Incremental Maintenance of Path-Expression Views

Arsany Sawires[1] *
arsany@cs.ucsb.edu

Junichi Tatemura[2]
tatemura@sv.nec-labs.com

Oliver Po[2]
oliver@sv.nec-labs.com

Divyakant Agrawal[2]
agrawal@sv.nec-labs.com

K. Selçuk Candan[2]
candan@sv.nec-labs.com

[1]Department of Computer Science
University of California Santa Barbara
Santa Barbara, CA 93106

[2]NEC Laboratories America
10080 North Wolfe Road, Suite SW3-350
Cupertino, CA 95014

ABSTRACT

Caching data by maintaining materialized views typically requires updating the cache appropriately to reflect dynamic source updates. Extensive research has addressed the problem of incremental view maintenance for relational data but only few works have addressed it for semi-structured data. In this paper we address the problem of incremental maintenance of views defined over XML documents using path-expressions. The approach described in this paper has the following main features that distinguish it from the previous works: (1) The view specification language is powerful and standardized enough to be used in realistic applications. (2) The size of the auxiliary data maintained with the views depends on the expression size and the answer size regardless of the source data size.(3) No source schema is assumed to exist; the source data can be any general well-formed XML document. Experimental evaluation is conducted to assess the performance benefits of the proposed approach.

Keywords

Caching, Incremental View Maintenance, XML Views, Path Expressions

1. INTRODUCTION

Caching data by maintaining materialized views has many well-known benefits. One of the main benefits is improving query performance by answering queries from the cache instead of querying the source data. To be useful, a materialized view needs to be continuously maintained to reflect dynamic source updates. It has been shown that incremental maintenance generally outperforms full view recomputation. The problem of efficient incremental view maintenance has been addressed extensively in the context of relational data models [4, 8, 9, 10] but only few works have addressed it in the context of semi-structured data models [2, 7, 11, 13, 18].

The XML semi-structured data model has become the choice both in data and document management systems [5] because of its capability of representing irregular data while keeping the data structure as much as it exists. Thus, XML has become the data model of many of the state-of-the-art technologies and applications. In general scenarios, XML source data can be dynamically updated; this requires updating any cached views to reflect the source updates. In this paper, we present a novel technique for maintaining XML views by incorporating the source updates in the materialized views incrementally.

Currently, XML caching applications follow a simple time based invalidation scheme in which the cached view results are invalidated after a pre-specified time period (lifetime). The drawbacks of such a scheme are: (1) the cached results are likely to be over-invalidated since the invalidation process does not take into account the relevance of the source updates to the cached results, (2) the invalidation operation implies recomputing the views whenever they are required again; this re-computation process is generally an expensive one, and (3) the "freshness" of the cached results is not guaranteed because source updates may take place just after a result has been cached; the effect of such updates will not be reflected in the cache before the lifetime of the cache expires. This might be unacceptable for critical applications which require a high level of consistency between the source and the cache. The approach proposed in this paper incrementally maintains the cache by analyzing the source updates and updating the cache based on the relevance of these updates to the cached results.

The XML views maintained at the cache are assumed to be the results of certain queries (view specifications) issued against a source XML document. The W3C consortium [1] is currently working towards standardizing XPath and XQuery as XML query and view specification languages. Path expressions form the core of the XPath and XQuery languages; they are the language constructs which are used to select and

*The work has been done during the author's summer internship at NEC.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

retrieve data from XML data sources. The retrieved data is then manipulated by other language constructs to form the final XML query result. Therefore, caching the results of path expressions could be potentially beneficial to answer general XML queries efficiently. Recent research [3] has addressed the problem of using materialized path expression views to answer XML queries. Thus, we have chosen the view specification language to be the language of path expressions. We discuss how to efficiently and incrementally maintain the results of such expressions to reflect dynamic source updates.

The main features of the presented approach that distinguish it from the previous related works are as follows: (1) The view specification language, which is the language of path expressions, is powerful and standardized enough for a large class of real life applications. (2) The size of the auxiliary data maintained beside the view result depends only on the expression size (the number of steps) and the actual result size. (3) No source schema is assumed to exist; the source data can be any general well-formed XML document.

Generally, in order to maintain cached views, a maintenance algorithm needs to issue queries to the data source. Querying the source is generally an expensive operation in terms of time and processing since the data source is usually huge in size. The main theme of our approach is to minimize the number and the size of the source queries which are used to maintain the cached results.

The rest of this paper is organized as follows. In the next section, we summarize related work in the area of XML view maintenance. Section 3 presents the XML data and update model as well as the view specification model for specifying XML views. Section 4 presents the incremental view maintenance algorithm for XML path expressions. In Section 5, we report the results of a performance study we conducted using the XMark benchmark. We conclude with a discussion of our results in Section 6.

2. RELATED WORK

Some previous research has addressed the problem of incremental maintenance of views in the context of semistructured data models. Different data models and view specification languages have been assumed by different researchers. In this section we briefly describe and comment on these works.

One of the earliest papers that has addressed the problem is [18]. This work has identified some of the main challenges of the problem and proposed a solution assuming a general tree data model. However, the view specification language assumed in this early work is very limited. For example, the selection path expressions are not allowed to have wild cards, and the predicates are very simple conditions which are restricted only to the last step of path expressions.

The work in [2] addresses the problem assuming a graph data model. The main idea of the approach presented in this work is to incrementally maintain the views by issuing efficient source queries. The efficiency of the queries is quantified relative to the efficiency of the original view specification query based on the number of nodes processed during the query processing. Theoretical analysis reveals that the proposed incremental maintenance approach outperforms the approach of recomputing the view in many cases. However, the view specification language of this work is also limited; for example, the selection conditions are not allowed to include negations. Excluding some language features, such as negation, makes the view maintenance operation easier because it guarantees that the views are *monotonic* in the sense that addition updates in the source document can not cause deletions in the view result. To achieve more efficiency, the approach suggests using auxiliary data to detect the relevance of the updates to the cached views. However, the size of the proposed auxiliary data can grow arbitrarily regardless of the size of the cached view results.

The work in [11] elegantly applies the property of *multilinearity* or *distributivity* to implement incremental view maintenance. This property applies for a view $\mathcal{V}$ if and only if for every data source $\mathcal{D}$ and for every source update $\mathcal{U}$, the following holds: $\mathcal{V}(\mathcal{D} \uplus \mathcal{U}) = V(\mathcal{D}) \uplus V(\mathcal{U})$. This property is desirable since it enables the incremental maintenance of the view $\mathcal{V}$ using the update $\mathcal{U}$ without querying the data source. In this work, the operator $\uplus$ is defined as the *deep tree union*. Although applying this property sounds very promising to achieve very efficient view maintenance, there are some problems with this approach: (1) this property applies only for monotonic views, i.e. views for which a source addition can not cause view deletions and a source deletion can not cause view additions, this limits the power of the view specification language, and (2) as shown in the same work, to apply this property, some views would have to be rewritten by duplicating the same data source in the view parameter list. Further analysis of this rewriting issue reveals that the maintenance process in this case must issue source queries anyway, which cancels the main advantage of applying the *multilinearity* property.

An approach based on using auxiliary data is introduced in [13], another more general approach is introduced in [7]. The latter approach has a special strength in that it is based on a formal XQuery algebra, which enables it to support a very powerful query language. The cost, however, is storing all the intermediate results of the query processing as auxiliary data. The size of such auxiliary data is not bounded, it can be arbitrarily large regardless of the size of the cached view results.

Other approaches [15, 16] address the problem based on an XML schema or a relational schema which they use to determine the appropriate view maintenance actions.

3. DATA AND VIEW MODEL

In this section we present a formal model for XML data and for the language of path expressions which will be used for specifying XML views. In the following we use the notation $\mathcal{S} = (a, b, c, d)$ to denote an ordered sequence $\mathcal{S}$. In any sequence of XML nodes, the order of the nodes corresponds to the pre-order traversal of the source XML tree. We use the following operators on sequences: *subseqence*: $\sqsubset$, *member-of*: $\in$, *intersection*: $\sqcap$, *union*: $\sqcup$, and *difference*: $-$.

