A SchemaGuide for Accelerating the View Adaptation Process*

Jun Liu[1], Mark Roantree[1], and Zohra Bellahsene[2]

[1] Interoperable Systems Group, Dublin City University
[2] LIRMM CNRS/University of Montpellier II
{jliu,mark.roantree}@computing.dcu.ie,
bella@lirmm.fr

Abstract. Materialization of XML views significantly improves query performance in the often slow execution times for XPath expressions. Existing efforts focus on providing approaches of how to reuse materialized view for answering XPath queries and, the problem of synchronizing materialized data in response to the changes taking place at data source level. In this paper, we study a closely related problem, the *view adaptation* problem, which maintains the materialized data *incrementally* after view definitions have been redefined/changed (*view redefinition*). Our research focuses on an efficient process for view adaptation upon the fragment-based view representation by segmenting materialized data into fragments and developing algorithms to update only those materialized fragments that have affected by the view definition changes. This serves to minimize the effect of view adaptation and provide a more efficient process for stored views. Additionally, we study the containment problem at fragment level under the constraints expressed in a so-name *SchemaGuide*. We have implemented our view adaptation system and we present in this paper the performance analysis.

1 Introduction

XML data is semi-structured, providing a flexibility and *bridge* between the more structured world of relational databases and the free form world of unstructured and web data. Its flexible nature also makes XML suitable for exchanging data between heterogeneous systems which led to its standardization as the format for information interchange on the Web. However, where applications are required to store data in native XML databases, query optimization is an ongoing problem. There have been many approaches to XML query optimization: SQL-based optimizers have been used in [1,2]; advanced tree-structured indexes were created to prune the search space in [3]; and XPath axis navigation algorithms were developed in [4]. More recently, efforts have focused on precomputing and storing the results of XML queries [5].

In relational database systems, materialized views are widely used for query result caching, especially where systems are queried more than updated such

* Funded by Enterprise Ireland Grant No. CFTD/07/201.

as in data warehouses. The XML systems, [6,7,8] provide different approaches using single views, while in [9,10,11,5], they focus on using multiple materialized XPath views. Furthermore, there has been increasing focus on the issue of XPath view updates and maintenance with [12,13] synchronizing the materialized view data with updates to source data. However, none of these approaches facilitate updates when view definitions have changed. This problem, first introduced by Gupta, et al [14], is known to as the *view adaptation* problem. Our motivation is to provide a more holistic approach to view based XML optimization by including view adaptation algorithms to manage query/view updates.

1.1 Contribution and Paper Structure

The core novelty in our work is in the provision of a framework where we materialize fragments of views that can be shared but also facilitate far more efficient view adaptation while view redefinition takes place very often.

– Based on our previous works [15,16], the XML Fragment-based Materialization (XFM) Approach, we have developed a new fragment-based containment checking mechanism which uses a construct called the *Schema Guide*. It is applied by our adaptation algorithms to facilitate the efficiency of detecting the containment relationship between different fragments in the XFM view graph.
– We devised a set of XPath view adaptation algorithms to efficiently handle the view redefinition caused by different type of changes.

This paper is structured as follows: in §2, we provide analysis of similar approaches to this topic; in §3, a brief review of the view graph and its components are provided; in §4, our containment checking approach is introduced, while in §5, the process for view adaptation is described; in §6, we provide details of our experiments and finally in §7, we offer some conclusions. A long version of the paper containing a detailed Related Work and an overview of the view adaptation system architecture can be found in [17].

2 Related Work

XPath query containment is a necessary condition for using materialized XPath views, and has been studied in [18,19,20]. Traditional the XPath containment problem is based on comparing two *tree patterns* containing a subset of XPath expressions (/, $*$, []). However, it has been proved in [19] that even for such a small subset of XPath expressions, the containment checking process is rather complex and time consuming. In this paper, we reduce the containment checking problem to fragment based comparison rather than tree patterns with the assistant of a so-named *SchemaGuide* which is a variation of the *QueryGuide* introduced in [21]. The cost of containment checking between two fragments is with respect to the number of schema nodes bound to each fragment.

Based on the containment checking approach, we also provide an XML Fragment Based View Adaptation approach. [22] claims that they are the first one providing a solution for the view adaptation problem in the XML world. They deal with view adaptation problem for XPath access-control views with a set of comprehensive incremental view adaptation techniques. Materialized data is represented to the users according to a set of access control rules. Based on these rules, data representation is restricted and dynamically changed to different users. However, this technique only operates when view adaptation starts from the output node of a query to its subtree. This is due to XPath semantics where only the XML fragment below the output node will be materialized. By getting inspiration from [23], we utilize multiple view fragments to achieve sharing of materialized data between difference XML views. Due to sharing of materialized fragments, view adaptation can theoretically take place at any point in the view construct. The number of accesses to the source data is significantly reduced by using our approach. While there will result in a significant benefit from our approach, one side effect is that the final result of each view still needs to be computed using the materialized fragments. However, the query evaluation is still more efficient than computing queries from scratch using the source data.

