Materialized View Selection in XML Web Query Systems

Issam Al-Azzoni
University of Waterloo
School of Computer Science
ialazzon@uwaterloo.ca

E. Cem Sozgen
University of Waterloo
School of Computer Science
cesozgen@uwaterloo.ca

ABSTRACT

XML has become the new standard for data integration and exchange over the Web. So, more and more Web query systems are being developed for querying XML data over the Web. As an XML query language, XPath has gained a lot of popularity due to its simplicity and expressive power. The performance of a Web query system can be significantly improved by using materialized views. Materialized view selection is a difficult problem that is further complicated by the nature of the Web. As more data is becoming in XML format, addressing the view selection problem becomes even more important for XML Web query systems. In this paper, we present a novel technique for recommending materialized XPath views in XML Web query systems. Our approach involves the use of views with minimal overlapping providing a better utilization of the available storage space in the central query server. We use an efficient heuristic algorithm for choosing the optimal set of views for materialization. The experimental results presented in this paper show significant performance improvements resulting from materializing the views that our algorithm recommends.

1. INTRODUCTION

XML has become a standard for data integration and exchange over the Web. As more and more data on the Web appear in XML format, there is currently a lot of interest in developing systems for querying XML data on the Web. Examples of XML Web query systems include Niagara [16] and Xyleme [19]. Furthermore, several query languages exist for querying XML data. Due to its simplicity and expressive power, XPath has become a very popular XML query language and an increasing number of Web query applications are using XPath to query XML data.

A typical Web query system consists of a central query server and many data sources. Users of the query system submit queries to the central server. Upon receiving the query, the central server parses the query and contacts the relevant data sources. When the central server receives the results from the data sources, it merges these results and then submits the final result back to the user. In an XML Web query system, incoming queries to the central server are XPath queries and the data sources store XML data.

There has been a significant work in improving the performance of Web query systems. One of such work deals with speeding up the query processing by storing some XML data from the data sources on the central server. Thus, when the central server receives a query, it may supply its results using the XML data stored in its cache. One way for storing the XML data in the central server is by using materialized XPath views. An XPath materialized view is a pre-computed XPath query result that is stored (or materialized).

Choosing the appropriate materialized views results in significant performance improvements in Web query systems. However, there are many challenges:

1. There is a limited storage space available in the central server. The size of XML data stored in the data sources is large, so only a small portion of it can be stored in the central server. Thus, it is important to select the materialized views that will lead to the most reduction in the total execution time of the most frequent queries.

2. Several characteristics of the Web make view selection even harder. Obtaining accurate statistics about the performance of the data sources and the communication medium may be difficult. Statistics are used to estimate the execution time of the queries and hence play a major role in evaluating and comparing the candidate materialized views.

3. Several challenges arise due to the nature of XML data and XPath queries. XML data is semi-structured data which may have a time varying schema. The XPath query language is inherently different from the query languages over relational databases. Furthermore, research on query optimization for XML data is a relatively new area. We note that the view selection problem has been studied for relational databases. However, due to the nature of XML data and XPath queries, techniques from relational databases may not apply (or other techniques may perform better) when addressing the view selection problem for XML data.

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

The view selection problem can be stated as follows. Given a workload, *i.e.* a representative set of queries and their frequencies, find a set of materialized views such that using them in the execution of the workload results in a reduction in the total cost. The selected set of materialized views should be optimal *i.e.* it should provide the most reduction in the total cost when compared with other alternatives of materialized views. Finally, the size of the materialized views can not exceed the available space devoted for materialized views in the central query server.

In this paper, we present a new technique for addressing the view selection problem for XML Web query systems. Given a workload consisting of XPath queries and given statistics about the data sources, our algorithm recommends a set of materialized views that provides the most improvement in the total performance of the workload.

To the best of our knowledge, the view selection problem for XML data and XPath queries is not addressed in the literature. This is due to the fact that research in the area of XML query processing and optimization is relatively new. Addressing this problem results in significant performance improvements in XML Web query systems. By materializing the optimal views, the total execution time of the workload reduces. Furthermore, consumption of network and data source resources is reduced. If financial costs are incurred for the consumption of these resources, using materialized views reduces the total financial costs incurred in querying the system. In essence, the algorithm presented in this paper is a novel idea addressing a crucial aspect for improving the performance of XML Web query systems.