3.1 XML Data Model

3.1.1 XML Nodes

An XML document is represented as an ordered tree in which every node n is a pair $\langle n.id, n.label \rangle$ where $n.id$ is a node identifier that uniquely identifies the node among all the nodes in the XML tree. Several approaches to assigning node identifiers have been proposed in literature; recent approaches [6, 12, 17] guarantee the following properties:

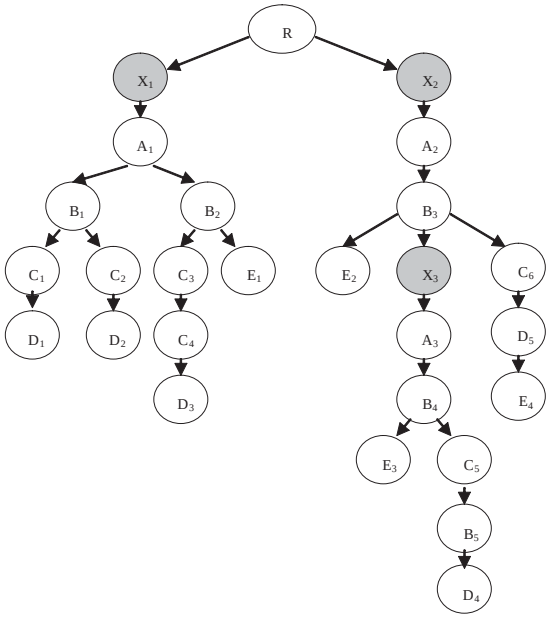


Figure 1: An Example XML Tree

1. Dynamic; i.e. adding and deleting nodes in the source tree do not require reassignment of node identifiers. This property is essential for caching applications because it preserves the source node identities.

2. Reflecting the document order; i.e. given the identifiers of any two nodes n_i and n_j, it can be determined if n_i is before or after n_j in the pre-order traversal of the source tree. This property is required to keep the order of nodes in the cached view in correspondence with the original document order of nodes.

3. Reflecting the containment relationships among the nodes; i.e. given the identifiers of two nodes n_i and n_j, it can be determined if n_i and n_j have ancestor or descendant relationship. This property is widely used by XML query processors.

The node label $n.label$ is a string that describes the node type and the node name or value. In particular:

- if n corresponds to an XML element then *label* represents the element name;
- if n corresponds to an XML attribute then *label* represents the attribute name and value;
- if n corresponds to a value of any data type then *label* is the value representation.

Based on the above definition of node labels, any selection condition in a query involving the node type, name, or value is represented as a label test. For example; a condition that retrieves "book" elements is a label test and a condition that retrieves nodes storing values greater than "5" is also a label test. A label test could also be the wildcard character "$*$" which matches all labels.

To maintain the brevity of the examples we will adhere to the following notation: we will use upper-case letters to represent the node labels. For example, A, B, and C are node labels. In contrast, we will not use node identifiers explicitly, instead, we will use numeric subscripts to distinguish different nodes that have the same label. Thus, A_i and A_j refer to two distinct nodes with the same label A. We will rely on pictorial illustrations of the examples to capture the ancestor and descendant relationships among the nodes. The tree order will be assumed to be from left to right in the pictorial illustrations. Figure 1 shows the XML source document that will be used as a running example throughout this paper.

3.1.2 XML Source Updates

A source update is a transformation of the source XML document. Although the transformation could be in the form of changes to the leaf nodes as well as internal nodes in the tree, we restrict ourselves to primitive transformations that operate at the level of the leaf nodes in an XML tree. It can be easily shown that any arbitrary transformation to the source tree, e.g. adding or deleting a sub-tree from the source, can be expressed in terms of the following two primitive operations: (1) *Add* a leaf node, and (2) *Delete* a leaf node.

More formally, an update $\mathcal{U}$ is a pair $\langle \mathcal{U}.type, \mathcal{U}.path \rangle$ where $\mathcal{U}.type$ is the type of the update: *Add* (add a leaf node) or *Delete* (delete a leaf node). $\mathcal{U}.path$ is the path of all the ancestors of the added or deleted node starting with the document root and ending with the added or deleted node itself. The added or deleted node is referred to as $\mathcal{U}.node$

For example, $\mathcal{U} = \langle Add, (R, X_1, A_1, B_1, Z) \rangle$ represents the addition of node Z as a child node of node B_1 in the XML document shown in Figure 1.

Note that every node in $\mathcal{U}.path$ is given by both its *label* and its *id*, this path information will be used by the maintenance algorithm presented in the next section.

3.2 The View Specification Language

3.2.1 Path Expressions

As mentioned in the introduction, path expressions are the basic building blocks of XML queries. In this paper we focus on a subset of the path expression language as defined in XPath 1.0 [1]. A path expression $\mathcal{E}$ of size N is a sequence of N steps: $(s_1, s_2, \cdots s_N)$. A step s_i is a triple $\langle s_i.axis, s_i.label, s_i.pred \rangle$ where:

- $s_i.axis$ is an axis test; it is either a child selector (denoted by "/") or a descendant selector (denoted by "//"). The axis test selects nodes based on the tree structure.
- $s_i.label$ is a label test; it selects some of the nodes that passed the axis test. The label test is evaluated by examining only the node label without examining any other nodes or structures in the tree.
- $s_i.pred$ is an optinal predicate test; it further filters the nodes that passed both the axis and the label tests. Unlike the label test, the predicate test can be any complex condition examining the labels and the structure of the nodes in the subtree of the node being tested. A predicate can use aggregate functions, user defined functions, operators, quantifiers ..., etc.

$$/\mathbf{A}//\mathbf{B}[\mathbf{Count}(//\mathbf{E}) \geq \mathbf{1} \vee \mathbf{Count}(/\mathbf{D}) \geq \mathbf{1}]//\mathbf{C}[\mathbf{Count}(//\mathbf{E}) = \mathbf{0}]//\mathbf{D}$$

Figure 2: An Example path-expression $\mathcal{E}$

Processing the first step s_1 starts at a pre-specified sequence of nodes in the source tree called the expression context $\mathcal{C}$. Given an expression $\mathcal{E}$, a document tree $\mathcal{D}$, and a sequence of context nodes $\mathcal{C}$ (a sequence of some of the nodes of $\mathcal{D}$), a query, $\mathcal{Q}$, denoted as

$$\mathcal{Q} = q(\mathcal{E}, \mathcal{C}, \mathcal{D})$$

returns a sequence of nodes $\mathcal{R}$ as a result. Conceptually, the execution of s_i $(i > 1)$ starts at the sequence outputted from executing s_{i-1}. Thus, we define the intermediate result of step s_i $(1 \leq i \leq N)$ as:

$$\mathcal{R}_i = q(s_i, \mathcal{R}_{i-1}, \mathcal{D}), \mathcal{R}_0 = \mathcal{C}$$

Every $\mathcal{R}_i, (1 \leq i \leq N)$ is a sequence of nodes ordered by the document order. The final result $\mathcal{R}$ is defined as the result of the last step; i.e. $\mathcal{R} = \mathcal{R}_N$.