3 The XFM View Graph

The XFM View Graph is a directed acyclic graph including a combination of XPath views built by a set of fragments and operators. The graph exposes the common subexpressions between different view definitions. *Figure 1* is a sample XFM view graph representing sample XPath views, where fragments in gray indicates those fragments that are materialized. A detailed description of the XFM view graph with fragments and logical operators can be found in our previous works [16,15], and here, we give only a brief overview. Fragments are categorized into 5 types. Each fragment (except *Source Fragment*) represents the *result* of a single step in an XPath expression, and it is these fragments that can be shared across XML views. All fragments contain a set of $\mathcal{V}$-typed instances (XML tree nodes) for each node label $\mathcal{V}$. In the case of RF and SF fragments,

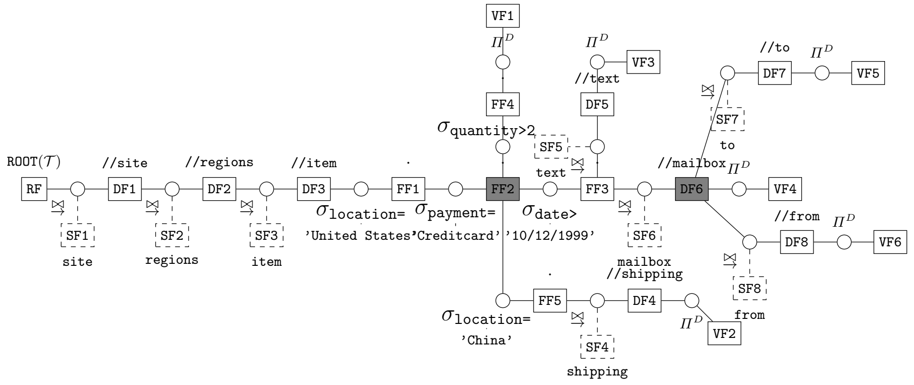


Fig. 1. XML Fragment Materialized View Graph

this will be the entire set of instances for $\mathcal{V}$. For the remaining fragments, there will generally be some subset of $\mathcal{V}$ generated for the fragment.

- `Root Fragment (`*RF*`)` - A Root Fragment represents a node sequence containing a single node, which is the root node of an XML tree $\mathcal{T}$ (also known as the document node). It always represents the starting point of a *XFM* view graph. While a view graph will contain multiple query representations, they are all joined by the same root fragment, as shown in *Figure 1* (e.g., *RF* with rectangle box).
- `Filter Fragment (`*FF*`)` - A filter fragment (e.g., *FF1* in *Figure 1*) represents the node sequence produced by a *select* operation. In our view model, the select operation always contains a predicate used to filter an input node sequence. e.g., **location='United States'** in *Figure 1* represents the filter operation that results *FF1.*
- `Dependency Join Fragment (`*DF*`)` - A Dependency Join Fragment (e.g., *DF1* in *Figure 1* represents the node sequence resulting from a *d-join* operation, e.g., the $\underset{\rightarrow}{\bowtie}$ before *DF1* represents a dependency join operation.
- `Source Fragment (`*SF*`)` - A Source Fragment represents the full set of $\mathcal{V}$-typed nodes. The major difference between this fragment and all others is that it cannot be reused and merely acts as an operand in a d-join operation. An example of a source fragment is shown in *Figure 1* with dashed boxes.
- `View Fragment (`*VF*`)` - A view fragment (e.g., *VF1* in *Figure 1*) represents the result of a view. It always follows a deep project operation, e.g., Π^D before *VF1* indicates a deep project operation.

Each fragment within a particular view is referenced by a corresponding *VF* fragment representing the context view. For example, *DF6* is referenced by *VF4*, *VF5* and *VF6* as it is shared by them whereas *DF8* is referenced only by *VF6.* We use V_n and *VFn* interchangeably to represent an XPath view, *VF1* and V_1 both represent XPath view 1. The fragmentation approach is used to facilitate this sharing of fragments across views as each fragment indicates a potential end point (materialization candidate) for a view. Furthermore, each fragment links to one or more fragments that directly after it, e.g., *FF2* links to *FF3*, *FF4* and *FF5* in *Figure 1.* Each fragment is also linked by one and only one fragment that directly precedes it, e.g., *FF2* is linked by *FF1.*

4 Containment Checking

In this section, we introduce our *SchemaGuide*, a metadata construct that optimizes the containment checking process. We then present the properties that manage the decision making process for containment.

4.1 The SchemaGuide

Our *SchemaGuide* is a heavily extended version of the *QueryGuide* introduced in [21] as it contains region encoding and a set of properties to govern containment checking and facilitate a more efficient fragment-based containment check. *Figure*