The first step of our algorithm identifies the candidate materialized views that seem to be relevant to the workload. Given a workload, we construct a minimal set of views that can answer all the queries in the workload. We use these views as our candidate materialized views. By using a minimal set of views, we reduce duplicate storage of overlapping XML data. Thus, our candidate materialized views achieve a better utilization of the available space in the central server.

After identifying the candidate materialized views, we apply a greedy algorithm for enumerating the view subsets. We assign each candidate view a benefit value based on the estimated total reduction in the execution time of the workload if the view is materialized. Similarly, we assign each view a cost value based on the view size. Then, we model the view selection problem as a Knapsack problem and use a greedy algorithm for selecting the optimal set of views subject to constraints on the available size. We use a simple cost model to estimate the query execution time.

Consider the following motivating example. Assume a Web query system having five data sources: $s1$, $s2$, $s3$, $s4$, and $s5$. The data sources differ in how fast they return the results. A representative workload is found to consist of the following queries, the frequency of each query is shown in brackets: $/s_1/a/*$ (0.1), $/s_1/a/b$ (0.15), $/s_2/a/b[c]$ (0.05), $/s_2/a/b[d]$ (0.12), $/s_3/a/b \cup /s_3/a/c$ (0.2), $/s_3/a/b$ (0.08), $/s_4/a/\overline{\{b\}}$ (0.1), $/s_4/a/\overline{\{c\}}$ (0.07), $/s_5/a/b/c_5$ (0.05), and $/s_5/a/*/c_5$ (0.08). Note that there is a very large number of potential candidate materialized views. Assuming that the central server has an upper bound on the available storage space, our problem is to select an optimal set of materialized views. A sample solution might be $\{/s_1/a/*, /s3/a/b \cup /s_3/a/c, /s_5/a/\overline{\{b\}}/c\}$.

The following are the main contributions of this paper:

1. An efficient heuristic algorithm for recommending materialized XPath views for XML Web query systems.
2. An approach to better utilize the available space in the central server by recommending materialized views with minimal overlapping.
3. A generic cost model which can be easily extended independent of the algorithm.
4. Our experimental results show an order of magnitude reduction in the total execution time of the workload when materializing the views recommended by our algorithm.

The rest of the paper is organized as follows. In Section 2, we give a brief background on the XPath query language and materialized XPath views. Section 3 is the related work section. Section 4 presents a formal definition of the materialized view selection problem along with the assumptions we make. In Section 5, we describe our algorithm for recommending the materialized XPath views. We present the results of a series of experiments in Section 6. Finally, brief conclusions and future work are provided in Section 7.

2. BACKGROUND

In this section, we first give a brief background on XPath query language. Then, we describe the materialized XPath views used in this paper.

XPath [2, 1] is a standard language for querying XML data. Essentially, XPath views an XML document as a tree with node labels. An XPath query specifies a pattern for extracting a set of subtrees rooted by nodes that match the pattern. XPath is only used for querying XML documents and it does not modify the contents of such documents.

In this paper, we use the following syntax for the XPath language. It is borrowed from [17]:
$q ::= /p \mid //p \mid q \cup q \mid q - q$
$p ::= a \mid \overline{\{a_1, \cdots, a_n\}} \mid * \mid p/p \mid p//p \mid p[p] \mid p\overline{[p]}$
A query q is either an absolute location path of the form $/p$ or $//p$, the union of two queries $q \cup q$, or the difference of two queries $q - q$. An absolute location path $/p$ selects nodes reachable from the root node through paths matching a relative location path p. An absolute location path $//p$ selects nodes matching the relative location path p starting from any node.

