Optimizing XML Data with View Fragments

Jun Liu[1] **Mark Roantree**[1] **Zohra Bellahsene**[2]

[1] Interoperable Systems Group
Dublin City University,
Dublin, Ireland
Email: jliu,mark.roantree
@computing.dcu.ie

[2] Computer Science at the University of Montpellier II,
161 rue Ada, F34392 Montpellier Cedex 5,
Email: bella@lirmm.fr

Abstract

As web-based applications and data continue to grow, large caches of XML data will result in many application domains. In sensor web applications, there are continuous streams of sensor data being generated, converted to XML and stored for domain queries and data mining purposes. The main problem with these XML caches is that existing XML database queries are very slow, especially for large databases or those with complex structures. In this work we propose a view-based system to XML optimization where the most popular or well-chosen queries are materialized and fragmented to greatly improve the performance of all XML queries.

Keywords: XPath View, multi-fragment, View Adaptation

1 Introduction

The Sensor Web is a natural extension to the WWW where small hardware devices are used to automatically generate new data. In our research area, we are monitoring a group of elite athletes using various personal monitors. The collecting, processing and integration of data was described in earlier research (Roantree et al. 2008). However in this work, we made no attempt to optimize XML queries and instead focuses on managing the sensor data.

Web-based data interfaces are used by athletes to upload this data where it is converted to XML, semantically enriched and integrated with other data streams. Databases are then queried using standard XML languages such as XPath or XQuery to query and mine these XML stores. One of the problems facing scientists and end-users is that XML queries perform poorly when compared with other database systems. There have been many approaches to XML query optimization where SQL-based optimizers are used (Boncz et al. 2006, Marks & Roantree 2009), advanced tree-structured indexes are created to prune search space (Bruno et al. 2002) or XPath axis navigation algorithms (Grust 2002).

Our solution to this problem is to adopt an existing approach to optimization from the relational database world where queries are precomputed and stored as views on data servers. Subsequent queries make use of these views in order to execute and generate query results in far quicker times. The challenge is to make a solution for table based data stores applicable in the tree-based world of XML data.

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

1.1 Paper Structure

This paper is structured as follows: in the remainder of this section we provide the motivation for this work and outline our contribution to research in this area followed by a detailed discuss of related research in §2; we then provide a brief overview of the XML view language together with a set of sample views in §3; in §4 we introduce our *Multiple Fragment Materialization* (*MFM*) view graph together with a detailed description of its constructs, fragments and operators; we give an example of how the *MFM* graph is constructed using the sample XML views, and a summary of the transformation between view expressions to *MFM* view graph in §5; in §6 we show the experimental results with a comparison of our approach against the traditional full materialization approach; and finally §7 concludes the paper and outlines future work.

1.2 Motivation

In relational database management systems (RDBMS), materialized views are widely used for query result caching and query answering. The impact on using materialized views for query answering is significant to the improvement of query performance. Materialized views are considered to be most beneficial where systems are queried more often than updated such as in data warehouses. As a result, views have also been studied in XML database management system (XDBMS). XML views are usually formed by a subset of the XPath language denoted by $XP^{\{[],*,//,/\}}$. In the context of XDBMS, materialized XPath views can also be used to expedite the processing of XML queries. (Arion et al. 2007, Balmin et al. 2004, Lakshmanan et al. 2006) studies the single view-based query answering problem. Whereas (Cautis et al. 2008, Gao et al. 2007, Tang et al. 2008, 2009) focus on using multiple materialized XPath views for answering queries.

In addition, there has been increasing attention on the issue of XPath view updates and maintenance. (Sawires et al. 2005, Lim et al. 2003) deal with synchronizing the materialized view data with updates to the source data. However, there is very little work that has been done for handling updates when view definitions are changed. This problem is referred to as the *view adaptation* problem, which was first introduced by Gupta, et al (Gupta et al. 1995) in RDBMS. Padmapriya et al. (Ayyagari et al. 2007) claimed to be the first to focus on XML view adaptation using access rules. However in their system, view adaptation can only take place below the output node of the XPath expression or it must retrieve the data from source. This is due to the fact that only XML