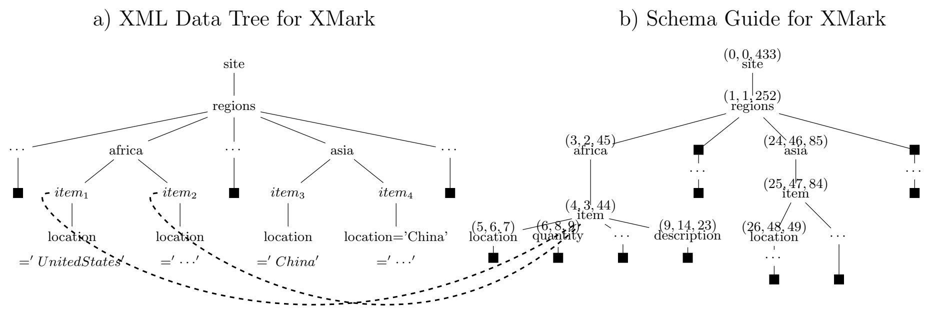


Fig. 2. Sample XML Data Tree and Schema Guide

2b is a sample *SchemaGuide* summarizing the XML tree in *Figure 2a.* They represent equivalent subsets of the *SchemaGuide* and the *XML Data Tree* for the XMark dataset respectively. Each node within the schema guide maps to a set of nodes with identical root-to-node path in an XML tree. For instance, $item_1$ and $item_2$ in *Figure 2a* both map to the schema node *item* within the subtree of *africa* in *Figure 2b.* From now on, we use the term *schema node* to represent any node within the *SchemaGuide*, and *node instance* for any node in the XML data tree.

A SchemaGuide (*SG*) is a tree-based data structure that describes the XML document structure. Each schema node (*sn*) in *SG* is defined by a tuple (*pid*, *start*, *end*), where *pid* is the unique identifier of each root-to-node (*rtn*) path within an XML data tree, `start` and `end` values encapsulate the sub-tree region (number of nodes) for that schema node. All nodes in the schema guide are encoded by the *StartEnd* encoding scheme, which is widely used in the XML twig pattern matching process, e.g., [24], to facilitate the determination of the parent-child or ancestor-descendant relationship between XML tree nodes. The *start* and *end* values of a node v together are called the *region* of node v, denoted as $\texttt{reg}(v)$. For two XML tree nodes, u and v, if v is in the subtree of u, then we say that u contains v, denoted by $\texttt{reg}(v) \sqsubset \texttt{reg}(u)$. In this paper, we make use of the *StartEnd* encoding scheme to determine the containment relationship between two sequences of schema nodes mapped to the sequences of node instances that are represented by the fragments. What is meant by containment between fragments is that as fragments represent sequences of node instances, therefore, for two sequences of node instances S_u and S_v, if every node instance in S_u is in the subtree of at least one node in S_v, then we say that S_v contains S_u, or S_u is contained in S_v. Because each node instance is mapped to a schema node within the *SchamaGuide*, therefore, in this paper we deal with containment problem at the schema level.

4.2 Region Containment

Mathematic notations $\sqsubset$, $\sqsupset$, $\equiv$ and $\|$ will be used to indicate the relationship *is contained in*, *contains*, *equivalent* and *incomparability* (disjoint) between either two schema nodes or sequences of schema nodes. Additionally, there is also *overlap* between two sequences of schema nodes. We use the function `Overlap` to determine how two sequences of schema nodes are overlapped.

Property 1 (Region Containment). For any two given schema nodes sn_u and sn_v in a schema guide SG, the followings hold:

1. $\texttt{reg}(sn_u) \sqsubset \texttt{reg}(sn_v)$, iff $sn_u.start > sn_v.start$ and $sn_u.end < sn_v.end$.
2. $\texttt{reg}(sn_u) \sqsupset \texttt{reg}(sn_v)$, iff $sn_u.start < sn_v.start$ and $sn_u.end > sn_v.end$.
3. $\texttt{reg}(sn_u) \equiv \texttt{reg}(sn_v)$, iff $sn_u.start = sn_v.start$ and $sn_u.end = sn_v.end$.
4. $\texttt{reg}(sn_u) \parallel \texttt{reg}(sn_v)$, iff $sn_u.end < sn_v.start$ or $sn_u.start > sn_v.end$.

Property 1 is used to determine the containment relationship between two schema nodes in a *SchemaGuide*. As shown in (see *Figure 2b*), we may wish to determine if *item* (4,3,44) is contained within *africa* (3,2,45). The start value 3 is greater than start 2, and end value 44 is less than end value 45, thus, the region of *item* **is contained in** the region of *africa*.

Based on *Property 1*, we use *Property 2* to detect the containment relationship between two sequences of schema nodes. This property is the core concept used to determine the containment relationship between fragments introduced in §3. This is due to the fact that each fragment represents a sequence of node instances, whereas each node instance further maps to a schema node within the *SchemaGuide*. Therefore, to check the containment between fragments, all we have to do is to check the containment relationship between two sequence of schema nodes associated with the fragments (or the sequence of node instances represented by the fragments). Therefore, we can detect the containment relationship between fragments using *Property 2*.