For example, consider the query $\mathcal{Q} = q(\mathcal{E}, \mathcal{C}, \mathcal{D})$ where: $\mathcal{D}$ is the document tree shown in Figure 1, $\mathcal{C} = (X_1, X_2, X_3)$ (the shaded nodes in Figure 1), and the steps of $\mathcal{E}$ are specified in Figure 2 and the steps are enumerated below:

1. $s_1 = /A$
2. $s_2 = //B[Count(//E) \geq 1 \vee Count(/D) \geq 1]$
3. $s_3 = //C[Count(//E) = 0]$
4. $s_4 = //D$

In this query, the first step s_1 starts at every node in $\mathcal{C}$ and selects all the children with label A; this results in $\mathcal{R}_1 = (A_1, A_2, A_3)$. Then s_2 starts at every node in $\mathcal{R}_1$ and selects all the descendants with label B that have at least one descendant labeled E or at least one child labeled D; this results in $\mathcal{R}_2 = (B_2, B_3, B_4, B_4, B_5, B_5)$. Note that B_4 - and also B_5 - occurs twice in $\mathcal{R}_2$ because it can be derived in two ways from nodes of $\mathcal{R}_1$, one from A_2 and another one from A_3. This shows the possibilities of multi-derivations in path expression views. Starting at $\mathcal{R}_2$, step s_3 selects all the descendants labeled C that have no descendants labeled E; this results in $\mathcal{R}_3 = (C_3, C_4, C_5, C_5, C_5)$. Finally, s_4 starts at $\mathcal{R}_3$ and selects all the descendants labeled D. Hence, the final result of $\mathcal{Q}$ is $\mathcal{R} = \mathcal{R}_4 = (D_3, D_3, D_4, D_4, D_4)$. We differentiate between the multiple occurrences of the same node in a sequence by using a numeric superscript. For example, we denote the result $\mathcal{R}$ as $\mathcal{R} = (D_3^1, D_3^2, D_4^1, D_4^2, D_4^3)$

3.2.2 Definitions

The presentation of the incremental maintenance algorithm in the next section uses the following definitions regarding path expressions.

DEFINITION 1. *$Pred_i(n)$ is* true *if and only if (1) Node n belongs to the source tree, and (2) $s_i.pred$ evaluates to* true *at node n or s_i does not have a predicate test.*

For example, $Pred_3(C_1)$ in the example query above is *true* because C_1 satisfies the condition $s_3.pred$ since C_1 has no descendants labeled E.

DEFINITION 2. *The* Result Path *of a node n in the result $\mathcal{R}$, referred to as $ResultPath(n)$,is the sub-sequence (may be non-contiguous) of the ancestors of n (including n) that matched the steps of $\mathcal{E}$ and thus caused n to appear in $\mathcal{R}$.*

In the example query described above,

$$ResultPath(D_3^1) = (X_1, A_1, B_2, C_3, D_3), and$$

$$ResultPath(D_3^2) = (X_1, A_1, B_2, C_4, D_3)$$

This example shows that no two result paths are equal; even if a single node from the source tree occurs multiple times in $\mathcal{R}$, each occurrence is associated with a different result path. Note that all the result paths have the same size, which is equal to $N + 1$, where N is the expression size. This is because every element in a result path matches exactly one step of $\mathcal{E}$ and every step of $\mathcal{E}$ is matched with exactly one element in each result path; the extra 1 is because the first node in each path result is a context node from the sequence $\mathcal{C}$ which does not match any step.

DEFINITION 3. *For every node n such that $n \in \mathcal{R}$, we define $ResultPath_i(n)$, $i \geq 0$ as the i^{th} element in the result path of n.*

By this definition, $\forall n \in \mathcal{R}$

$$ResultPath_0(n) \in \mathcal{C},\ ResultPath_N(n) = n$$

3.2.3 Restrictions

Although we admit a large class of path expressions defined by XPath 1.0 [1], we also enforce some restrictions to achieve efficient view maintenance:

- We handle only *child* and *descendant* axes in the axis test. The other axis types, such as *parent* and *ancestor*, are not handled. However, this restriction should not compromise the pragmatic power of the view specification language because the *child* and *descendant* axes are the most commonly used axes in practice.
- A Predicate can examine only the subtree of the node being tested. In other words: $Pred_i(n), \forall i$ is exclusively evaluated by examining the subtree rooted at n. This assumption imposes a theoretical restriction on the predicates, but it is reasonably supported by the fact that a node in an XML document is semantically described by its descendants, hence, selecting a node should depend on its label and its descendants. Within this restriction, we allow arbitrarily complex predicates that can include aggregation functions, user defined functions, negation, universal/existential quantifiers ..., etc.

4. INCREMENTAL MAINTENANCE

In this section, we develop an algorithm for incrementally maintaining the cached result $\mathcal{R}$ of a query based on a path-expression $\mathcal{E}$ in the presence of a source update $\mathcal{U}$. We start with some preliminaries and motivating examples to describe the basic structure of the proposed algorithm. The following subsections present the details of the algorithm.

4.1 Preliminaries

Assume that the result $\mathcal{R}$ of the example expression $\mathcal{E}$ defined in Figure 2 is to be materialized, and subsequently assume that the following update takes place at the source tree of Figure 1: $\mathcal{U} = (Add, (R, X_1, A_1, B_1, E_5))$. The effect of this update is that it changes $Pred_2(B_1)$ from *false* to *true*. The direct effect of this change on the evaluation process of $\mathcal{E}$ is to add B_1 to the intermediate result $\mathcal{R}_2$. Now, since there is a new node added to $\mathcal{R}_2$, there is a possibility that this addition can induce other indirect additions in the subsequent intermediate results $\mathcal{R}_i$, $i > 2$. This is indeed the case in this scenario since nodes C_1 and C_2 would now qualify to be in $\mathcal{R}_3$ as descendants of B_1. Moreover, the inclusion of C_1 and C_2 causes D_1 and D_2 to be added to $\mathcal{R}_4$, i.e. to the materialized result $\mathcal{R}$.

This example illustrates that an update $\mathcal{U}$ can affect the final result $\mathcal{R}$ by impacting any of the intermediate results R_i. In this example, $\mathcal{U}$ changed $Pred_i(n)$ for only one node $(n = B_1)$ and one value of $i(i = 2)$. This change effectively added B_1 to R_2. Consequently, other nodes were added to other intermediate results but without $\mathcal{U}$ changing any more predicate values; these are nodes C_1, C_2, D_1, and D_2. Thus, an update $\mathcal{U}$ causes a node n to be added to an intermediate result $\mathcal{R}_i$ under one of two possible scenarios:

1. $\mathcal{U}$ changes $Pred_i(n)$ from *false* to *true*;
2. $\mathcal{U}$ does not affect $Pred_i(n)$.

We refer to the first case as a *direct addition* and to the second one as an *indirect addition* because it is caused indirectly through a direct addition.

Similarly, we use the term *direct deletion* when $\mathcal{U}$ changes $Pred_i(n)$ from *true* to *false* causing n to be deleted from $\mathcal{R}_i$. And we use the term *indirect deletion* when n is deleted from $\mathcal{R}_i$ without $\mathcal{U}$ affecting $Pred_i(n)$. For example, if $\mathcal{U} = (Add, (R, X_1, A_1, B_2, C_3, E_6))$ then $\mathcal{U}$ directly deletes C_3 from $\mathcal{R}_3$ because it changes $Pred_3(C_3)$ from *true* to *false*. This direct deletion induces the indirect deletion of the first occurrence of D_3 from $\mathcal{R}$.

In addition to illustrating the notions of direct and indirect deletions, this example shows that unlike other approaches [2, 11], we do not restrict view definitions to be monotonic. That is, we handle cases where an addition in the source could result in additions or deletions in the view. Similarly, we handle cases where a deletion in the source could result in additions or deletions in the view results. For example, deleting E_3 from the source tree directly deletes all the occurrences of B_4 from $\mathcal{R}_2$ while deleting E_4 directly adds C_6 to $\mathcal{R}_3$.

For brevity of the presentation, we use the following simple definitions: δ_i^+ is the sequence of all nodes that $\mathcal{U}$ directly adds to $\mathcal{R}_i$. δ_i^- is the sequence of all nodes that $\mathcal{U}$ directly deletes from $\mathcal{R}_i$, and $\delta_i = \delta_i^+ \sqcup \delta_i^-$.

Notice that each of δ_i^+ and δ_i^- could have repetition in it due to multiderivation possibilities. Also notice that δ_i^+ and δ_i^- are mutually disjoint because a node n can not be directly added to and deleted from $\mathcal{R}_i$ at the same time; that is because $\mathcal{U}$ can not change $Pred_i(n)$ from *false* to *true* and from *true* to *false* at the same time.