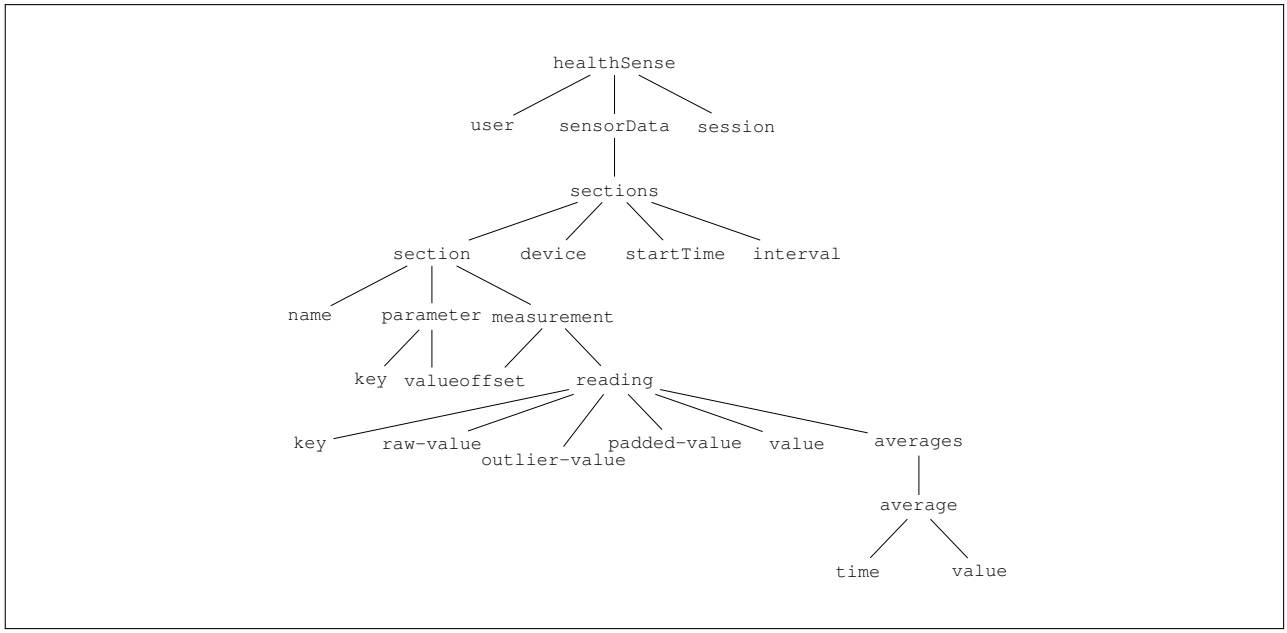


Figure 1: Sensor Database Schema

data fragment below the output node is materialized as depicted in (Tang et al. 2008). Furthermore, data reusability is rather poor as materialized data fragments cannot be shared across views.

The view adaptation problem is well understood in relational database systems. A solution by Bellahsene (Bellahsene 2004) proposed a multi-fragment based approach, where view materialization takes place on the fragment level. Here, data reusability is significantly improved as fragments are shared between different views. However, in order to apply this approach in an XML database, a query transformation approach is required to convert XML query expressions (XPath or XQuery) into SQL expressions. Since the difference between XML query languages (nested, irregular, heterogeneous and ordered) to the SQL language (flat, regular, homogeneous and unordered), the language transformation will be difficult. We adopted the idea of the fragment approach and applied it to XML databases. Our multi-fragment view materialization approach improves the data reusability by sharing common sub-expressions among views in the form of view fragments. In this way, any change takes place in view definition above the output node can be resolved by searching for materialized data fragments across different views.

1.3 Problem Description

The schema illustrated in *Figure 1* represents a subset of the **HealthSense** dataset, compiled in our project. We now present a list of views, expressed in XPath, which form part of our materialized fragments. Each are based on the *reading* node (*outlier*, *padded*, *value* and *average*) between specified times (determined by timing offsets). We also provide a simple explanation for those readers unfamiliar with XPath.

Example 1
Find all ***reading*** *data from* ***healthsense measurement*** *recorded by the sensor device where the* ***offset*** *exceeds 10 seconds.*

//healthSense//measurement[./offset>10000]/reading

This XPath view materializes all subtrees rooted at nodes named ***reading****, where each* ***reading*** *node must have a parent node called* ***measurement*** *with child* ***offset*** *having values greater than 10000 (as values are recorded in milliseconds).*

Example 2
Locate sensor ***reading*** *nodes where* ***offset*** *exceeds 5 seconds.*

//healthSense//measurement[./offset>5000]/reading

This XPath view materializes all subtrees rooted at the node ***reading****, where each must have a parent node called* ***measurement*** *with a child* ***offset*** *with value great than 5000.*

Example 3
Find sensor data with ***offset*** *values between 5000 and 10000 milliseconds.*

//healthSense//measurement[./offset>5000][.//offset< 10000]/reading

Example 1 demonstrates an existing XML view that stores all measurement **reading** data recorded after *10000* milliseconds. If we modify the definition of the view in *Example 1* by replacing *10000* to *20000*. The answer to this new query can be computed easily from the data that has already been stored in *Example 1*. This can be achieved by removing all reading data that were measured before *20000* milliseconds. This incremental computation is much more efficient than recomputing the view from scratch.

However, the change to the view definition is not always so easily computable. Assuming that we modify *Example 1* by changing the offset time to *5000*, e.g., *Example 2*, then the reading data measured between *5000* and *10000* are not computable using *Example 1*. Nevertheless, we can still reuse the old view (see *Example 1*) for all reading data after *10000* milliseconds, and then all the rest from the base data, XML database.

Finally, assume that *Example 3* is also an existing materialized XML view. By observation, we know that *Example 2* can be answered using *Example 1* and *Example 3*. However, existing approaches for XML query answering can make use of one materialized view at a time as fragments cannot be shared between materialized views.

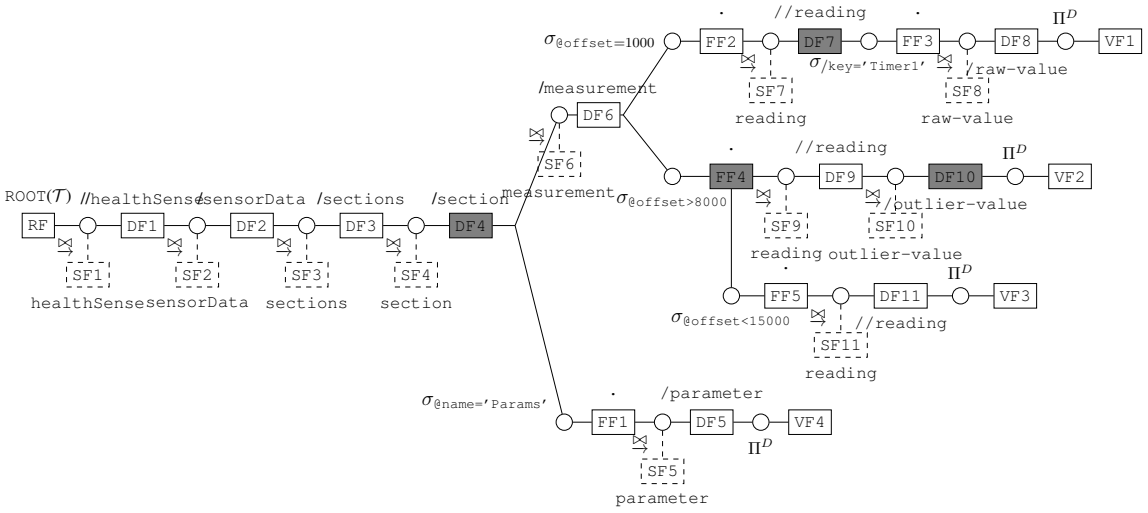


Figure 2: Multi-Fragment View Graph

1.4 Contribution

In this paper, we present a multi-fragment based XPath view materialization approach for an XML database that can utilize a set of materialized fragments as opposed to a collection of single materialized views. This approach can be easily applied by view adaptation algorithms.