A relative location path p can be a label test a matching nodes having the label a or a negative label test $\overline{\{a_1, \cdots, a_n\}}$ matching nodes with labels other than $a_1, \cdots, a_n$. p can also be a wild card matching nodes with any label. The relative location path can also be a concatenation of two

```
<s1>
    <A>
        <a> Data 1 </a>
        <b> Data 2 </b>
        <c> Data 3 </c>
    </A>
    <A>
        <a> Data 7 </a>
    </A>
    <B>
        <a> Data 8 </a>
        <b> Data 9 </b>
    </B>
    <C>
        <a> Data 10 </a>
        <b> Data 11 </b>
        <c> Data 12 </c>
    </C>
</s1>
```

Figure 1: A sample XML document

location paths *e.g.* p/p and $p//p$. Finally, a location path $p_1[p_2]$ matches nodes reachable through p_1 and also having at least one path matching p_2 beneath them. On the other hand, $p_1[\overline{p_2}]$ matches nodes reachable through p_1 and having no path matching p_2 beneath them.

The following are several sample XPath queries along with the results of their execution on the XML document shown in Figure 1. Consider the query $//a$ which returns all a nodes no matter where they are in the document. The result of executing this query is "$Data$ 1, $Data$ 7, $Data$ 8, $Data$ 10". Another query is $/s_1/A$ which returns all A nodes that are children of the root node s_1: "$< a > Data\ 1 < /a >$, $< b > Data\ 2 < /b >$, $< c > Data\ 3 < /c >$, $< a > Data\ 7 < /a >$". The query $/*/A[b]/a$ returns a nodes that are children of an A node that is a child of a root node and has at least one child b: "$Data$ 1". Lastly, the query $/s_1/A/\overline{\{b, c\}}$ returns any child of a node matching $/s_1/A$ such that it is not b nor c: "$Data$ 1, $Data$ 7".

We note that the intersection operation $q_1 \cap q_2$ is not included in the definition of the syntax above since it can be computed by $q_2 - (q_1 - q_2)$. Similarly, the complement of q can be computed by $//* - q$. An example of a query using the $\cup$ operator is $/s_1//a \cup //B/*$ which returns "$Data$ 1, $Data$ 7, $Data$ 8, $Data$ 9, $Data$ 10". Another example involving the set difference operator is $/s_1/C/* - /s_1/C/\overline{\{a\}}$ which returns "$Data$ 10."

The next part of this section describes materialized XPath views. As in relational databases, materialized XPath views are used to speed up the processing of queries. A materialized view is a pre-computed query whose results are actually stored (or materialized). Thus, a materialized XPath view stores the result of the XPath view query statement. For instance, a view $/s_1//a$ stores all elements a that are reachable from a root node s_1.

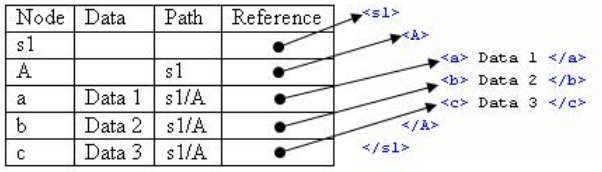

Figure 2: A sample view data structure

To answer queries with a materialized view, the view should store certain information about its nodes. These include node references, full paths, typed data values as well as copied XML fragments [7]. Figure 2 illustrates the contents of a view storing the XML data shown in the figure. Note that node references to the XML data are stored since a query might extract subelements of the view contents. We note that a view definition language for creating materialized XPath views is beyond the scope of this paper.

A materialized view can be used in answering a query when the query result is contained in the view. The problem of deciding whether a query can be answered using a view is referred to as the query containment problem. In [14], the containment problem is shown to be co-NP complete even for a restricted class of XPath queries.

A polynomial-time matching algorithm that is sound and works in most practical cases is proposed in [7] for addressing the XPath query containment problem. To determine whether an XPath query can be answered by a view, the algorithm is used to check whether the query matches the view. If there is a match, then this asserts that the view can be used in answering the query. In this case, the answer to the query needs to be extracted from the view. This is done by creating a compensation expression that returns the answer to the query if applied on the view. Consider, for example, the view $/s_1/A$ and the query $/s_1/A/a$. Using the framework in [7], the query is formally matched to the view and hence the view can be used to answer the query. Applying the compensation step, the compensation expression a can be used to obtain the answer to the query from the view.