The notion of direct and indirect effects is intrinsic to our algorithm; the algorithm depends on the fact that every indirect addition originates from a direct addition and every indirect deletion originates from a direct deletion. Thus, the algorithm first discovers the direct effects and then uses them to discover the indirect ones. Let us assume, for now, that we have discovered all the direct additions and deletions at $\mathcal{R}_i$; now the problem is how to discover the indirect effects that are induced by the direct effects. Note that we ultimately want to discover the indirect effects on the cached result $\mathcal{R}$; the indirect effects on all the intermediate results $\mathcal{R}_i$, $i < N$ are not required per se, but they need to be identified in order to determine the final effects on $\mathcal{R}$. To discover indirect effects from the direct ones, we need to handle two cases:

1. Direct additions: when a node n is directly added to $\mathcal{R}_i$, then the maintenance algorithm has to issue a query to the source to determine the indirect additions that might happen due to this direct addition. For example, when B_1 is added to R_2, the indirectly added nodes C_1, C_2, D_1, and D_2 can not be retrieved without querying the source because they had no existence in the cached view before $\mathcal{U}$ occurred. In general, when a node n is directly added to $\mathcal{R}_i$ then, in order to retrieve the indirect additions at all $\mathcal{R}_j$, $j > i$, the maintenance algorithm needs to issue a source query with context as the singleton sequence (n) and with the steps sequence $(s_{i+1}, s_{i+2}, \cdots s_N)$. Following our formal notation, this query is denoted as: $q((s_{i+1}, s_{i+2}, \cdots, s_N), (n), \mathcal{D})$.
2. Direct deletions: when a node n is directly deleted from $\mathcal{R}_i$, then all the nodes of $\mathcal{R}$ that came to $\mathcal{R}$ because n used to belong to $\mathcal{R}_i$ must be deleted from $\mathcal{R}$. In other words, all the nodes r of $\mathcal{R}$ that have $ResultPath_i(r) = n$ must be deleted from $\mathcal{R}$. In the example, the direct deletion of C_3 from $\mathcal{R}_3$ results in deleting D_3^1 from $\mathcal{R}$ because $ResultPath_3(D_3^1) = C_3$.

The reasoning used above to discover indirect deletions shows that if we know the result path of each node of $\mathcal{R}$, then we can discover the necessary indirect deletions from $\mathcal{R}$ without issuing any source queries. Motivated by this argument, we actually keep with every node $n \in \mathcal{R}$ the result path $ResultPath(n)$. The collection of all the result paths is kept as auxiliary data which is not itself a target, but it is just used to achieve efficient incremental maintenance of the cached result $\mathcal{R}$. This is the only auxiliary data required by the algorithm.

Note that keeping all the result paths is not equivalent to keeping all the intermediate results $\mathcal{R}_i$s. In particular, if a node n in $\mathcal{R}_i$ does not lead to a node in $\mathcal{R}$ then we do not keep n in the auxiliary data. For example, in the case of the expression of Figure 2: $B_5 \in \mathcal{R}_2$. However, B_5 did not lead to any node in $\mathcal{R}$ because none of its descendants were qualified to be in $\mathcal{R}_3$ or $\mathcal{R}_4$. Thus, B_5 is not kept in the auxiliary data. Obviously, the number of such nodes like B_5 can be arbitrarily large in the source tree.

Therefore, unlike other approaches [2, 7, 13], by keeping only the result paths as the auxiliary data, we guarantee that the size of the auxiliary data is bounded regardless of the source tree. To compute this size, recall that each result path is of length $N + 1$; let M be the size of the cached result $\mathcal{R}$, then the size of the auxiliary data is $O(M * N)$. Note, also, that we need to store only the node *id*s in the result paths, the node *labels* are not needed. This limits the size of the auxiliary data because the node *id*s are usually machine-generated compact codes.

After we have described how to discover the indirect effects given the direct ones, we now turn back to the problem

of discovering the direct effects in the first place. Our approach solves this problem in two phases for every $\mathcal{R}_i$: the **Axis&Label Test** and the **Predicate Test**. The following two subsections describe these two phases.

4.2 The Axis&Label Test

For every $\mathcal{R}_i$, discovering the sequence of direct effects δ_i requires querying the source because it might involve predicate evaluations to determine the nodes n for which $Pred_i(n)$ has changed due to $\mathcal{U}$. Since we want to minimize the amount of source queries, we have developed this phase to identify a sequence Δ_i such that we guarantee, without any source queries, that $\delta_i \sqsubset \Delta_i$. In the next phase, Predicate Test, Δ_i is further filtered by predicate evaluations to identify the exact sequence δ_i. In other words, the Axis&Label Test works as a first-level filter for identifying δ_i.

The first observation on which this phase is based is that every node n in δ_i must be in $\mathcal{U}.path$. The following lemma asserts this observation.

LEMMA 1. $\sqcup_{i=1}^{N}\delta_i \sqsubset \mathcal{U}.path$. In other words: If, due to $\mathcal{U}$, a node n belongs to δ_i for any i, then n must also belong to $\mathcal{U}.path$.

Proof. Since n belongs to δ_i, then n is directly added to or deleted from $\mathcal{R}_i$. Since this is a direct effect, then $Pred_i(n)$ is changed due to $\mathcal{U}$. Since $Pred_i(n)$ is assumed to reference only nodes in the subtree rooted at n, then it cannot be changed unless this subtree is changed. Therefore, n must be an ancestor of the updated node $\mathcal{U}.node$. That is, n must belong to $\mathcal{U}.path$. □

Lemma 1 limits the search space to the nodes in $\mathcal{U}.path$. In addition to that, we observe that for a node n to be directly added to or deleted from $\mathcal{R}_i$ it must have an ancestor in every $\mathcal{R}_j$, $(j < i)$. For the example shown above where $\mathcal{U} = (Add, (R, X_1, A_1, B_1, E_5))$, for node B_1 to be directly added to $\mathcal{R}_2$, it is not enough that $\mathcal{U}$ changes $Pred_2(B_1)$ from *false* to *true* but it is also necessary that:

1. B_1 has an ancestor in $\mathcal{R}_1$; this is true because $A_1 \in \mathcal{R}_1$.
2. the ancestor A_1 has an ancestor in $\mathcal{R}_0$; this is also true because $X_1 \in \mathcal{R}_0$ (because $X_1 \in \mathcal{C}$)

A similar argument applies for the case of direct deletions.

The observation stated above shows that for every node n in δ_i, n must have an ancestor m in $\mathcal{R}_{i-1}$, and m must have an ancestor in $\mathcal{R}_{i-2}$, and so forth, until we reach an ancestor in $\mathcal{R}_0$, i.e. in the expression context $\mathcal{C}$. Note that all these ancestors are ancestors of n. Since Lemma 1 states that n itself belongs to $\mathcal{U}.path$, then all its ancestors also belong to $\mathcal{U}.path$. This suggests that $\mathcal{U}.path$ has much of the information needed to identify the nodes of δ_i.

In fact, $\mathcal{U}.path$ has all the information needed to conduct the axes and labels tests needed to identify δ_i. However, it does not have enough information to evaluate the predicates at any of its nodes n because we allow a predicate to refer to any node in the subtree of n; examining such a subtree requires querying the source tree which is the only place guaranteed to have the whole subtree of any arbitrary node. Thus, we apply the axes and label tests to $\mathcal{U}.path$ ignoring the predicate tests. As a result, we get the sequence Δ_i which is guaranteed to be a supersequence of δ_i as shown below.

Computing the different Δ_i's proceeds similar to computing the intermediate results $\mathcal{R}_i$'s of the original view specification query except that the latter selects from the source tree $\mathcal{D}$ while the former selects from the single branch $\mathcal{U}.path$. To start, we note that, as mentioned above, any node n in any δ_i must have a node of the expression context $\mathcal{C}$ as an ancestor. Thus, we initialize Δ_0 to be all the context nodes that exist in $\mathcal{U}.path$, i.e. $\Delta_0 = \mathcal{C} \sqcap \mathcal{U}.path$. After this initialization, we proceed by computing Δ_i $(\forall i > 1)$ as all the nodes in $\mathcal{U}.path$ that satisfy $s_i.axis$ and $s_i.label$ starting at nodes in Δ_{i-1}. We conveniently denote this query as $\Delta_i = q(s_i.axis\&label, \Delta_{i-1}, \mathcal{U}.path)$.