Our key contributions can be outlined as follows:

- We provide Multiple Fragments Materialization View Graph (*MFM* view graph) that exploits the materialization of common XPath subexpressions to reduce the cost of view adaptation in terms of multi-fragment approach. While this approach has been previously employed in the set-based relational model, it has never been applied to tree-based systems.
- We present a specialized set of *operator*s and *fragment*s for constructing the *MFM view* graph.
- Our experimental evaluation demonstrates that significant performance improvement over the more traditional single-fragment approach can be achieved.

2 Related Research

Early efforts on view adaptation focused on keeping the materialized view up-to-date in response to query changes (Gupta et al. 1995). In this work, the authors demonstrate how the view adaptation problem differs from the problem of *query rewriting* by showing the new view to not always being equivalent to existing views. The view adaptation problem will only apply a sequence of *local change*s, such as add an attribute or delete a base relation. They also claim that this problem is closely related to the systems that are queried more frequently than updated. They show how these views can be adapted using an existing materialization for the cases where it is possible to do so. They identify extra information that can be kept with a materialization to facilitate redefinition.

In (Bellahsene 2004), the authors presented a fragment based view adaptation approach for the relational data model. They exploit materialization of common subexpressions to reduce the cost of view adaptation in the fragment-based approach. Nevertheless, the data independence is preserved for the views that are not affected by the change. In doing so, they provide the ability to reuse all the materialized fragments in the system to reduce the number of access commands on source data. However, to apply this approach to a semi-structured data model, a query transformation approach is required to convert the XML query languages into the SQL language. Due to the difference between XML query languages and SQL, this requires a new set of operations to manage fragments and materialization.

In the area of semi-structured data, view updates with changes to source data has been studied by a number of research teams, e.g., (Sawires et al. 2005, Lim et al. 2003). However, the view adaptation problem in XML has attracted little focus. Recently, a related approach, *access control view*, has gained an increasing interest in supporting fine-grained XML access control (Damiani et al. 2000). Towards this end, techniques such as XPath security views (Fan et al. 2004, Kuper et al. 2005, Stoica & Farkas 2002, Ayyagari et al. 2007) are being used. However, (Ayyagari et al. 2007) are among the first to provide a solution for view adaptation in XML databases. They provide 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 (XPath expressions). Based on these rules, data representation is restricted and dynamically changed for 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 of the output node will be materialized.

We differ from these approaches as we adapted an idea from (Bellahsene 2004) utilizing multiple view fragment to efficiently maintain both common or uncommon fragments of different views in semi structured data. Due to sharing of materialized fragments, view adaptation can theoretically take place at any point in the view construct. Although our main focus is not to provide a solution to the access controls views, our approach can be easily modified to deal with the XPath security views mentioned above.

3 Introducing Sample Views

Existing research on XML views focused mainly on a subset of XPath expressions, $XP^{\{/,//,[],*\}}$. / and // indicate the parent-child and ancestor-descendant relationships respectively, additionally, they are also the abbreviation of the child and descendant axes defined in XPath query language. [] is a predicate and * is the wildcard that represents all type of nodes. While we support all of these expressions, we also include attribute axis (@), string value comparison and number comparison.

To assist the rest of this discussion, we provide the notation that will be used throughout the paper. An XML document can be modeled as a rooted, ordered and node-labeled tree, $\mathcal{T}$. Let Σ denotes the alphabet of all distinct

Fragment	Name	Description
RF	Root Fragment	represents the starting point in a view model, a single node
FF	Filter Fragment	represents a node sequence after an *select* operation
DF	Dependency Join Fragment	represents a node sequence after an *d-join* operation
SF	Source Fragment	a sequence of *V-typed* within nodes an XML tree
VF	View Fragment	a sequence of nodes indicating the result of a view

Table 1: MFM Fragments

tag names in $\mathcal{T}$. We treat all nodes v_1, v_2, $\cdots$, v_n with same label (*tag-name*) as type $\mathcal{V}$, where $v_i \in \mathcal{T}$ ($0 \leq i \leq n$), $\mathcal{V} \in \Sigma$. v_i is called a $\mathcal{V}$*-typed* node.

The MFM view graph is a combination of XPath views, where common subexpressions of XPath views are displayed once. XPath views are represented by a set of **fragment**s (§4.1) and **operator**s (§4.2) in the MFM view graph.

We now introduce a set of XPath views based on the schema given in *Figure 1*. In *Example 4*, raw values are returned where the `offset` is 1000 and the `key` attribute contains the value `Timer1`. In other words, all Timer1 values with 1000 offset are contained in view $VF1$.

Example 4 (*VF1*)
```
//healthSense/sensorData/sections/sect
ion/measurement[./@offset=1000]//reading
[./key='Timer1']/raw-value
```

`Outliers` are heart rate values that are most likely to be incorrectly determined by the sensors (often impossible heart rate values) and are generally removed before analysis can begin. In *Example 5*, view $VF2$ returns all average time where offsets exceed 8000 milliseconds.

Example 5 (*VF2*)
```
//healthSense/sensorData/sections/sect
ion/measurement[./@offset>8000]//read
ing/average/time
```

In *Example 6*, $VF3$ returns all reading data (outlier, padded, value and average) between specified times (determined by timing offsets).