Algorithm 1: ContainmentCheck(F_u, F_v)

```
Input: two fragments
Output: a state indicating the containment
relationship between two inputs
1  if both $F_u$ and $F_v$ are Root Fragment then
2      $F_u \equiv F_v$;
3  else if $F_u$ is Root Fragment then
4      $F_u \sqsupset F_v$;
5  else if $F_v$ is Root Fragment then
6      $F_u \sqsubset F_v$;
7  else
8      $S_u = F_u \rightarrow$ GetSchemaNodes();
9      $S_v = F_v \rightarrow$ GetSchemaNodes();
10     FindRelationship($S_u$, $S_v$);
11     if both $F_u$ and $F_v$ are Filter Fragment then
12         compare predicate $p_u$ and $p_v$
13         associated with $F_u$ and $F_v$ respectively;
```

Property 2 (Sequence Containment). For any two given sequence of schema nodes $\mathcal{S}_u$ and $\mathcal{S}_v$, where $sn_i \in \mathcal{S}_u = \{sn_0 \cdots sn_n\}$ and $sn_j \in \mathcal{S}_v = \{sn_{n+1} \cdots sn_m\}$,

1. $\mathcal{S}_u \sqsubset \mathcal{S}_v$, iff for $\forall sn_i \in \mathcal{S}_u, \forall sn_j \in \mathcal{S}_v \rightarrow \texttt{reg}(sn_i) \sqsubset \texttt{reg}(sn_j)$
2. $\mathcal{S}_u \sqsupset \mathcal{S}_v$, iff for $\forall sn_i \in \mathcal{S}_u, \forall sn_j \in \mathcal{S}_v \rightarrow \texttt{reg}(sn_i) \sqsupset \texttt{reg}(sn_j)$
3. $\mathcal{S}_u \equiv \mathcal{S}_v$, iff for $\forall sn_i \in \mathcal{S}_u, \forall sn_j \in \mathcal{S}_v \rightarrow \texttt{reg}(sn_i) \equiv \texttt{reg}(sn_j)$
4. $\mathcal{S}_u \parallel \mathcal{S}_v$, iff for $\forall sn_i \in \mathcal{S}_u, \forall sn_j \in \mathcal{S}_v \rightarrow \texttt{reg}(sn_i) \parallel \texttt{reg}(sn_j)$
5. $\texttt{OVERLAP}(\mathcal{S}_u, \mathcal{S}_v) = true$, iff for $\exists sn_i \in \mathcal{S}_u$ and $\exists sn_j \in \mathcal{S}_v \rightarrow \texttt{reg}(sn_i) \sqsubset \texttt{reg}(sn_j)$ or $\texttt{reg}(sn_i) \sqsupset \texttt{reg}(sn_j)$

Property 2 is an extension to *Property 1* where containment checking involves two sequences of schema nodes. (1) to (4) are similar to *Property 1* except for the fact that we are comparing sequences of schema nodes. However, a new sub-property can emerge as sequences may overlap (5).

4.3 Containment Algorithm

Algorithm 1 is the containment checking algorithm used by our adaptation process based on the properties listed above. It takes two fragments as input. If none of the fragments is the *Root Fragment*, then we compare the sequence of schema nodes associated with each fragment (*Line 8-10 Algorithm 1*). *Algorithm 2* detects the relationship between two sequence of schema nodes mapped to the fragments. It takes two sequences of schema nodes as input. If both fragments are the *Filter Fragment*, we then compare the predicates as well. Comparing predicates is straight forward and due to the space limitation, we do not list the algorithm here. The output of the `ContainmentCheck` is one of the **containment relationship** mentioned in *Property 2*. The time complexity of our fragment-based containment checking algorithm is $O(n \times m)$, where n and m are the number of the schema nodes within the sequences associated with the input fragments.

Algorithm 2: FindRelationship(S_u, S_v)

```
Input: $S_u$, $S_v$ contains a sequence of schema nodes labeled $u$ and $v$ respectively
Output: a state indicating the containment relationship between two inputs
1  if $|S_u| = |S_v|$ then
2      if $\forall u_i \in S_u, \forall v_j \in S_v \rightarrow u_i \equiv v_j$ then
3          return $S_u \equiv S_v$;
4      else if $\exists u_i \in S_u, \exists v_j \in S_v$
5      $\rightarrow u_i \sqsubset v_j$ or $u_i \sqsupset v_j$ then
6          return $S_u$ is overlap with $S_v$;
7      else $S_u \parallel S_v$;
8  else
9      if $\forall u_i \in S_u, \forall v_j \in S_v \rightarrow u_i \sqsubset v_j$ then
10         return $S_u \sqsubset S_v$;
11     else if $\forall u_i \in S_u, \forall v_j \in S_v \rightarrow u_i \sqsubset v_j$ then
12         return $S_u \sqsupset S_v$;
13     else if $\exists u_i \in S_u, \exists v_j \in S_v$
14     $\rightarrow u_i \sqsubset v_j$ or $u_i \sqsupset v_j$ then
15         return $S_u$ is overlap with $S_v$;
```

5 View Adaptation

The adaptation methods consist of two types of adaptations, *structural adaptation* and *data adaptation*. The structural adaptation is the adaptation process maintaining the structure of the XFM view graph in response to the changes. This is followed by the data adaptation process that decides on the materialized fragments to be adapted. Our effort so far has focused mainly on structural maintenance and containment. We begin with an overview of the adaptation process and identify the key algorithms.