Computing the different Δ_i's in the way described above guarantees that $\delta_i \sqsubset \Delta_i$ for all $1 \leq i \leq N$ because this process uses a relaxed selection condition (ignoring the predicate tests) over a tree branch that is guaranteed to have all the nodes of all the δ_i's, namely, this branch is $\mathcal{U}.path$. The following example shows the computation of Δ_i's.

Example. Consider an update $\mathcal{U}$ of adding a node D_6 as a child of D_4. In this case, $\mathcal{U}.path$ is the tree branch that starts with the root R and ends with D_6. Computing the different Δ_i's as described above results in: $\Delta_0 = (X_2, X_3)$, $\Delta_1 = (A_2, A_3)$, $\Delta_2 = (B_3, B_4, , B_4, B_5, B_5)$, $\Delta_3 = (C_5, C_5, C_5)$, $\Delta_4 = (D_4, D_4, D_4, D_6, D_6, D_6)$.

Note that Δ_i is just a supersequence of δ_i; i.e. there are nodes in Δ_i that are not directly added to or deleted from $\mathcal{R}_i$. For the example shown above, using the predicates as defined in the example path expression of Figure 2, we see that the only nodes that will be directly added are the three occurrences of D_6 that appear in Δ_4; all the other nodes n in all the computed Δ_i's will not be added or deleted because $\mathcal{U}$ did not affect $Pred_i(n)$. Note that, because D_6 did not exist before $\mathcal{U}$ occurred, the value $Pred_i(D_6)$, $\forall i$ is *false* before $\mathcal{U}$. Similarly, if an update deletes a node n from the source tree, the value $Pred_i(n)$, $\forall i$ is *false* after $\mathcal{U}$.

4.3 The Predicate Test

The goal of this test is to identify, the sequence δ_i from the sequence Δ_i. To accomplish this task, we need to determine which nodes n in Δ_i had their $Pred_i(n)$ changed due to $\mathcal{U}$. Let us refer to the value of $Pred_i(n)$ before $\mathcal{U}$ occurred as $Pred_i^{before}(n)$ and to the value after $\mathcal{U}$ occurred as $Pred_i^{after}(n)$. To detect such changes we need to compare, for every node $n \in \Delta_i$, the values $Pred_i^{before}(n)$ and $Pred_i^{after}(n)$

Nodes for which $Pred_i^{after}(n) = Pred_i^{before}(n)$, are excluded because they are not directly affected by $\mathcal{U}$. Nodes that have their $Pred_i(n)$ changing due to $\mathcal{U}$ are directly added to or deleted from $\mathcal{R}_i$. Hence, the question that we need to answer now is: How to compute the values of $Pred_i^{after}(n)$ and $Pred_i^{before}(n)$ for every node n in Δ_i?

The value of $Pred_i^{after}(n)$ is computed simply by querying the source. This query has only one node n in its context, thus its processing is relatively fast; the answer is a single boolean value *true* or *false*. We denote this query as: $predq(s_i.pred, (n), \mathcal{D})$. We delegate this query to the source query processor. The benefit of delegating predicate evaluations to the source is that we do not need to keep any auxiliary data that might be needed to evaluate complex predicates. Since a predicate can be any complex condition referring to any node in the subtree of the node being tested, testing for predicates without source queries

would entail maintaining arbitrarily large amounts of auxiliary data with the cached view result. Furthermore, this approach preserves the privacy of the data source by enabling it to hide the source data that is needed to evaluate the predicates, and also to hide the predicate definitions themselves.

Unlike $Pred_i^{after}(n)$, the value of $Pred_i^{before}(n)$ cannot be computed by a source query because the update $\mathcal{U}$ has already been incorporated at the source. We deduce the value of $Pred_i^{before}(n)$ as follows: if node n appears as the i^{th} element in the result path of any node in $\mathcal{R}$ then this implies that n was qualified for $\mathcal{R}_i$ before $\mathcal{U}$ occurred; hence, $Pred_i^{before}(n)$=*true*. Let us define $RP_i(n)$ to be *true* if and only if n is the i^{th} element of the result path of some node in $\mathcal{R}$. Hence, the argument mentioned above can be formally stated as $RP_i(n) \Rightarrow Pred_i^{before}(n)$. This shows how the auxiliary data - which was originally intended to be used for discovering indirect deletions - could help in the predicate test as well.

However, if $RP_i(n)$ is *false* then the value of $Pred_i^{before}(n)$ cannot be determined because it may be *false* or *true*. It is obvious how it could be *false*: simply, if $Pred_i^{before}(n)$ is *false* then n could not have qualified to be in $\mathcal{R}_i$ before $\mathcal{U}$ occurred, and hence, $RP_i(n)$ must be *false*. To see how $Pred_i^{before}(n)$ could be *true* while $RP_i(n)$ is *false*, assume an update $\mathcal{U}$ occurred after the original evaluation of the expression $\mathcal{E}$ in Figure 2. Before $\mathcal{U}$ occurred, the evaluation reveals that node B_5 was qualified to be in $\mathcal{R}_2$, i.e. $Pred_2^{before}(B_5)$ was *true*. However, B_5 did not lead to any node in $\mathcal{R}$ because none of its descendants was qualified to be in $\mathcal{R}_3$ or $\mathcal{R}_4$; and hence, $RP_2(B_5)$ is *false*. Note that one possible solution to this situation is to include in the auxiliary data all the nodes that qualify to be in any intermediate result $\mathcal{R}_i$ instead of only including those nodes that actually lead to nodes in the final result $\mathcal{R}$. However, we do not adopt this solution because it implies unbounded amount of auxiliary data.

Thus, if $RP_i(n)$ is false, there is ambiguity about the value of $Pred_i^{before}(n)$. We solve this ambiguity by simply assuming the worst case, i.e., we assume that $Pred_i^{before}(n)$ is *false*. Now we show that this assumption, if wrong, does not compromise the correctness of the algorithm.

LEMMA 2. if $Pred_i^{before}(n) = true$ and $RP_i(n) = false$ then assuming that $Pred_i^{before}(n) = false$ does not affect the result of discovering the indirect effects in $\mathcal{R}$.

Proof. We need to prove the correctness of this statement in two cases: $Pred_i^{after}(n) = true$ *and* $Pred_i^{after}(n) = false$.

1. $Pred_i^{after}(n) = true$*: in this case, node n would be falsely considered as a direct addition to $\mathcal{R}_i$ because we are falsely assuming that* $Pred_i^{before}(n) = false$*. To retrieve the indirect additions due to this false direct addition, the maintenance algorithm would issue the following query:*

$$q((s_{i+1}, s_{i+2}, \cdots, s_N), (n), \mathcal{D})$$

The answer sequence of this query is either empty or non-empty; we will show that, in either case, no false indirect additions can take place in $\mathcal{R}$.

   - *If the answer is empty then nothing will be added to the materialized view result and the algorithm is trivially true.*
   - *If the answer is not empty then this implies that $\mathcal{U}$ directly added some descendants m of n to some $\mathcal{R}_j$, $(j > i)$ such that this effect led to adding one or more nodes r in the final result $\mathcal{R}$. These nodes r form the non-empty answer of the maintenance query mentioned above. We know that none of these nodes r was in $\mathcal{R}$ before $\mathcal{U}$ because $RP_i(n) = false$, meaning that n did not lead to any node in $\mathcal{R}$ before $\mathcal{U}$ happened. Considering the directly added descendants m at $\mathcal{R}_j$, they would be discovered and processed when computing δ_j, this would happen later in the maintenance process $(j > i)$. Thus, all what happened because of the false assumption is that those directly added descendants m were discovered and processed earlier in the process $(i < j)$. In stage j, the maintenance algorithm can easily avoid duplicating this processing* [1].