3. RELATED WORK

There is currently a lot of interest in developing query systems that query XML data over the Web. Examples of such systems include Niagara [16] and Xyleme [19]. Selecting the appropriate set of views to materialize in a central server is important to improve the performance of Web query systems. This work is a step towards improving XML Web query systems.

There have been several papers addressing the view selection problem in relational databases [5, 6, 9, 10, 15, 20]. However, there are several difficulties for addressing the problem in XML databases. First, XML data has a semi-structured schema which may vary with time. The techniques for relational databases assume constant schema. Second, there does not exist optimizers for XML databases as mature as the optimizers used in relational databases. XML query

optimization is a relatively new area [13, 18]. Thus, any technique recommended for selecting materialized views for XML databases can not rely solely on the existence of efficient optimizers.

The work in [7] presents a matching algorithm that can be used to decide whether an XPath query Q is contained in a given view V. The paper also suggests an algorithm for generating a compensation expression that is applied on the view to extract the solution to a query. Thus, results of [7] address the following questions: "Can a view V be used to answer a query Q", and "if yes, how to extract the solution to Q from V".

Having a solution to such questions has a significant impact on our work. First, our algorithm must recommend materialized views that can be used in answering queries of the workload. Thus, if a query is not matched to a view using a matching algorithm such as the one in [7], then the view should be considered as contributing no benefit at all to the execution of the query. In other words, estimating the benefit of using a materialized view to answer a query depends foremost on the query being contained in the view. Second, our work does not consider partial matching. Thus, if a view can not be used to provide the full answer to a query but can provide a partial answer, we can not use the view at all since there is no algorithm that supports answering queries using partially matched views.

A related problem to view selection is the view minimization which is addressed for XPath queries in [17]. The view minimization problem is stated as follows: "Given a set of queries, construct another set of queries such that the answers to the original queries can be constructed from their answers, and the total size of their answers is minimal". The main goal of view minimization is to minimize the data sent between servers and clients. On the other hand, the main goal of the view selection problem, addressed in this paper, is to choose a set of views that minimizes the cost of answering queries subject to constraints on the storage size of the views.

The first step in our algorithm for recommending materialized views involves finding an equivalent set of queries that can be used in answering the original workload and the intersection of these queries is minimal. This step aims to minimize the redundancy of the views by not storing the same nodes belonging to different queries more than once. Thus, [17] provides an algorithm that can be exploited in our work in solving the view selection problem.

Techniques for obtaining statistics about the data sources are inherently related to our work. Our cost model uses these statistics in the estimation of query execution time. In the literature, several techniques for building statistics for XML data are proposed. Some of these techniques build statistics for static XML data [3, 8]. Static XML data refers to data that are in XML files on the Web. This is different from the "hidden" XML data which sets in databases and can only be accessed by posing queries over these databases. Several techniques are proposed for collecting statistics for the hidden Web [4, 12]. We note that our cost model is very simplistic and we assume static data sources.

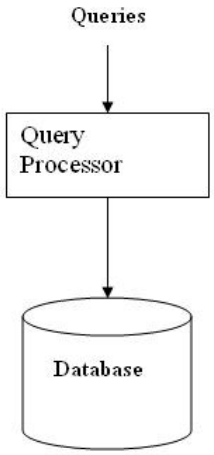

Figure 3: A Central query processing system

To the best of our knowledge, the view selection problem for XPath queries is not addressed in the literature.

4. PROBLEM DEFINITION

Our goal is to suggest a heuristic algorithm that addresses the view selection problem. The algorithm suggests a set of XPath views to be materialized in a central server such that the total execution time for a set of XPath queries is minimized. Several solutions are suggested in relational databases.