Example 6 (*VF3*)
```
//healthSense/sensorData/sections/sect
ion/measurement[./@offset>8000][./@off
set<15000]//reading
```

In *Example 7*, $VF4$ contains all `key` and `value` attributes (of `parameter`) where the `name` attribute contains the value `Params`.

Example 7 (*VF4*)
```
//healthSense/sensorData/sections/sect
ion[./@name='Params']/parameter
```

The common parts between queries are listed once in the graph as shown in *Figure 2*, and the intermediate results are represented by the fragments. Fragments in gray represented materialized fragments and are selected manually for the purpose of illustration. The focus of this paper is to provide a fragment-based materialisation and automatic selection forms part of our current work.

4 View Fragments and Operators

In this section, we introduce the fragment set that provides the constructs for the Multiple Fragments Materialization (MFM) View Graph. A number of operators are then used to represent different XPath commands within the MFM graph.

4.1 MFM Fragments

Fragments are categorized into 5 types as illustrated in *Table 1*. 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 for each node label $\mathcal{V}$. In the case of RF and SF fragments, 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 *MFM* view graph. While a view graph will contain multiple query representations, they are all joined by the same root fragment, as shown in *Figure 2* (for example, *RF* with rectangle box).
- `Filter Fragment (FF)` - A filter fragment (e.g., *FF1* in *Figure 2*) 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., *@name='Params'* in *Figure 2* represents the filter operation that results *FF1*.
- `Dependency Join Fragment (DF)` - A Dependency Join Fragment (e.g., *DF1* in *Figure 2*) represents the node sequence resulting from a *d-join* operation described in §4.2, e.g., the $\overrightarrow{\bowtie}$ 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 2* with dashed boxes.
- `View Fragment (VF)` - A view fragment (e.g., *VF1* in *Figure 2*) represents the result of a view. It always follows a deep project operation (§4.2), e.g., Π^D before *VF1* indicates a deep project operation.

Each fragment within a particular view is referenced by a corresponding *VF* fragment that represents the context view. For example, *DF6* is referenced by *VF1*, *VF2* and *VF3* as it is shared by all whereas *DF8* is referenced only by *VF1*. 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.

4.2 MFM Operators

Within our MFM view graph, fragments such as *DF*, *FF*, and *VF* are connected by one of a number of operators (see *Table 2*). We now present a description of these operators and explain why they are necessary.

Before we discuss the operations, we first introduce some notations that used for describing the operations. An XPath expression can be divided into several steps. Generally speaking, each step within an XPath expression contains an *Axis* and a *NameTest*, and it produces a sequence of nodes as an intermediate result to the next step in an XPath expression. A node sequence that results from a step s_i is denoted by $\mathcal{S}^k(\mathcal{V}_i)$, where k indicates the number of nodes in the sequence, $k \leq |\mathcal{S}(\mathcal{V}_i)|$. $\mathcal{S}(\mathcal{V}_i)$ is a sequence of all nodes labeled to $\mathcal{V}_i$. For instance, $\mathcal{S}(reading)$ contains all nodes labeled **reading** within an XML document, whereas, $\mathcal{S}^3(reading)$ indicates a sequence of **reading** nodes, where the size of the sequence is *3*.

Operator	Name	Description	Operands	Operation Type
$\overset{\bowtie}{\rightarrow}$	d-join	perform a dependency join operation between two sequences of nodes	SF and 1 of DF, FF, RF	binary
σ_{pred}	select	perform a select operation over a set of nodes with a specified condition	DF, FF	unary
Π^D	dproject	perform a deep project operation on a node sequence	DF, FF	unary

Table 2: Algebraic Operators

4.2.1 Dependency Join

An XPath expression is represented by a chain of *dependency join*s (*d-join*s), represented by the algebraic operator $\overset{\bowtie}{\rightarrow}$. The output is another sequence of nodes resulting from the axis operation and predicate filtering. In VF2, there are numerous dependency joins, for example, between `healthSense` and `sensorData`, and between `sensorData` and `section`.

Definition 1 (d-join operator)
A d-join operation is a binary operation written as $\mathcal{S}(\mathcal{V}_{i-1}) \overset{\bowtie}{\rightarrow} \mathcal{S}(\mathcal{V}_i)$*. The result of a d-join operation is a sequence of nodes* $\mathcal{S}^k(\mathcal{V}_i)$ *which fulfills the dependency condition (*d-cond*) between* s_{i-1} *and* s_i*, where* $k \leq |\mathcal{S}(\mathcal{V}_i)|$.

The result node sequence is generated using two steps: i) generates a set of *2-tuple* sequences, where each satisfies the dependency condition; ii) projects only nodes ($\mathcal{V}_i$) from tuples within the set. The semantics of a dependency join is listed below:

$$\begin{aligned} \mathcal{S}(\mathcal{V}_{i-1}) \overset{\bowtie}{\rightarrow} \mathcal{S}(\mathcal{V}_i) &= \Pi_m(\{(n,m) |\, n \in \mathcal{S}(\mathcal{V}_{i-1}), m \in \\ &\quad \mathcal{S}(\mathcal{V}_i)), \texttt{REL}(n,m) \rightarrow \textit{d-cond}\} \\ &= \mathcal{S}^k(\mathcal{V}_i), where\ k \leq |\mathcal{S}(\mathcal{V}_i)| \end{aligned}$$

where $\mathcal{S}(\mathcal{V}_i)$ is dependent on $\mathcal{S}(\mathcal{V}_{i-1})$ in terms of *d-cond* and, the method `REL` returns the relationship between two nodes n and m.

4.2.2 Select

A step within an XPath expression may contain an optional set of predicates. Each predicate performs a filtering operation over the context node sequence together with a *selection operation* which selects nodes satisfying the predicate. In VF1, the select operation is `[./key='Timer1']`.