5.1 Adaptation Method

The main idea of view adaptation is to maintain the XFM view graph both structurally and physically updated in response to the changes applied to the XFM view graph. Based on the effect taking place on the XFM view graph, the *structural adaptation* is further categorized into three types, *integration*, *deletion* and *modification* of a fragment. An abstraction of the adaptation process flow can be summarized as follows:

1. Detect whether the adaptation process (e.g., the integration, deletion or modification of a target fragment) affects other views beyond the target view.

2. For integration and modification, iteratively calling the containment check subprocess to find an existing fragment containing (including) the target fragment. Such a fragment must be the most restricted one compared to other fragments that also contain the target fragment. This means that there may be many fragments containing the target one, however, the desired one is the one contains least node instances.
3. Adapt the structure of the XFM view graph based on the results returned by the above two processes.
4. Find any existing materialized fragment that is affected by the requested change.
5. Search for any existing materialized fragment that can be reused to physically adapted the affected fragments in response to the change.

The following methods are used to implement the adaptation process.

– `GetNextByRef` retrieves the fragment directly is directed linked by the context fragment in a target view.
– `GetPrevious` retrieves the fragment preceding the context fragment.
– `ProcessNFs` and `ProcessPFs` recursively check the containment relationship between the fragments directly linked by or precedes the context fragment respectively, and then adapt the XFM view graph in response to different containment relationship detected.
– `GetDuplicatedFragment` returns a copy of the specified fragment. Only the structural and mapping information will be duplicated, not the materialized data.
– `RemoveReference` deletes the reference between a fragment and a view.
– `AddNext` adds a link between two fragments.
– `RemoveNext` removes link between two fragments.

A detailed description of the methods listed above can be found in our technical report [25]. In the rest of this section, due to space limitation, we will give a detailed description of one type of adaptation method only. The others can also be found in our technical report [25].

5.2 Worked Example

In this section, we use a worked example for the modification of a predicate. The main issue in this case is the containment relationship between the target predicate and its preceding predicates (if exists). Additionally, it is also essential to check whether the predicate being modified is shared by other views or not. It is straight forward when the target predicate is only referenced by one view, however, further effort is required when the predicate in question is shared by more than

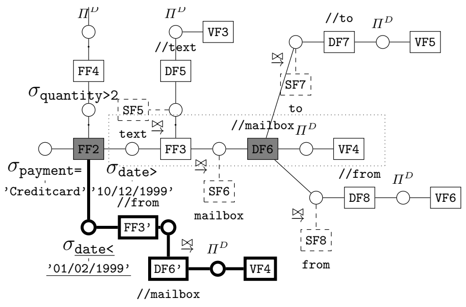


Fig. 3. Modify Predicate

one XPath views in the XFM view graph. *Figure 3* gives an example of modifying a predicate, suppose we would like to modify the predicate **./date** $>$ **'10/12/1999'** represented by $FF3$ in V_4 to **./date** $<$ **'01/02/1999'**. Note that $DF6$ is materialized by V_4 and shared by different views as well.

Initially, the requested change is transformed into the fragment representation, $FF3'$. The new fragment ($FF3'$) is then compared to the target fragment ($FF3$). If they are equivalent (*Line 2*), then the original XFM view graph is returned. This is more like a validation process.

Nevertheless, if they are not equivalent (*Line 3*), then depending on whether the target fragment, $FF3$, is shared by different views (*Line 5*) or not (*Line 21*), we take a different approach for structural adaptation.

In this case, $FF3$ is shared by views, V_3, V_4, V_5 and V_6. The actual containment relationship between the target fragment ($FF3$) and the new fragment ($FF3'$) is then required to be determined. Different containment relationship is expected (*Line 6,8 and 10*). If $FF3$ contains (includes) $FF3'$ (*Line 6*), `ProcessNFs` is then called to process the following fragments, e.g., $DF6$, linked by $FF3$ (*Line 7*) in the target view. The purpose of this method is to recursively detect the containment relationship between the new fragment and the fragments linked by the target fragment, and eventually, adapt the structure of $\mathcal{G}$.

Algorithm 3: modifyPredicate($\mathcal{G}$, F_c, F_n, *ref*)

```
Input: the XFM View Graph $\mathcal{G}$,
$F_c$ is the target fragment being modified,
$F_n$ is the new fragment after applying the modification,
the context view reference ref
Output: an updated MFM View Graph $\mathcal{G}$
1  begin
2      if $F_c \equiv F_n$ then return $\mathcal{G}$;
3      else
4          $F_p = F_c \rightarrow$GetPrevious();
5          if $F_c \rightarrow$IsShared() then
6              if $F_n \sqsubset F_c$ then
7                  return ProcessNFs($\mathcal{G}$, $F_c$, $F_n$, ref);
8              else if $F_c \sqsubset F_n$ then
9                  return ProcessPFs($\mathcal{G}$, $F_p$, $F_c$, $F_n$, ref);
10             else
11                 if $F_p \sqsubset F_n$ then
12                     ProcessPFs($\mathcal{G}$, $F_p$, $F_c$, $F_n$, ref);
13                 else
14                     $F_c \rightarrow$RemoveReference(ref);
15                     $F_p \rightarrow$AddNext($F_n$);
16                     $size = F_c \rightarrow$GetNextsCount() ;
17                     for $int\ i = 0$ to $size$ do
18                         $F_{cn} = F_c \rightarrow$GetNext($i$);
19                         add ref to all fragments linked to $F_{cn}$;
20                 return $\mathcal{G}$;
21         else
22             if $F_p \sqsubset F_n$ then
23                 return ProcessPFs($\mathcal{G}$, $F_p$, $F_c$, $F_n$, ref);
24             else if $F_n \sqsubset F_p$ or OVERLAP($F_p$,$F_n$)$\rightarrow$true
25             or $F_p \parallel F_n$ then
26                 return ProcessNFs($\mathcal{G}$, $F_p$, $F_n$, ref);
27             else
28                 $F_p \rightarrow$RemoveNext($F_c$);
29                 $F_{cn} = F_c \rightarrow$GetNextByRef(ref);
30                 $F_p \rightarrow$AddNext($F_{cn}$);
31                 return $\mathcal{G}$;
```

