to a factor of 1.8 for query Q6, Q9 and Q13. For most of the queries, `VJ+LE` outperforms `VJ+E` by a factor up to 1.9. However `VJ+E` outperforms `VJ+LE` by a factor up to 2.0 for query Q6, Q9 and Q13. For these queries, the nodes in the materialized views are evenly distributed and the overhead of pointers offsets its benefit of skipping non-solution nodes. For Q6, Q9 and Q13, `VJ+E` outperforms $\texttt{VJ+LE}_\text{p}$ by a factor of no more than 1.1.

B. The Impact of Interleaving Conditions

This set of experiments examines the impact of interleaving conditions between the tree patterns of the query and the views on the performance of different techniques. We also conducted experiments on evaluating the same query using different sets of views.

Figs. 6(a) & 6(b) compare the processing time for evaluating two queries N_p and N_t on the Nasa dataset with different sets of views as specified in Table III ($PV1$-$PV4$ for N_p and $TV1$-$TV4$ for N_t), where N_p = *//dataset//tableHead//field//definition//footnote//para*, N_t = *//dataset//tableHead[//tableLink//title]//field//definition//para*. The last column of Table III shows the number of inter-view edges in the query w.r.t. the views, which measures the complexity of interleaving conditions between the query and the set of views.

TABLE III
QUERY EVALUATION WITH DIFFERENT VIEWS

	Views	#Cond
$PV1$	//dataset//field//footnote; //tableHead//definition//para	5
$PV2$	//dataset//field//footnote//para //tableHead//definition	4
$PV3$	//dataset//field; //tableHead//definition//footnote//para	3
$PV4$	//tableHead; //dataset//field//definition-//footnote//para	2
$TV1$	//dataset[//tableLink]//definition; //tableHead//title; //field//para	6
$TV2$	//dataset//tableHead;//field//para; //tableLink//title;//definition;	4
$TV3$	//dataset//definition//para; //tableHead//field; //tableLink//title	3
$TV4$	//field//definition//para; //dataset//tableHead; //tableLink//title	2

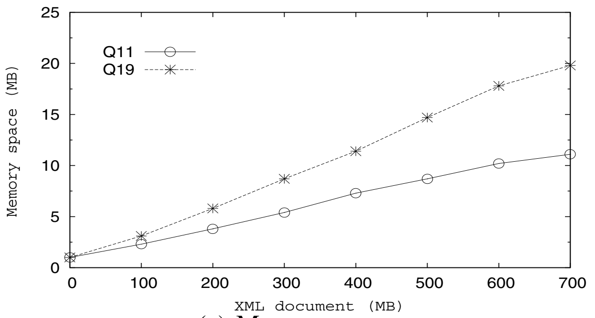
As shown in Fig. 6, the complexity of interleaving conditions has little impact on the performance of `TS` as it does not make use of precomputed join results in its computation. The impact of interleaving conditions is reflected on `IJ` (Fig. 6(a)) and `VJ` (with exception for `VJ+E`), as both algorithms exploit the precomputed join results in the views. Fewer interleaving conditions imply larger precomputed structural join results available for re-using in both algorithms. As the number of interleaving conditions decreases, the efficiency of `IJ`, `VJ+LE` and $\texttt{VJ+LE}_\text{p}$ increases.

C. Space Usage

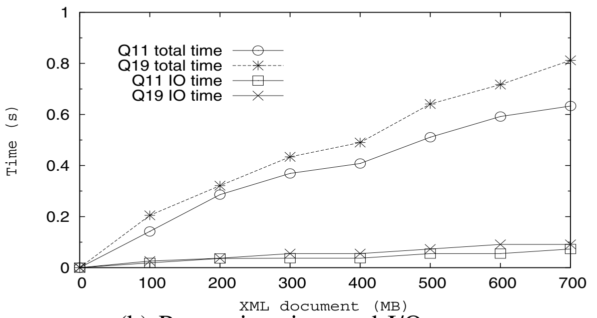
As observed by some prior work [6], the data nodes involved in a query result is a small portion of the XML documents. Table IV shows the size and the number of materialized pointers of two views on the XMark dataset of 700MB. These views are the most frequently occurring patterns in the XMark datasets. However, the views are rather small regardless of the storage scheme. The result indicates that views in the `E` scheme have the smallest size. There is no clear winner in terms of space usage between $\texttt{LE}/\texttt{LE}_\text{p}$ scheme and `T` scheme. For v_1, the materialized view in `T` scheme is larger than the size of $\texttt{LE}/\texttt{LE}_\text{p}$ scheme; while for v_2, the materialized view in $\texttt{LE}/\texttt{LE}_\text{p}$ scheme is slightly larger than the size of `T` scheme. Note that in the XMark dataset, a data node may occur in multiple matches of v_1, but not for v_2. Views in the $\texttt{LE}_\text{p}$ scheme are smaller than the `LE` scheme due to a smaller number of materialized pointers. The size of views is important in data storage and cost of the query evaluation. The above result implies that both the storage and the I/O cost is rather small.