Definition 2 (select operator)
A select operation is an unary operation written as $\sigma_{pred}(\mathcal{S}(\mathcal{V}_i))$ *where pred is a condition of the selection. This operation selects a sequence of nodes in* $\mathcal{S}(\mathcal{V}_i)$ *for which pred holds.*

The result of a *select* operation is a subsequence of the input sequence containing the same typed nodes. The semantics of the operation are:

$$\begin{aligned} &\sigma_{pred}(\mathcal{S}(\mathcal{V}_i)) \\ =\ &\{n \mid n \in \mathcal{S}(\mathcal{V}_i), n\ satisfy\ pred\} \\ =\ &\mathcal{S}^k(\mathcal{V}_i), where\ k \leq |\mathcal{S}(\mathcal{V}_i)| \end{aligned}$$

4.2.3 Deep Project

A Project operation returns a specified set of attributes. When an XPath expression generates its final set of result nodes, it will always return the entire subtree beneath each node. For this reason, we use a *deep project operation* (*dproject*) to project the *entire* subtree content of each node in a node sequence. In VF2, the *dproject* operation returns the value for a single node as with a normal project operation. However, if `averages` was the requested node (see *Figure 1*), the Deep Project operation would return the entire `averages` subtree.

Definition 3 (dproject operator)
A deep project operation is a unary operation written as $\Pi^D(\mathcal{S}(\mathcal{V}_i))$*. This operation projects the entire subtree content of nodes within* $\mathcal{S}(\mathcal{V}_i)$*.*

The result of a *dproject* is a union of subtrees that are rooted at nodes of type $\mathcal{V}_i$. The semantic of a *dproject* operation is listed below. We use the method `SUB` to returns the subtree of a node.

$$\begin{aligned} &\Pi^D(\mathcal{S}(\mathcal{V}_i)) \\ =\ &\Pi_{t_1,\ldots,t_n}(SUB(v)),\ where\ v \in \mathcal{S}(\mathcal{V}_i), t_i \in SUB(v) \\ =\ &\{c \mid \forall c \in SUB(v),\ v \in \mathcal{S}(\mathcal{V}_i)\} \\ =\ &\bigcup_{k=1}^{n} SUB(v_k),\ where\ v_k \in \mathcal{S}(\mathcal{V}_i) \end{aligned}$$

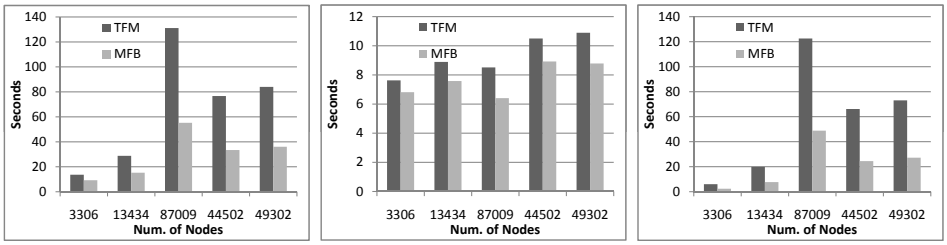
5 Constructing the View Graph

In this section, we first describe a set of rules to be followed during MFM view graph construction. We then provide a detailed description of how the view graph is built using the fragments and operators introduced in §4.

5.1 Construction Rules

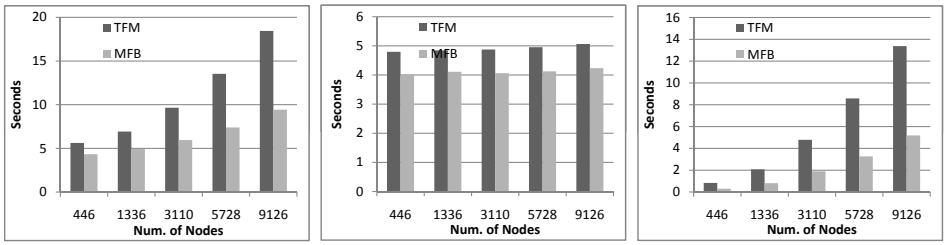
As the MFM graph is constructed from XPath queries, the following is a set of rules used to derive the graph structure.

1. Any number of views may be defined on a database but they must all contain the same root (*RF*) fragment.
2. The *RF* fragment indicates the node sequence containing the document node (the root node of $\mathcal{T}$) and must always be the first fragment in the view model. It can be followed only by a *d-join* operator.
3. A *deep project* operator can never be applied to a *RF* fragment as the result is the entire XML document.
4. A *select* operator can never be applied to a *RF* fragment. This is because applying a predicate to a document node always results in the entire XML document.
5. A *VF* fragment always occurs after an operation node representing the *deep project* operator.
6. A *VF* fragment is always the final fragment in the view model as it contains the final result set.
7. A *SF* fragment always occurs as the right operand of a dependency join operation.



(a) Query Recomputing (b) Network Communication (c) Total Cost

Figure 3: Add DF fragments



(a) Query Recomputing (b) Network Communication (c) Total Cost

Figure 4: Add FF fragments

Assuming the four views introduced in §3 will comprise the view graph, the first step is to define a common root for views. We use the document root of the context XML document, ROOT($\mathcal{T}$), to represent *RF* fragment. By observation, we can see that `healthSense` is the common part to all views in the first step. Therefore, as shown in *Figure 2* `healthSense` is joined with ROOT($\mathcal{T}$) (d-join). `healthSense` is represented by *SF1* containing all instances of `healthSense` nodes, and *DF1* indicates the result node set generated by the dependency join operation between *RF* and *SF1*. `sensorData` (*SF2*) is then joined with the result generated by the first d-join operation (DF1). This then generates *DF2*.