2. $Pred_i^{after}(n) = false$*: in this case n should be directly deleted from $\mathcal{R}_i$. But this direct deletion would go unnoticed because we are falsely assuming* $Pred_i^{before}(n)$ *to be $false$. However, we guarantee that there would be no indirect deletions in $\mathcal{R}$ due to this unnoticed direct deletion. The reason is that $RP_i(n)$ is $false$, this means that node n is not the i^{th} element in the result path of any node in $\mathcal{R}$. Hence, no node in $\mathcal{R}$ depends on the existence of n in $\mathcal{R}_i$. Hence, ignoring this direct deletion of n does not compromise the correctness of discovering indirect deletions.*

□

Next, we present the complete algorithm for view maintenance of XML path expressions based on the basic ideas described so far.

4.4 The Maintenance Algorithm

The maintenance algorithm, presented in this section, combines the two phases described above to discover the direct effects at every $\mathcal{R}_i$ and uses the discovered direct effects to discover the ultimate effects on the cached result $\mathcal{R}$. The presentation in the previous two subsections suggests the following straightforward algorithm:

- Initialize: $\Delta_0 = \mathcal{C} \cap \mathcal{U}.path$
- FOR ($i = 1$; $i \leq N$ AND Δ_{i-1} is not empty; $i++$)
  - Compute Δ_i by applying the Axis&Label test of s_i starting at nodes of Δ_{i-1}
  - Compute δ_i by applying the Predicate test of s_i to nodes of Δ_i
  - Use δ_i to find all the indirect effects on $\mathcal{R}$
- Update $\mathcal{R}$ by incorporating the discovered additions and deletions

In the first step of the loop, every Δ_i is computed from Δ_{i-1}. Or, in other words, every Δ_{i+1} is computed from Δ_i. However, it is possible to improve the algorithm performance

[1]The algorithm in figure 3 avoids such duplication by using lemma 3

by excluding some nodes from Δ_i before moving on to the computation of Δ_{i+1} in the next loop iteration. This will result in a smaller Δ_i and hence in improved performance. We refer to the sequence that we get by reducing Δ_i as Λ_i. The idea is to show that, in order to discover all the ultimate effects on $\mathcal{R}$, it is sufficient to start every iteration $i+1$ only at the nodes n of the previous iteration (i) for which $RP_i(n) = Pred_i^{after}(n) = true$. The following lemma asserts this observation.

LEMMA 3. Given the following definition of Λ_is:

- $\Lambda_0 = \Delta_0 = \mathcal{C} \cap \mathcal{U}.path$.
- $\Lambda_1 = \Delta_1$
- for $i \geq 1$: $\Lambda_{i+1} = q(s_{i+1}.axis\&label, \mathcal{X}_i, \mathcal{U}.path)$ where $\mathcal{X}_i$ is a reduction of Λ_i by applying the following computation: $\mathcal{X}_i = (n | n \in \Lambda_i \wedge Pred_i^{after}(n) = true \wedge RP_i(n) = true)$.

Then, using the Λ_is instead of the Δ_is will discover all the ultimate effects of $\mathcal{U}$ on $\mathcal{R}$.

Proof. To prove this lemma, we need to show that no additions or deletions of nodes in $\mathcal{R}$ would go undiscovered if, before every iteration $i+1$, we exclude from Λ_i the nodes n which do not satisfy the condition $RP_i(n) = Pred_i^{after}(n) = true$. Logically, there are three cases of not satisfying this condition:

1. *$RP_i(n) = false$ and $Pred_i^{after}(n) = false$: in this case, the existence of n in Λ_i can not lead to an addition in $\mathcal{R}$ because $Pred_i^{after}(n) = false$. Also, it can not lead to a deletion from $\mathcal{R}$ because there is no node r in $\mathcal{R}$ such that $ResultPath_i(r) = n$, we guarantee that because $RP_i(n) = false$.*

2. *$RP_i(n) = true$ and $Pred_i^{after}(n) = false$: in this case also, the existence of n in Λ_i can not lead to an addition in $\mathcal{R}$ because $Pred_i^{after}(n) = false$. We can also show that excluding n from Λ_i after iteration i and before iteration $i+1$ can not cause a deletion in $\mathcal{R}$ to be undiscovered: recall that $RP_i(n) \Rightarrow Pred_i^{before}(n)$, therefore, the algorithm must have considered $Pred_i^{before}(n)$ to be true, and thus processed node n as a direct deletion at iteration i. Hence, the deletions of all the nodes of $\mathcal{R}$ that depend on the existence of n in $\mathcal{R}_i$ have already been discovered in iteration i, and there is no need to discover them again in any later iterations.*

3. *$RP_i(n) = false$ and $Pred_i^{after}(n) = true$: in this case, the existence of n in Λ_i can not lead to a deletion from $\mathcal{R}$ because $Pred_i^{after}(n) = true$. We can also show that excluding n from Λ_i after iteration i and before iteration $i+1$ can not cause an addition in $\mathcal{R}$ to be undiscovered: because $RP_i(n) = false$, at iteration i, the algorithm has assumed that $Pred_i^{before}(n) = false$ (see lemma 2). Therefore, the algorithm must have already processed the direct addition of n at iteration i, hence, all the consequent indirect additions in $\mathcal{R}$ have already been discovered and processed in iteration i, and there is no need to discover them again in any later iterations.*

Hence, in all the three cases, excluding node n from Λ_i before proceeding to compute Λ_{i+1} can not compromise the correctness of the algorithm because node n either has no effect on the subsequent iterations or its effects have already been discovered during the processing of Λ_i at iteration i. □

Figure 3 presents the final incremental view maintenance algorithm. Based on Lemma 3, this algorithm computes and uses the reduced sequences Λ_is instead of the Δ_is. We refer to the sequences of nodes which will be added to/deleted from $\mathcal{R}$ due to $\mathcal{U}$ as $\mathcal{R}^+/\mathcal{R}^-$ respectively. Step 1 of the algorithm initializes Λ_0 and the variables $\mathcal{R}^+$ and $\mathcal{R}^-$.

The loop at step 2 iterates to compute the effects of $\mathcal{U}$ on each of the intermediate results $\mathcal{R}_i$. The first step in the loop, step 2-1, generates Λ_i from Λ_{i-1}. Step 2-2 evaluates $Pred_i^{after}(n)$ for every node n in Λ_i. According to the results of these queries, Λ_i is partitioned into two disjoint sequences: $\mathcal{T}$ and $\mathcal{F}$. Then, step 2-3 uses $\mathcal{T}$ and $\mathcal{F}$ to identify the direct additions and deletions at $\mathcal{R}_i$, i.e. δ_i^+ and δ_i^- respectively.

Step 2-4 adds to $\mathcal{R}^+$ the nodes that will be added to $\mathcal{R}$ due to the direct additions discovered at the current iteration. Similarly, Step 2-5 adds to $\mathcal{R}^-$ the nodes that will be deleted from $\mathcal{R}$ due to the direct deletions discovered at the current iteration. Conforming to the process of discovering indirect effects as shown in section 4.1, step 2-4 issues a source query while step 2-5 only uses the auxiliary data. Instead of issuing a separate source query for every direct addition, step 2-4 uses a single query with a combined context sequence which incorporates all the direct additions at one shot, this should perform better than issuing many queries. At the end of each iteration, step 2-6 computes the reduced Λ_i to be used directly by step 2-1 of the next iteration. Finally, after all the iterations are executed, step 3 updates $\mathcal{R}$ by incorporating the nodes of $\mathcal{R}^+$ and $\mathcal{R}^-$.