D. Scalability

To test the scalability of the `VJ` algorithm, we processed the benchmark queries over the XMark datasets with increasing size from 100MB to 700MB. The queries and views are fairly complex, having up to 10 and 5 nodes respectively. The `VJ`



(a) Memory space



(b) Processing time and I/O cost

Fig. 7. Scalability of ViewJoin

TABLE IV
SIZE AND #POINTERS OF VIEWS ON XMARK, v_1 = //ITEM//TEXT//KEYWORD, v_2 = //PERSON//EDUCATION

	Size of view (MB)				**#pointers**	
	E	T	LE	LE$_p$	LE	LE$_p$
v_1	6.87	8.77	7.89	7.38	508276	252583
v_2	0.92	0.92	1.08	1.00	78614	39307

algorithm on the tested queries on XMark datasets has a rather small memory space requirement. Fig. 7 shows the memory space required and the processing time of VJ+LE evaluating benchmark queries $Q11$ and $Q19$ on the XMark datasets of 100MB to 700MB. The memory space and processing time follows a linear trend as document size increases. For the largest XMark document, the memory space needed is less than 20MB. Fig. 7(b) also shows the I/O time, which is less than 15% of the total processing time.

E. The Disk-based Approach

So far we have evaluated the performances of TS and VJ with regard to the memory-based approach. In this section, we examine the performance of the disk-based approach of both TS and VJ.

We used the same twig queries as in Section VI-A to evaluate the performance of TS and VJ in this approach. Table V shows the total processing time in milliseconds (I/O time is within the parentheses) for TS+E and VJ+LE. Similar experimental results are observed for other view storage schemes. For a clear comparison of the memory-based approach and the disk-based approach, we show the results of both approaches in Table V: TS-M and VJ-M indicate the results of the memory-based approach whereas TS-D and VJ-D indicate the results of the disk-based approach.

The results indicate that VJ-D outperforms TS-D by a factor up to 4.9. Compared to the memory-based approach, the disk-based approach of both algorithms takes a longer processing time, which is mainly caused by the increase in the I/O time. For both algorithms, a higher percentage of I/O time over total processing time is observed.

VII. Related work

There has been considerable research on efficient tree-pattern query evaluation focusing on structural joins [1] and holistic twig joins [4], [6]. There are also several work (e.g., [7], [8], [11], [15], [16], [28]) that studied how indexes can be used to optimize the join evaluation.

A containment forest structure was proposed [8] to organize the entries in the leaf nodes of a B+-tree index for XML data. It was designed to capture the containment relationships among nodes of the same type. A containment forest is a connected graph consisting of all the nodes of the same type. Each node in the containment forest is linked to some other nodes in the same containment forest via pointers (e.g., *first-child* and *right-sibling*). Our proposed DAG structure for representing materialized views is similar to but more general than the idea of containment forests, as our scheme organizes elements of mixed types using additional $child$ pointers.

There are native storage schemes ([10], [14]) proposed, which store the XML data by partitioning the XML document into disk pages. However, such native storage schemes are not optimal for materialized views due to the following inadequacies. (1) Unlike our LE scheme which supports skipping certain portion of materialized views in answering a given query Q by evaluating the view-segmented query Q' to avoid the scanning of the entire lists for the element types not in Q' (refer to the advantage 3 of LE scheme illustrated in Section III-B), native schemes may not support such advantage due to the lacking of $child$ pointers. (2) Native schemes do not support the skipping of reading elements in a materialized view which are not needed for evaluating a TPQ. If a node q in a query Q appears in more than one view, there are then multiple copies of q-type nodes in different views. With native schemes, we may have to scan through multiple copies of q-type nodes, as each view containing q-type nodes may contain relevant data of other types. With LE scheme which stores a view as individual lists by element types, we can easily avoid accessing multiple lists of q-type.