The d-join operation is repeated for all views until `section` (*SF4*) is encountered, which results in *DF4*. At this point, a select operation is performed for VF4 on `name` attribute (`@name='Params'`). A d-join operation is performed on VF1, VF2 and VF3, which joins `section` to `measurement` (*SF6*). As shown in *Figure 2*, this step produces a set of nodes represented by *DF6*.

The d-join and select operations are repeated until the output nodes are located for all views. DF5 represents a set of instances for the output node for VF4. A dproject operation can be performed on DF5, which retrieves all instance of output nodes *together with their subtree content*.

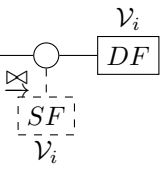
The transformation between XPath expressions and MFM view graph can be summarized by the following transformation rules.

5.2 D-join Transformation

Transformation 1 shows the mapping from a *d-join* operation to our *MFM view graph*.

Transformation 1 (djoin)

$$\mathcal{S}(\mathcal{V}_{i-1}) \overrightarrow{\bowtie} \mathcal{S}(\mathcal{V}_i) = \mathcal{S}^k(\mathcal{V}_i) \quad \Longleftrightarrow$$



The transformation comprises the following steps:

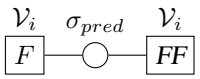
1. $\overrightarrow{\bowtie}$ is transformed into the operation $\texttt{d-join} \Rightarrow \overrightarrow{\bigcirc}$
2. $\mathcal{S}(\mathcal{V}_{i-1})$ is represented by a fragment,
$$\mathcal{S}(\mathcal{V}_{i-1}) \Rightarrow \overset{\mathcal{V}_{i-1}}{\boxed{\texttt{F}}},\ F \in \{VF, DF, FF\}$$
3. $\mathcal{S}(\mathcal{V}_i)$ is represented by a source fragment,
$$\mathcal{S}(\mathcal{V}_i) \Rightarrow \overset{\mathcal{V}_i}{\boxed{\texttt{SF}}}$$
4. $\mathcal{S}^k(\mathcal{V}_i)$ is represented by a dependency join fragment,
$$\mathcal{S}^k(\mathcal{V}_i) \Rightarrow \overset{\mathcal{V}_i}{\boxed{\texttt{DF}}}$$

5.3 Select transformation

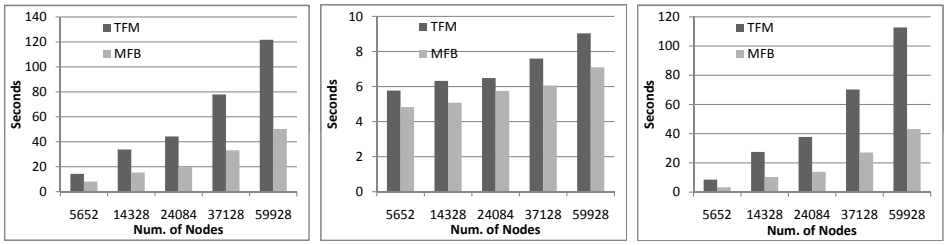
The mapping between a *select operation* and our *MFM* graph is shown in *Transformation 2*.

Transformation 2 (Select)

$$\sigma_{pred}(\mathcal{S}(\mathcal{V}_i)) = \mathcal{S}^k(\mathcal{V}_i) \quad \Longleftrightarrow$$

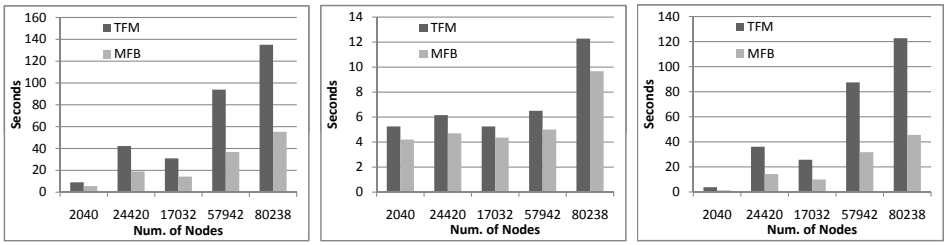


The transformation is achieved through the following steps:



(a) Query Recomputing (b) Network Communication (c) Total Cost

Figure 5: Delete DF fragments



(a) Query Recomputing (b) Network Communication (c) Total Cost

Figure 6: Delete FF fragments

1. σ_{pred} is transformed into the operation $\sigma_{pred} \Rightarrow \overset{\sigma_{pred}}{\bigcirc}$
2. $\mathcal{S}(\mathcal{V}_i)$ is represented by the fragment
$$\mathcal{S}(\mathcal{V}_i) \Rightarrow \overset{\mathcal{V}_i}{\boxed{\texttt{F}}}, \texttt{F} \in \{\texttt{DF}, \texttt{FF}\}$$
3. $\mathcal{S}^k(\mathcal{V}_i)$ is represented by the fragment $\mathcal{S}^k(\mathcal{V}_i) \Rightarrow \overset{\mathcal{V}_i}{\boxed{\texttt{FF}}}$

5.4 dproject transformation

The mapping from the *dproject* operation to our *MFM view graph* is shown below.