To have a better understanding of the complexity of selecting the optimal set of materialized views in Web query systems, we discuss issues relating to a centralized database system containing relational data as shown in Figure 3. The query processor receives queries and executes them using the central database. Given a workload, the corresponding view selection problem is to find the optimal set of materialized views that can be used to expedite the workload query processing. The optimal configuration (materialized views) should respect space constraints i.e. there is a space budget that restricts the size and number of possible materialized views. In typical relational database management systems, the optimizer can be used to estimate the cost of executing a query using a given configuration. In other words, there already exists a cost model that can be used in enumerating the candidate materialized views.

Figure 4 shows an XML Web query system. Here, there are many nodes (sites). We assume that these sites store XML data, or wrappers can be used to return data in XML format. Users can run queries on these sites by accessing a central query processor. Given a workload, our problem is to find the optimal set of materialized views that can be cached and used by the central query processor. Comparing this with a centralized database system (Figure 4), we can not rely on the existence of optimizers in the clients that can be used to estimate the cost of queries. Furthermore, communication costs between the central server and the clients must be taken into account. This suggests that we need to develop our own cost model that can be used to estimate the cost of executing a query with a given set of materialized views.

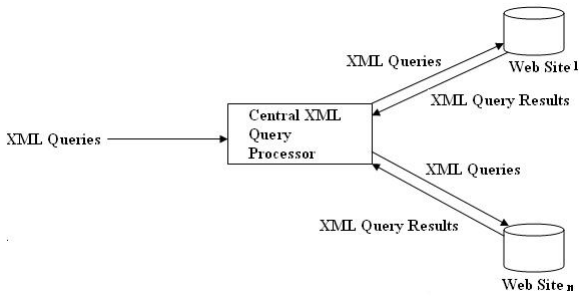

Figure 4: An XML Web query processing system

In this paper, we address the view selection problem for Web query systems involving XPath queries over XML data. In this section, we first provide a formal definition of the view selection problem that we attempt to solve. Then, we discuss several assumptions that we make.

4.1 Formal Definition

A workload in the context of view selection refers to a set of representative queries and their frequencies. The queries in the workload are representative of the typical executed queries in the query system. In our context, these queries are XPath queries that are to be executed on the data sources. Thus, a workload can be characterized as a set of pairs:

$$W = \{(q_i, f_i), i = 1, 2, \ldots, n\}$$

where each q_i is an XPath query and each f_i is its assigned *frequency*.

The view selection problem we address can be stated as follows. Given a workload of XPath queries on XML data stored in the data sources of a Web query system (Figure 4), select a set of views to materialize in the central server $V = \{V_1, \ldots, V_m\}$ such that:

1. The total execution time taken by the central server to process the workload using the materialized views is minimal.
2. The total space occupied by V can not exceed the available space devoted to materialized views in the central server.

More formally, Let $C(q_i, V)$ be the total cost incurred by the central server in executing q_i when the set of views V is materialized. Note that there can be different measures for the cost. For example, the execution time of a query can be used as a cost measure. Another cost measure can be based on other resource consumptions such as storage resources or financial expenses incurred in accessing the data sources. A combination of these can also be used as a cost measure. In this paper, we focus on the execution time as a cost measure and we use the term "cost of a query" to mean the total execution time of the query. Note that cost measures are independent from the view selection problem and algorithms, thus a different cost measure can be substituted without any need to modify the algorithms.

Given a workload $W = \{(q_i, f_i), i = 1, 2, \ldots, n\}$, our objective is to find a set of materialized views V such that:

1. $\sum_{i=1}^{n} f_i C(q_i, V)$ is minimal.
2. $\sum_{v \in V} S_v \leq S$ where S_v is the size of the materialized view v and S is the space limit available in the central server for storing the materialized views.