Query answering using views (QAV) [12] is a well-studied problem in relational databases. Recently in the context of XML, QAV was studied in [2], [3], [9], [17], [19], [21], [25], [26], [27], which focused predominantly on two *logical*

TABLE V

EXPERIMENTAL RESULTS: PROCESSING TIME (IN MILLISECONDS) OF THE TWO APPROACHES OF OUTPUTTING SOLUTIONS

	Q4	Q8	Q9	Q10	Q11	Q13	Q14	Q19	N5	N6	N7	N8
TS-M	1038(54)	893(17)	578(27)	927(20)	684(12)	834(22)	1011(26)	812(15)	684(12)	114(5)	577(15)	236(2)
TS-D	1391(267)	1249(203)	780(187)	1016(109)	906(125)	890(78)	1204(188)	1187(235)	891(109)	156(16)	829(78)	258(8)
VJ-M	483(92)	232(30)	200(44)	218(31)	161(21)	152(38)	372(44)	232(29)	141(16)	37(6)	119(19)	34(3)
VJ-D	528(139)	313(63)	248(92)	250(79)	259(78)	181(63)	457(111)	309(101)	218(78)	48(16)	235(64)	55(8)

optimization issues: view selection and query rewriting. Such logical optimization work employs a query model where each query has exactly one output node. This is different from our query model where each query node is an output node. Thus the techniques in such work are not applicable to our work.

The most relevant work to ours is the $InterJoin$ method [22] proposed to evaluate a path query using a pair of interleaving path views, for instance, joining views *//a//c* and *//b* to evaluate query *//a//b//c*. The view *//a//c* is stored as a list of (a, c)-tuples sorted by the composite key $(a.start, c.start)$. The proposed solution is an extension of the structural join algorithm in [1]. It scans through the (a, c)-list and b-list to compute (a, b, c)-tuples by joining a and b; and verifies each (a, b, c)-tuple as a query match by checking whether the b node is an ancestor of the c node.

However, the class of queries and views handled by InterJoin is rather limited. First, it only explores path views; twig views remain completely unexplored. Second, when more than two path views are needed to answer a path query, InterJoin is evaluated as a sequence of binary joins. As this is not a holistic approach, potentially large amount of useless intermediate results are generated from InterJoins. Our ViewJoin algorithm is a more general approach, which can handle both the path queries/views as well as twig queries/views. Our experimental results show that our proposed solution outperforms InterJoin by a factor up to 4.6.

VIII. CONCLUSION

The existing work on XML query processing with views has focused predominantly on *logical optimization* issues. In this paper, we examined the important *physical optimization* issue of how to efficiently organize materialized views and evaluate TPQs using them. We proposed novel storage schemes (LE and LE_p) for materialized TPQ views and a new TPQ evaluation algorithm using materialized views (VJ). Our experimental results demonstrated that the combination of VJ and LE_p significantly outperformed the state-of-the-art approaches by a factor up to 5.8.

REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structual joins: a primitive for efficient XML query pattern matching. In *ICDE*, 2002.

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

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

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

[5] D. Chen and C.-Y. Chan. ViewJoin: Efficient view-based evaluation of tree pattern queries. http://www.comp.nus.edu.sg/~chending/viewjoin/.

[6] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *VLDB*, 2006.

[7] T. Chen, J. Lu, and T. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *SIGMOD*, 2005.

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

[9] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. In *ICDE*, 2007.

[10] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4), 2002.

[11] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *CIKM*, 2005.

[12] A. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4), 2001.

[13] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[14] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB Journal*, 11(4), 2002.

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

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

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

[18] Q. Li and B. Moon. Indexing and querying XML data for regular expressions. In *VLDB*, 2001.

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

[20] G. Miklau. The XML data repository. 2002. http://www.cs.washington.edu/research/xmldatasets/.

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

[22] D. Phillips, N. Zhang, I. F. Ilyas, and M. T. Ozsu. INTERJOIN: exploiting indexes and materialized views in XPath evaluation. In *SSDBM*, 2006.

[23] A. Schmidt, F. Waas, M. Kersten, D. Florescu, and I. Manolescu. The XML benchmark project. Technical Report INS-R0103, CWI, Netherlands, 2001.

[24] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, 2005.

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

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

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

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