Note that the maintenance algorithm needs to maintain the auxiliary data as well as the cached result $\mathcal{R}$. For every node n removed from $\mathcal{R}$, $ResultPath(n)$ is removed from the auxiliary data; and for every node n added to $\mathcal{R}$, $ResultPath(n)$ must be added to the auxiliary data. Note that computing the result paths requires some cooperation from the source query processor: the query processor should return with every node n in the answer of the query in step 2-4 its result path $ResultPath'(n)$. This result path is a partial path of length $N - i < N$ because the query in step 2-4 uses only steps $s_{i+1}, s_{i+2}, \cdots, s_N$ of the original expression. Thus, to get the full result path $ResultPath(n)$, we concatenate $ResultPath'(n)$ to the right end of a second result path of length i. This second path is the one which led from a node in the original expression context $\mathcal{C}$ to the first node in $ResultPath'(n)$; it can be found by tracing the sequences $\Lambda_0, \Lambda_1, \cdots \Lambda_i$ through the iterations $1, 2, \ldots, i$. For simplicity, this secondary process of maintaining the auxiliary data is not included in the algorithm in Figure 3.

A general look at the algorithm reveals that it issues several source queries; however, the processing of these queries is much less expensive than the alternative of issuing the original view specification query. The reason is that these queries are much smaller regarding their sizes and contexts than the original view specification query. This advantage of incremental maintenance over full re-computation is asserted by the experimental results shown in the following section.

// Initialization
1- $\Lambda_0 = \mathcal{C} \cap \mathcal{U}.path$
$\mathcal{R}^+ = \mathcal{R}^- = ()$ //Empty sequences
// Loop to compute Λ_i and δ_i, $\forall i \geq 1$
2- FOR ($i = 1$; $i \leq N$ AND Λ_{i-1} is not empty; $i++$)
{
// Generate Λ_i from Λ_{i-1}
2-1 $\Lambda_i = q(s_i.axis\&label, \Lambda_{i-1}, \mathcal{U}.path)$
// Evaluate $Pred_i^{after}(n)$ for every node n in Λ_i
2-2 $\forall n \in \Lambda_i$, compute $Pred_i^{after}(n) = predq(s_i.pred, (n), \mathcal{D})$
Let $\mathcal{T} = (n | n \in \Lambda_i \wedge Pred_i^{after}(n) = true)$
Let $\mathcal{F} = (n | n \in \Lambda_i \wedge Pred_i^{after}(n) = false)$
// Identify δ_i^+ and δ_i^-.
2-3 $\delta_i^+ = (n | n \in \mathcal{T} \wedge RP_i(n) = false)$
$\delta_i^- = (n | n \in \mathcal{F} \wedge RP_i(n) = true)$
// Add to $\mathcal{R}^+$ all the additions that happen in $\mathcal{R}$ based on the direct additions δ_i^+
2-4 $\mathcal{R}^+ = \mathcal{R}^+ \sqcup q((s_{i+1}, s_{i+2}, \cdots, s_N), \delta_i^+, \mathcal{D})$
// Add to $\mathcal{R}^-$ all the deletions that happen in $\mathcal{R}$ based on the direct deletions δ_i^-
2-5 $\mathcal{R}^- = \mathcal{R}^- \sqcup (n | n \in \mathcal{R} \wedge ResultPath_i(n) \in \delta_i^-)$
// Reduce Λ_i.
2-6 $\Lambda_i = (n | n \in \Lambda_i \wedge Pred_i^{after}(n) = true \wedge RP_i(n) = true)$
}
// Update the cached result $\mathcal{R}$ by $\mathcal{R}^+$ and $\mathcal{R}^-$
3- $\mathcal{R} = \mathcal{R} \sqcup \mathcal{R}^+$
$\mathcal{R} = \mathcal{R} - \mathcal{R}^-$

Figure 3: Incremental View Maintenance Algorithm for XML Path Expressions

5. EXPERIMENTS

In this section, we evaluate the performance of our incremental view maintenance algorithm. In our experiments, the system maintains one cached object (i.e., an XPath query result) and processes node updates one by one. For each update we compare the time required for incremental maintenance against the time required for the full view re-computation.

We used the XMARK benchmark [14] to generate source documents. We have generated two data sets of different sizes: Data set 1 (325,236 nodes), and Data set 2 (1,281,843 nodes).

The XML data is stored in a relational database. The node *id*s are generated using the ORDPATHs scheme [12]. Each node is represented as a row of a table with the following columns:

$$\{id, type, label, value, parent\_id\}$$

where *id* is a node identifier and *type* is a node type (element,attribute, or value). When *type* is 'element', *label* represents the element name. When *type* is 'attribute', *label* represents the attribute name, and *value* represents the attribute value. When *type* is 'value', *value* represents the data value. Although an ORDPATHs node *id* contains information about the *id* of the parent node, we use a column *parent_id* that stores the *id* of the parent for performance optimization.

Experiments are done using an Oracle 9i database on a PC with Linux 8.0, Pentium 4 1800 MHz CPU, and 1 GB memory.

We used the following two XPath queries:

- XPath Query 1:

```
/site/people/person[like(@id,'person2%')]/
name/text()
```

- XPath Query 2:

```
/site/people[person[like(@id,'person1%')]]/
person[like(@id,'person2%')]/name/text()
```

where "`like`" is a boolean predicate that corresponds to SQL's "`like`" operator. As we have mentioned before, the view maintenance algorithm does not have to know the definition of any predicate because the predicate evaluation is always delegated to the source data query processor.

The XPath Query 1 is implemented as the following SQL join query:

```
SELECT DISTINCT f.id
FROM x a, x b, x c, x d, x e, x f
WHERE a.type = 'element' and a.label = 'site'
and a.parent_id = '0' and b.type = 'element'
and b.label = 'people' and b.parent_id = a.id
and c.type = 'element' and c.label = 'person'
and c.parent_id = b.id and d.type = 'attribute'
and d.label = 'id' and d.value like 'person2%'
and d.parent_id = c.id and e.type = 'element'
and e.label = 'name' and e.parent_id = c.id
and f.type = 'value' and f.parent_id = e.id;
```

where "`x`" is the name of the table that contains the source nodes. Similarly, the XPath Query 2 is also implemented as a join query.

	Data set 1		Data set 2	
	Query 1	Query 2	Query 1	Query 2
Full re-computation (msec)	1459.61	4412.2	6549.28	83066.25
Incremental maintenance (msec)	134.13	237.01	355.03	1108.11

Table 1: Average time of full query and incremental maintenance

The Predicate test query for the XPath query 1 is implemented as the following SQL query:

```
SELECT *
FROM x c, x d
WHERE c.id = ?
and d.type = 'attribute' and d.label = 'id'
and d.value like 'person2%'
and d.parent_id = c.id;
```

where "?" represents a context node.

For each pair of a data set and a query, 100 source updates were randomly generated. The results of the time comparison for all the updates are shown in Figures 4(a), 4(b), 5(a), and 5(b). The average time of the full re-computation and of the incremental view maintenance for all the 100 updates in the four different configurations are shown in Table 1.

The figures and the table clearly establish the advantage of incremental view maintenance approach presented in this paper. For example, for the second data set and second query, the full query takes 80 times longer to execute. In fact, the results also show that the proposed algorithm scales well with both data size and query complexity. In particular, the full re-computation time for query 1 increases by 4.5X from data set 1 to data set 2. In contrast, the incremental view maintenance time only increases by 2.6X. Similarly, the full re-computation time for query 2 increases by 19X from data set 1 to data set 2. The incremental view maintenance time, on the other hand, increases only by 5X.

The table also shows more performance benefits for more complex queries over larger data sets: for the small data set and simple query (Data set 1 and Query 1), the improvement of incremental view maintenance is 10X over full re-computation whereas for the large data set and complex query (Data Set 2 and Query 2) the improvement is almost 80X.

The figures show that some updates have taken almost no time to be processed while other updates have taken a relatively significant time; this is because the former class of updates did not result in any source queries being issued, while the latter class resulted in issuing some source queries.