4.2 Assumptions

In addressing the XPath view selection problem for XML Web query systems, we make the following assumptions:

1. We assume static XML data sources. In other words, the XML data on the sources are not updated. Otherwise, the costs incurred in updating the materialized views in the central server must be taken into account in the objective cost function.
2. Each data source stores its data in an XML file. The root element of the XML file identifies the source. Thus, an XPath query that extracts subelements of a particular source starts with the source id as the root node for location paths. For instance, the query $/s_1//*$ extracts all elements in the XML file of the source s_1.
3. We assume static workload. In other words, our algorithm recommends a set of materialized views that is optimal for the queries in a given workload. However, workload changes happen frequently in a Web query system. Instead of adaptively recommending a new set of materialized views when the workload changes, our algorithm must be re-executed on the new workload. Even if this is the case, the recommended materialized views should still be relatively optimal for minor changes to the workload.
4. The central query processor applies a matching algorithm to decide if a materialized view can be used to answer a query [7]. We assume no partial matching. Thus, a materialized view can not be used at all in answering a query when it holds partial results to the query. In calculating the benefit of using materialized views, a view would provide no benefit at all in answering a query if it does not match the query.
5. Since data sources in a Web query system may not be completely cooperative, we can not rely on the existence of optimizers in the data sources to provide query execution cost estimates. We assume that optimizers present in the data sources do not provide cost estimates and therefore we develop a simple cost model to approximate the query execution time at the data source.

5. HEURISTIC ALGORITHM FOR RECOMMENDING MATERIALIZED XPATH VIEWS

The view selection problem is NP-hard even in the case of no updates [11]. Thus, we focus on developing approximations and heuristics that recommend a set of materialized

views that is as close as possible to the optimal recommendations. This is similar to the query optimization problem in databases where the goal is to find a query execution plan that is as close as possible to the optimal plan, rather than finding the optimal plan which is proven to be difficult. Another result is the avoidance of materialized views that have no benefits at all.

In this section, we propose a heuristic approximate algorithm that addresses the view selection problem. First, we present the architecture of our proposed system. Then, we give a brief overview of the view selection algorithm. Finally, we discuss the algorithm in details.

5.1 Architecture for Materialized View Selection

Figure 5 shows the architecture of our approach for view selection. We assume that the input to the system is a representative workload for which we need to recommend materialized XPath views. The key components of the architecture are: candidate materialized view selection, view set enumeration, and the cost model.

Given a workload, the first step is to identify candidate views. The aim of this step is to prune the large space of possible materialized views in order to consider a smaller subset of worthy views for enumeration. The space of possible views that can be constructed for a workload can be huge. Consider for instance a query $/s_1/A/a$. The following views (among many others) are syntactically relevant: $/s_1$, $/s_1/A$, $s_1/A/a$, $/s1//*$, $/s_1//a$, $/s_1/*/*$, $/s_1/A/\overline{\{b,c\}}$, and $/s_1/A/a \cup /s_2/*$. Other views are not relevant at all *e.g.* $/s_2/*$ and $/s_1/B$. Given the large number of possible views, it is important to eliminate spurious views from consideration early and consider views that are worthy for further consideration.

Once the candidate materialized views are chosen, the next step is to identify the optimal subset of these candidate views. Since the number of candidate materialized views may be large, there can be a very large number of possible view subsets that need to be evaluated. Therefore, naive enumeration of all view subsets may be infeasible. We use a heuristic algorithm for recommending an optimal subset of materialized views without enumerating all passible subsets.

In order to evaluate the benefit and cost of using a materialized view in answering a query, the view set enumeration component uses a cost model for estimating the query execution time. The cost model is used to approximate the query execution time in two cases:

1. A materialized view matches the query and thus is used in the processing of the query.
2. There is no matching view and hence a data source is contacted to execute the query and the results need to be sent to the central server.

Due to the reasons mentioned in Section 4.2, cost model component should not assume the existence of optimizers at the data sources to provide cost estimates.

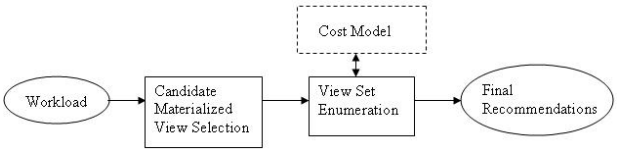

Figure 5: Architecture of the materialized view selection Tool

5.2 An Overview of the Algorithm

Given a workload, we construct a minimal set of views that can answer all queries in the workload. We use these views as the candidate materialized views. The intuition behind using a minimal set of views is to have views that are as disjoint as possible in order to better utilize the available space in the central server by preventing the storage of duplicated XML data.