Transformation 3 (dproject)

$$\Pi^D(\mathcal{S}(\mathcal{V}_i)) = \bigcup_{k=1}^{n} SUB(v_k)$$
$$\Longleftrightarrow \quad \overset{\mathcal{V}_i}{\boxed{F}} \text{—} \overset{\Pi_D}{\bigcirc} \text{—} \overset{\mathcal{V}_i}{\boxed{VF}}$$

The following steps are achieved to represent a *dproject* operation in our *MFM view graph*:

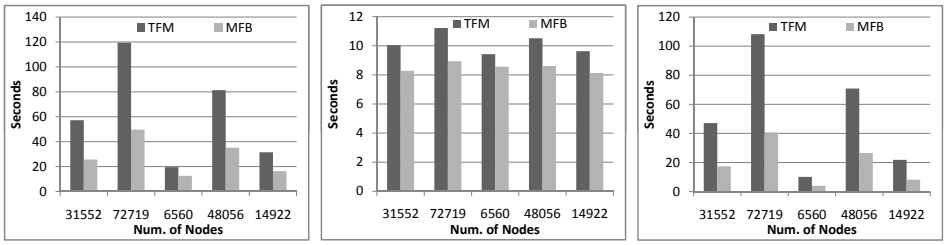
1. Π^D is encapsulated into an operation node, $\Pi^D \Rightarrow \overset{\Pi^D}{\bigcirc}$
2. $\mathcal{S}(\mathcal{V}_i)$ is encapsulated into an fragment,
$$\mathcal{S}(\mathcal{V}_i) \Rightarrow \overset{\mathcal{V}_i}{\boxed{\texttt{F}}}, \texttt{F} \in \{\texttt{DF}, \texttt{FF}\}$$
3. $\bigcup_{k=1}^{n} SUB(v_k)$ is encapsulated into an fragment,
$$\bigcup_{k=1}^{n} SUB(v_k) \Rightarrow \overset{\mathcal{V}_i}{\boxed{\texttt{VF}}}$$

6 Experimental Analysis

Our fragment-based approach is well suited to the XPath view adaptation problem. We now demonstrate the performance of view adaptation using our *fragment-based* materialization approach and the traditional *full* materialization method. The experiment is initialized by materializing XPath views with data obtained from a remote XML server. For the traditional approach, views are required to be recomputed for materialization with new data obtained from this remote server. Our Multi-Fragment approach uses data from materialized fragments currently shared across queries. Clearly there are occasions when our multi-fragment approach may also need to obtain data from the remote server. This happens when query or view adaptation takes place in the steps *before* all materialized fragments. Concerning the query processing cost, the full materialisation approach clearly outperformed the fragment-based approach since some fragments are still virtual (not materialised). However, we show that when considering the global cost, including the query plus view maintenance costs, our fragment-based approach provides superior optimization. Our experiment demonstrates the general costs of adapting existing fragments in response to the view redefinition changes take place to the existing materialised XPath views.

6.1 Logistics

Two servers were deployed for this experiment: a remote (XML database) eXist server and a local eXist server. The remote server contains all XML source data, and the local server stores all materialized views and their fragments. We use eXist version 1.2.5 build 8668 for both remote and local eXist servers. The remote eXist server is distributed on an Intel Core(TM)2 Duo 2.66GHz workstation running Windows XP. The local eXist server is installed on an Intel Core(TM)2 Duo CPU 3.00GHz Windows XP workstation. The second server contains all XPath materialized views



(a) Query Recomputing (b) Network Communication (c) Total Cost

Figure 7: Modify FF fragments

with data obtained from the remote site. An actual dataset collected from heart monitors placed on a various groups of athletes, resulting in 230MB of data, was used as the underlying database.

6.2 Performance

The cost of view adaptation is computed as the total cost of query processing (query recomputing) and network communication between servers (data transformation between servers). In practical terms, we will always see an improvement as each example assumes that the change in view definition will *always* be accommodated in our view graph. This issue is discussed in our conclusions.

6.2.1 Adaptation: Reducing View Sizes

Adding an extra step to the XPath query, with or without predicates, normally requires deletion of some view data. Using the traditional approach, there is no way to determine which part of the materialized data that is now redundant. Therefore, this method requires a complete recomputation of the view and the extraction of data from the remote eXist server. With our multi-fragment approach, adding an extra step with optional set of predicates means that an additional *DF* fragment with optional *FF* fragments are inserted into the *MFM* graph. The multi-fragment approach can achieve rematerialization using the existing materialized fragments shared between different views.

Assume that $FF4$ and $DF10$ have been selected for materialization in *Figure 2*, where $FF4$ is materialized by view $VF3$ and $DF10$ is materialized by view $VF2$. We would like to set a new condition after $FF4$, *name="Data"*, in $VF2$. In this case, $DF10$ must be adapted to fulfill the new condition. $VF2$ must be rematerialized using the traditional approach as the result are restricted by the new condition. However, our approach needs only to rematerialize $DF10$. Rather than retrieve more data from the server, this can be achieved by using the existing materialized fragment $FF4$ in $VF3$.

Improvements in performance of our approach are illustrated in *Figure 3a*. This is in comparison with the full materialization approach(TFM) where entire views are always materialized. *Figures 3a and 4a* show that although query recomputing time is very close for both approaches, the network communication cost increases significantly for the traditional approach when the number of nodes in the sequence is increased (*Figure 3b, 4b*). As shown in *Figure 3c* and *4c*, the multi-fragment approach requires approximately 50% of the time used in the traditional approach.

6.2.2 Adaptation: Increasing View Sizes

Deleting a step from an XPath query expression should result in a request for more data from the database server as the result is less restricted. The traditional approach must obtain the data from the remote eXist server. However, this change does not effect the multi-fragment approach, as deleting a step from a view definition indicates that a *DF* fragment and *FF* fragments are deleted from the *MFM* graph.

Assume that $DF4$ is materialised for view $VF4$, and $FF5$ is materialized for view $VF3$. If we were to delete $FF4$ from $VF3$, this indicates that the result of $VF3$ becomes less restricted as the condition *@offset>8000* is removed. In the traditional approach, it is necessary to recompute the view but in our approach, we rematerialize only $FF5$ by using $DF4$ in $VF4$.

The result is that the multi-fragment approach retrieves additional data from *existing* fragments. As shown in *Figure 5c* and *6c*, reusing existing data from different fragments is significantly faster than the traditional approach. In a worst case scenario, retrieving 59,928 nodes by the traditional approach takes approximately 3 times that of the multi-fragment approach.