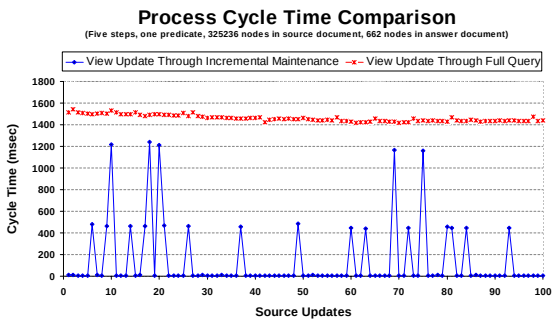
6. DISCUSSION

In this paper we have presented a new incremental view maintenance approach for XML views that are expressed by path expressions. The supported view specification language of path expressions is standard and powerful enough for a large class of real life applications. The size of the auxiliary data used is bounded as $O(M * N)$ where M is the size of the cached result and N is the size of the view specification expression. The size of the auxiliary data can not exceed this bound regardless of the complexity of the source XML tree and regardless of the complexity of the predicates used in the view specification path expression. Our algorithm delegates any predicate evaluation to the source query processor. The benefit of this delegation is that no auxiliary data is maintained for the evaluation of predicates. Without this delegation, the size of the auxiliary data can not be bounded. The proposed algorithm does not depend on any schemas for the source XML document, it can handle any general XML document. Regarding the efficiency of the maintenance process, the experimental results show that incrementally maintaining path expression views using the approach presented here is much faster than maintaining the views by recomputing the view specification query.

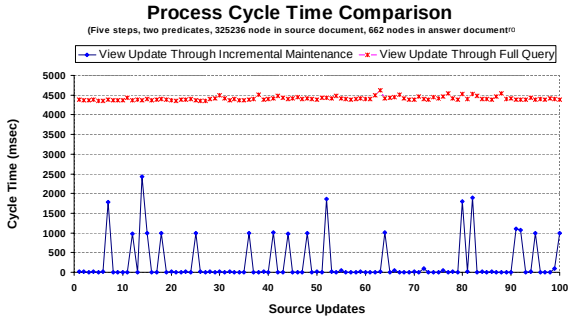
We have focused on processing the two primitive update operations of adding and deleting leaf nodes. Although these two operations are enough to express any complex transformation, it might be more efficient to handle a complex update, such as adding or deleting subtrees, holistically rather than by decomposing it into the primitive operations. In fact, the same ideas presented for the primitive updates can be extended to the complex updates of adding or deleting subtrees. In this case, the $\mathcal{U}.path$ becomes a branch that ends with a subtree dangling from the last node, this is the added or deleted subtree. Then, applying the Axis&Label test and the Predicates test on this branch will discover the direct effects. Once the direct effects are discovered, the indirect ones can be discovered in the same way as described above.

Generally, source updates may occur concurrently with the view maintenance process. For example, assume that an update $\mathcal{U}_1$ occurs and is reported to the view manager. Thus, the view manager initiates a view maintenance process to update the cached views to account for $\mathcal{U}_1$. At this time a new update $\mathcal{U}_2$ occurs at the source before the source query processor processes the queries which the maintenance process of $\mathcal{U}_1$ has issued to the source to maintain the views. In this case, processing these queries at the source will include the effects of $\mathcal{U}_2$ as well as those of $\mathcal{U}_1$. Then, when $\mathcal{U}_2$ is reported to the cache manager, a new maintenance process will be initiated to maintain the views according to $\mathcal{U}_2$. This second maintenance process will typically need to issue queries to the source to maintain the views. However, this second maintenance process could take advantage of the fact that the effect of $\mathcal{U}_2$ has already been incorporated in the answers of the queries that were issued in response to $\mathcal{U}_1$. If such cases are detected, the view maintenance process could be made more efficient by reducing the number of source queries used to maintain the views. One possible approach to detect such cases is to use timestamps for all the updates and the query answers received from the source. With that, the cache manager can determine which update effects have been incorporated in which answers. This approach needs to be investigated further in the context of caching XML views that are based on path expressions.

Caching systems normally cache the results of multiple expressions. Upon receiving an update $\mathcal{U}$ the proposed maintenance algorithm will be initiated to maintain every expression separately. However, if many of these expressions have significant overlap in terms of their structure, maintaining

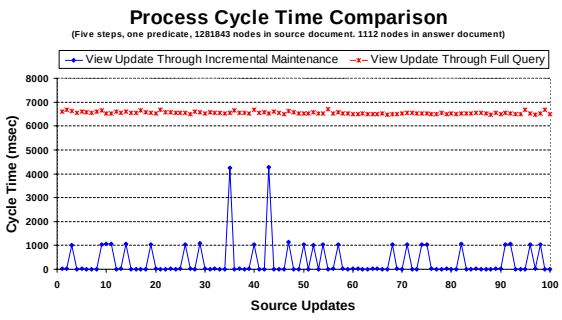


(a) Query 1 on Data Set 1

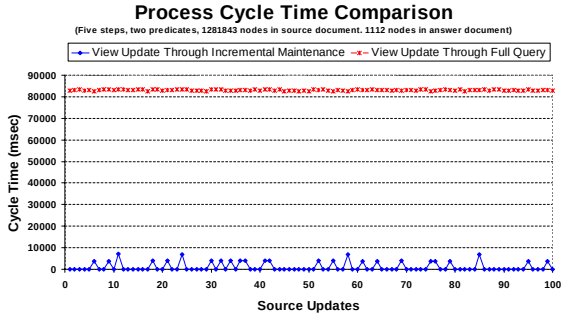


(b) Query 2 on Data Set 1

Figure 4: Incremental View Maintenance versus Full Re-Computation (Data Set 1)



(a) Query 1 on Data Set 2



(b) Query 2 on Data Set 2

Figure 5: Incremental View Maintenance versus Full Re-Computation (Data Set 2)

such expressions collectively will be more efficient. In particular, a cache server that maintains a large number of XML views will necessarily need a support of collective maintenance. We plan to investigate the possibilities of extending the proposed algorithm for maintaining multiple expressions collectively.

The approach presented in this paper delegates all predicate evaluations to the source query processor in order to avoid keeping unbounded auxiliary data and to preserve the privacy of the data provider. However, in some circumstances, the predicates might not need much of auxiliary data and the data provider might be willing to disclose the definition of some of the predicates. In this case, we could gain more efficiency by evaluating the predicates without source queries.

7. REFERENCES

[1] http://www.w3c.org/.

[2] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998.

[3] Andrey Balmin, Fatma Ozcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, 2004.

[4] Jos A. Blakeley, Per ke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[5] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An xml query language for heterogeneous data sources. In *WebDB (Selected Papers)*, pages 1–25, 2000.

[6] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling dynamic xml trees. In *PODS*, pages 271–281, 2002.

[7] Katica Dimitrova, Maged El-Sayed, and Elke A. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *ER*, pages 144–157, 2003.

[8] Timothy Griffin and Leonid Libkin. Incremental maintenance of views with duplicates. In *SIGMOD Conference*, pages 328–339, 1995.

[9] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems and techniques and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[10] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, 1993.

[11] Hartmut Liefke and Susan B. Davidson. View maintenance for hierarchical semistructured data. In *DaWaK*, pages 114–125, 2000.

[12] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD Conference*, pages 903–908, 2004.

[13] Luping Quan, Li Chen, and Elke A. Rundensteiner. Argos: Efficient refresh in an xql-based web caching system. In *WebDB*, pages 78–91, 2000.

[14] Albrecht Schmidt, Florian Waas, Martin L. Kersten, MichaelJ. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[15] Vnia Maria Ponte Vidal and Marco A. Casanova. Efficient maintenance of xml views using view correspondence assertions. In *EC-Web*, pages 281–291, 2003.

[16] Vnia Maria Ponte Vidal, Marco A. Casanova, and Valdiana da Silva Araujo. Generating rules for incremental maintenance of xml view of relational data. In *WIDM*, pages 139–146, 2003.

[17] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A dynamic index method for querying xml data by tree structures. In *SIGMOD Conference*, pages 110–121, 2003.

[18] Yue Zhuge and Hector Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, pages 116–125, 1998.