If the target fragment ($FF3$) is contained in (more restricted than) the new fragment ($FF3'$) (*Line 8*), we then call `ProcessNFs` to recursively detect the containment relationship between the fragments preceding the target fragment (e.g., $FF3$) and the new fragment ($FF3'$) until either a relationship other than *incomparability* is detected or the *Root Fragment* is reached. The structure of $\mathcal{G}$ is then adapted. However, if the relationship between the target fragment and the new fragment is either *incomparability* or *overlap* (*Line 10*), the process will then take the third approach (*Line 11-20*).

In this case, it is obvious that the predicates do not overlap, which means $FF3$ and $FF3'$ is disjoint. As a result, we compare the fragment ($FF2$) preceding $FF3$ to $FF3'$ (*Line 11*). If the preceding fragment is contained in $FF3'$, `ProcessPFs` is then called to continuously process the preceding fragments and adapt the structure of the graph. However, if any other relationship is found, we link the new fragment to the fragment preceding the target fragment (*Line 15*), and we remove the view reference between the target fragment and the target view (*Line 14*). In the current example, because $FF2$ contains $FF3'$, we add a link between $FF2$ and $FF3'$. As fragments after $FF3$ in V_4 are also shared by other views (in this case $DF6$ only), therefore, we need to also get a copy of the $DF6$, $DF6'$, mapping to the same sequence of schema nodes. We then link $DF6'$ to $FF3'$ as shown in *Figure 3*.

```
Algorithm 8: AdaptFragment($\mathcal{G}$, $F_n$, $ref$)
Input: the MFM View Graph $\mathcal{G}$,$F_n$ is the adapted fragment, ref specified the
       target view
Output: an adapted MFM view $\mathcal{G}$
1  $F_{nn} = F_n \rightarrow$GetNextByRef(ref);
2  while $F_{nn}$ is not View Fragment do
3      if $F_{nn}$ is materialized then
4          $F_p = F_{nn}$;
5          while $F_p$ is not Root Fragment do
6              if $F_p$ is materialized then
7                  adapt $F_m$ using $F_p$;
8                  return;
9              $F_p = F_p \rightarrow$GetPrevious();
10         materialize $F_m$ from scratch;
11         return;
12     $F_{nn} = F_{nn} \rightarrow$GetNextByRef(ref);
13 return $\mathcal{G}$;
```

Fig. 4. Data Adaptation

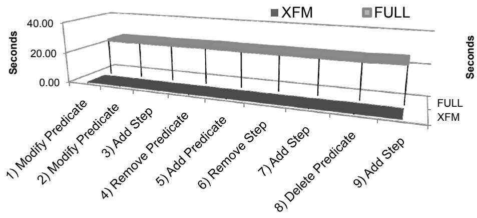
5.3 Maintenance of Materialized Fragment

The above process maintains only the structure of the XFM view graph $\mathcal{G}$ in response to the change. We now give a brief description of the data adaptation algorithm for maintenance of the existing materialized fragments that are affected by the change. *Algorithm 8* is the algorithm we used for adapting the materialized data in response to the change. In the current version, we simply search for an existing materialized fragment that we can reuse to maintain the affected fragment (*Line 5-9*). The cost of using such an existing fragment is less than other approaches as we will show in §6.

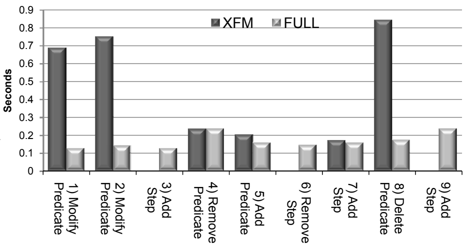
If there is a materialized fragment that is affected by the adapted view (*Line 3*), we search for the fragments that precede the adapted fragment which can be reused by the affected one (*Line 5*). In our future work, we will provide a solution to adapt the affected materialized fragments by either inserting extra data into or removing redundant data from the fragments.

6 Results of Experiments

In this section, we demonstrate the performance of different adaptation methods. We then compare our *XFM* adaptation approach to the Full Materialization (*FULL*) approach which is more traditional and based on the materialization of entire views.