6.2.3 Adaptation: Predicate changes

Changing view predicates could result in either more or less data required for the view. In either circumstance, the view must be rematerialized by the traditional approach. This is because only the node sequence produced by the last step is materialized. Therefore, any modification to the previous steps will always lead to view rematerialization. Using our approach, this requires a change to a single FF fragment. The multi-fragment approach can utilize existing materialized fragments to avoid gathering data from remote server as data has already been materialized within the fragments. As shown in *Figure 7*, the multi-fragment approach outperforms the traditional approach for all situations, and more so, for larger node quantities.

Assuming again that only $DF4$ is materialised by view $VF2$, and $FF5$ is materialised by view $VF3$. If we were to change the condition from *@offset<15000* to *@offset<10000* in $VF3$, the new predicate restricts the result of $VF3$. To update the materialized data for $VF3$, the traditional approach must access new data from the database server. Our approach updates the $FF5$ by retrieving the data from $DF4$ in $VF2$.

7 Conclusions

In this paper, we presented a multi-fragment materialization approach that works with semi-structured data. Fragments can be shared by multiple views, which improves the data reusability and reduces duplication involved in the traditional materialization approach. For the research described in this paper, we selected the view fragments for materialization by hand to show those query types that benefit from the multi-fragment approach. Our current focus is on developing a *cost-based view selection* algorithm and *view adaptation* algorithms to work with the multi-fragment platform created here. This will provide full automation of our view materialization system and with further analytical capabilities for scientists working with

the increasing volumes of data generated in today's sensor web.

References

Arion, A., Benzaken, V., Manolescu, I. & Papakonstantinou, Y. (2007), Structured Materialized Views for XML Queries, *in* 'VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases', VLDB Endowment, pp. 87–98.

Ayyagari, P., Mitra, P., Lee, D., Liu, P. & Lee, W.-C. (2007), Incremental adaptation of XPath access control views, *in* 'ASIACCS '07: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security', ACM, New York, NY, USA, pp. 105–116.

Balmin, A., Özcan, F., Beyer, K. S., Cochrane, R. & Pirahesh, H. (2004), A Framework for Using Materialized XPath Views in XML Query Processing, *in* 'VLDB', pp. 60–71.

Bellahsene, Z. (2004), 'View Adaptation In The Fragment-Based Approach', *IEEE Trans. Knowl. Data Eng.* **16**(11), 1441–1455.

Boncz, P. A., Grust, T., van Keulen, M., Manegold, S., Rittinger, J. & Teubner, J. (2006), MonetDB/XQuery: A Fast XQuery Processor Powered By A Relational Engine, *in* 'SIGMOD Conference', ACM, pp. 479–490.

Bruno, N., Koudas, N. & Srivastava, D. (2002), Holistic twig joins: optimal XML pattern matching, *in* 'SIGMOD Conference', ACM, pp. 310–321.

Cautis, B., Deutsch, A. & Onose, N. (2008), XPath Rewriting Using Multiple Views: Achieving Completeness and Efficiency, *in* 'WebDB'.

Damiani, E., de Capitani di Vimercati, S., Paraboschi, S. & Samarati, P. (2000), 'Design And Implementation Of An Access Control Processor For XML Documents', *Comput. Netw.* **33**(1-6), 59–75.

Fan, W., Chan, C.-Y. & Garofalakis, M. (2004), Secure XML Querying With Security Views, *in* 'SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data', ACM, New York, NY, USA, pp. 587–598.

Gao, J., Wang, T. & Yang, D. (2007), MQTree Based Query Rewriting over Multiple XML Views, *in* 'DEXA', pp. 562–571.

Grust, T. (2002), Accelerating XPath Location Steps, *in* 'SIGMOD Conference', ACM, pp. 109–120.

Gupta, A., Mumick, I. S. & Ross, K. A. (1995), Adapting Materialized Views after Redefinitions, *in* 'SIGMOD Conference', pp. 211–222.

Kuper, G., Massacci, F. & Rassadko, N. (2005), Generalized XML Security Views, *in* 'SACMAT '05: Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies', ACM, New York, NY, USA, pp. 77–84.

Lakshmanan, L. V. S., Wang, H. & Zhao, Z. (2006), Answering Tree Pattern Queries Using Views, *in* 'VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases', VLDB Endowment, pp. 571–582.

Lim, C.-H., Park, S. & Son, S. H. (2003), Access Control of XML Documents Considering Update Operations, *in* 'XMLSEC '03: Proceedings of the 2003 ACM Workshop on XML Security', ACM, New York, NY, USA, pp. 49–59.

Marks, G. & Roantree, M. (2009), Metamodel-Based Optimisation of XPath Queries, *in* 'BNCOD', Vol. 5588 of *Lecture Notes in Computer Science*, Springer, pp. 146–157.

Roantree, M., McCann, D. & Moyna, N. (2008), Integrating Sensor Streams in pHealth Networks, *in* '14th International Conference on Parallel and Distributed Systems (ICPADS 2008), December 8-10, 2008, Melbourne, Victoria, Australia', IEEE, pp. 320–327.

Sawires, A., Tatemura, J., Po, O., Agrawal, D. & Candan, K. S. (2005), Incremental Maintenance of Path-Expression Views, *in* 'SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data', ACM, New York, NY, USA, pp. 443–454.

Stoica, A. & Farkas, C. (2002), Secure XML Views, *in* 'DBSec', IFIP Conference Proceedings, Kluwer, pp. 133–146.

Tang, N., Yu, J., Ozsu, M., Choi, B. & Wong, K.-F. (2008), Multiple Materialized View Selection for XPath Query Rewriting, *in* 'ICDE', IEEE, pp. 873–882.

Tang, N., Yu, J. X., Tang, H., Özsu, M. T. & Boncz, P. A. (2009), Materialized View Selection in XML Databases, *in* 'DASFAA', pp. 616–630.