(a) Query Processing Cost



(b) Materialisation Cost

Fig. 5. Evaluation Result

6.1 Experiment Setup

Two XML database servers were deployed for this experiment: a remote MonetDB server and a local MonetDB server. The remote server contains all XML source data, and the local server stores all XPath views and their fragments. This models a typical data warehousing system where data is often distributed due to the high volumes generated. We use version 4.34.4 for both remote and local MonetDB servers. The remote MonetDB server is distributed on an Intel Core(TM)2 Duo 2.66GHz workstation running 64-bit Fedora Server version 12. The local MonetDB server is installed on an Intel Core(TM)2 Duo CPU 3.00GHz Windows 7 workstation. The second server contains all XPath materialised views with data obtained from the remote site.

We use XMark benchmark in our experiment, which generates a narrow and deep XML document with maximum 13 levels. The document we generated is 1GB. We implemented all our adaptation algorithms using Java. The *SchemaGuide* is built during the document parsing time and stored in the main memory. Our XFM view graph contains 15 XPath views including the XPath views demonstrated in *Figure 1*. The XFM view graph is built automatically by an implemented Java programm. The total size of the materialized data within the chosen fragments is 27MB, only 2.6% of the original document. The list of queries are contained in an accompanying technical report which contains a more detailed breakdown and analysis of the experiments [25].

6.2 Experimental Analysis

We now demonstrate the performance of view adaptation using *XFM* and *FULL* approaches. The experiment is initialized by materializing XPath views with data obtained from the remote MonetDB server. To demonstrate the benefit of *XFM* approach, we assume that the *XFM* approach will always uses data from the materialized fragments shared across the XPath views. Concerning the query processing cost, the full materialization approach clearly outperformed the fragment-based approach since some fragments are still virtual (not materialized). However, we show that when considering the total adaptation cost, our fragment-based approach provides superior optimization. Our experiment demonstrates the general cost of both structurally and physically adapting the existing fragments in response to a sequence of mixed types of changes. Our approach provides a trade off between maintenance cost and query processing cost, which is important in a volatile query environment.

We apply a sequence of changes to the 15 XPath views. The changes are a combination of different types, e.g., Add a Predicate or Remove a Step. Examples of changes that were introduced (from the full set in [25]) are modifying the predicate **date>'10/12/1999'** to **date>'01/04/2002'** in V_4 and adding a step **/descendent::description** in V_3 before the step **/descendant::text**. This sequence of changes are evaluated sequentially by both *XFM* and *Full* approaches. The evaluation analysis is based on the following costs: i) Recomputing Cost, the cost of evaluating the query (view) after applying the change; ii) Data Transferring Cost, the cost of transferring the query result from the remote XML database server to the local XML database server; iii) Materialization Cost, the cost of parsing the XML query result (an XML document) and storing them into the local XML database server. The data transferring cost does not count for the *XFM* approach as we assume that it reuses the existing fragments on the local server in response to the changes. Therefore, we use the terms **Query Processing Cost** indicating the sum of the Recomputing Cost and Data Transferring Cost.

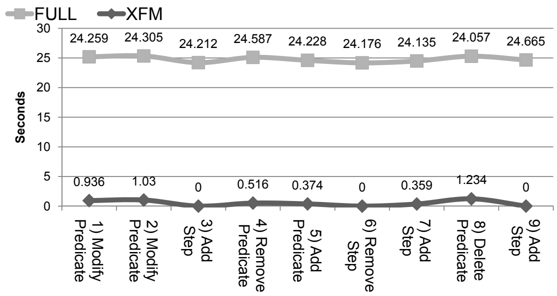


Fig. 6. Total Adaptation Cost

Figure 5a lists the query processing cost for performing a sequence of changes on the XFM view graph. The type of the changes are displayed on the x-axis and are applied from left to right. It is clear that our approach is far more efficient than *FULL* approach as we reuse the existing materialized fragment (27MB) during the adaptation process. On the other hand, the *Full* approach must recompute the query after applying each change, and additionally it has to query the remote MonetDB database (>1GB) and retrieve data from it. *Figure 5b* demonstrates the efficiency of the materialization between both approaches.

- For changes 1, 2 and 8, the *XFM* approach requires much more time for materialization. The reason for this is because the *FULL* approach rematerializes only the target view after each requested change. However, in the *XFM* approach, we must rematerialize the affected fragment which in this

case is a super set of the result set generated by the target view. In another words, the data set the fragment contains is larger than the actual size of the view result.
- For changes 4, 5 and 7, as the data set contained in the affected fragment is close to the final view result, therefore, the materialization cost are similar for both approaches.
- There is no materialization cost for changes 3, 6 and 9 for the *XFM* approach. This is due to the fact that, in some cases, a change may not affect any existing materialized fragment at all, it will only require an structural adaptation of the XFM view graph.

Although both approaches have similar average performance cost for materialization, when discussing the overall view adaptation cost, our approach provides superior *adaptation* performance. *Figure 6* gives the total adaptation cost after applying each change. It is obviously that our approach is much more efficient than the *FULL* approach. This is because rather than recomputing the query from scratch after applying each change, we adapt the affected materialized fragment by reusing the existing ones. In our example the total size of the existing materialized fragments is only approximately 2.6% of the source dataset. Therefore, even through there were no data transferring cost, querying the existing materialized data is much more efficient than querying the source dataset (1GB). However, the *FULL* approach has to recompute the query and retrieving data from the remote database. It will be even worse when data required is located on the multiple sites.

7 Conclusions

In this paper, we presented a fragment based view adaptation method for materialized XPath views. The benefit of our approach is in the event of a change to views, it is not necessary to recompute the entire view. Additionally, the containment problem between two XPath views is reduced down to the fragment based comparison rather than query based. We provide a SchemaGuide in this paper to facilitate the view adaptation process by efficiently determining the containment relationship. From our experimental analysis, we have shown the significant gains in performance can be achieved from this approach. While the fragmented approach has been applied in relational databases, it has not been used in XML systems, as in our approach. The structure of XML trees together with the more complex format for expressions provides a more significant challenge.

In our current version of XFM, we manually select fragments for materialization when building the view graph. In future work, we are creating a cost based method for fragment selection, which will provide for a more fully automated fragment based approach to XML view materialization.

References

1. Boncz, P.A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J., Teubner, J.: MonetDB/XQuery: A Fast XQuery Processor Powered By A Relational Engine. In: SIGMOD 2006 (2006)

2. Marks, G., Roantree, M.: Metamodel-Based Optimisation of XPath Queries. In: BNCOD 2009 (2009)
3. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD 2002 (2002)
4. Grust, T.: Accelerating XPath Location Steps. In: SIGMOD 2002 (2002)
5. Tang, N., Yu, J.X., Tang, H., Özsu, M.T., Boncz, P.A.: Materialized view selection in xml databases. In: DASFAA 2009 (2009)
6. Arion, A., Benzaken, V., Manolescu, I., Papakonstantinou, Y.: Structured Materialized Views for XML Queries. In: VLDB 2007 (2007)
7. Balmin, A., Özcan, F., Beyer, K.S., Cochrane, R., Pirahesh, H.: A Framework for Using Materialized XPath Views in XML Query Processing. In: VLDB 2004 (2004)
8. Lakshmanan, L.V.S., Wang, H., Zhao, Z.: Answering Tree Pattern Queries Using Views. In: VLDB 2006 (2006)
9. Cautis, B., Deutsch, A., Onose, N.: XPath Rewriting Using Multiple Views: Achieving Completeness and Efficiency. In: WebDB 2008 (2008)
10. Gao, J., Wang, T., Yang, D.: MQTree Based Query Rewriting over Multiple XML Views. In: Wagner, R., Revell, N., Pernul, G. (eds.) DEXA 2007. LNCS, vol. 4653, Springer, Heidelberg (2007)
11. Tang, N., Yu, J., Ozsu, M., Choi, B., Wong, K.F.: Multiple materialized view selection for xpath query rewriting. In: ICDE 2008 (2008)
12. Sawires, A., Tatemura, J., Po, O., Agrawal, D., Candan, K.S.: Incremental Maintenance of Path-Expression Views. In: SIGMOD 2005 (2005)
13. Lim, C.H., Park, S., Son, S.H.: Access Control of XML Documents Considering Update Operations. In: XMLSEC 2003 (2003)
14. Gupta, A., Mumick, I.S., Ross, K.A.: Adapting Materialized Views after Redefinitions. In: SIGMOD 1995 (1995)
15. Liu, J., Roantree, M., Bellahsene, Z.: Optimizing XML Data with View Fragments. In: ADC 2010 (2010)
16. Liu, J., Roantree, M.: Precomputing Queries for Personal Health Sensor Environments. In: MEDES 2009 (2009)
17. Liu, J.: A SchemaGuide for Accelerating the View Adaptation Process. Technical report, Dublin City University (2010), `http://www.computing.dcu.ie/~isg/`
18. Deutsch, A., Tannen, V.: Containment and Integrity Constraints for XPath. In: KRDB 2001 (2001)
19. Miklau, G., Suciu, D.: Containment and Equivalence for a Fragment of XPath. Journal of the ACM 51, 2–45 (2004)
20. Neven, F., Schwentick, T.: On the Complexity of XPath Containment in the Presence of Disjunction, DTDs, and Variables. CoRR (2006)
21. Izadi, S.K., Härder, T., Haghjoo, M.S.: S3: Evaluation of Tree-Pattern XML Queries Supported by Structural Summaries. Data and Knowledge Engineering 68, 126–145 (2009)
22. Ayyagari, P., Mitra, P., Lee, D., Liu, P., Lee, W.C.: Incremental Adaptation of XPath Access Control Views. In: ASIACCS 2007 (2007)
23. Bellahsene, Z.: View Adaptation In The Fragment-Based Approach. IEEE Transactions on Knowledge and Data Engineering 16, 1441–1455 (2004)
24. Liu, J., Roantree, M.: OTwig: An Optimised Twig Pattern Matching Approach for XML Databases. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM 2010. LNCS, vol. 5901. Springer, Heidelberg (2010)
25. Liu, J.: Schema Aware XML View Adaptation. Technical report, Dublin City University (2010), `http://www.computing.dcu.ie/~isg/`