After identifying the candidate materialized views, we apply a simple greedy algorithm for enumerating the view subsets. We assign each candidate view a benefit value based on the estimated reduction in the execution time of the workload if the view is materialized. Similarly, the cost of using a view is the size of the view. Then, we model the problem as an application of the classic Knapsack problem where the objective is to maximize the benefit of all items in the knapsack.

5.3 Algorithm Details

In this section, we give the details of the three components of our algorithm. The first component is finding candidate materialized views. The second component is the cost model used to evaluate the benefits and costs of the views. The third component deals with enumerating the space of view subsets in order to pick the optimal set of materialized views. Our algorithm recommends this optimal set of materialized views.

5.3.1 Finding Candidate Materialized Views

Figure 6 shows the algorithm for finding candidate materialized views. In Step 1, a minimal view set is computed such that it can answer the original queries of the workload. An algorithm that computes a minimal view set is presented in [17]. The intuition behind using a minimal view set is to have a better utilization of the available space in the central server. The answer sets of the queries of the original workload may have a high degree of overlapping. Since a minimal view set consists of views that are as disjoint as possible, storing these views results in eliminating the repeated storage of the overlapping parts in the answer sets. This saving maximizes the amount of XML data that can be materialized in the central server.

The central query processor should be capable of constructing the answer set for an original query $q \in M$ by using the views in M'. To construct the answer set, the central query processor uses the techniques such as those discussed in [17].

Some of the views that are needed to answer a query $q \in M$ may not be selected in the optimal set, so they may not be

Let $M = \{q_1, \ldots, q_n\}$ be the queries of the workload.
Let $freq(q_i)$ be the frequency of the query q_i as specified in the workload.

1. Transform M into $M' = \{q'_1, \ldots, q'_m\}$ such that M' is a minimal view set that can answer all queries in M.

2. Assign the frequency values for the views in M' using the following rule: "if $q_i \in M$ requires the merging of l views in M', then add $freq(q_i)/\ l$ to the frequency value of each merged view".

3. Use M' as the set of candidate materialized views

Figure 6: Algorithm for finding candidate materialized views

materialized. In this case, partial results to the query can be computed using the materialized views available while the rest of the results are obtained from the data sources. It is important to distinguish this from partial matching.

Partial matching is not assumed by our algorithm. Thus, if a query does not belong to the workload, then one of the following two cases results when executing the query in the central query server:

1. One of the materialized views matches the query. This is the case when the query result is contained in the view. The containment decision is made by the central query optimizer which uses a matching algorithm such as the algorithm in [7].

2. There are no matching materialized views. In this case, the query results have to be obtained by contacting the data sources and shipping back the results. Thus, even if a view contains partial results to the query, the data sources still have to supply the full results to the query since the central query optimizer does not support partial matching.

Thus, a view may partially benefit a query that is in the workload since the central query processor knows the transformation rules used in constructing the minimal view set from the original workload. However, for queries not in the workload, the query processor may not be able to apply any transformation rule and a view can not benefit the query at all when the view does not match the query.

Step 2 of the algorithm (Figure 6), aims to assign frequencies to the views $q' \in M'$. The new frequencies must preserve the original frequency distribution. Thus, for example, assume that q_1 and q_2 are the only two queries in the workload and that their frequencies are f_1 and f_2, respectively. Also, assume that obtaining the results for q_1 requires the merging of q'_1 and q'_2 where $q'_1, q'_2 \in M'$. Similarly, obtaining the results for q_2 requires the merging of q'_2 and q'_3 where $q'_2, q'_3 \in M'$. Then we assign frequencies as follows: q'_1 is assigned $f_1/2$, q'_2 is assigned $f_1/2 + f_2/2$, and q'_3 is assigned $f_2/2$.

5.3.2 Cost Model

We use a simple cost model to estimate the query execution time. The execution time of a query can vary depending on whether there exists a matching materialized view in the central server or not. In the case that there is a matching materialized view in the central server, the execution time can be obtained by invoking the query optimizer of the central server. In the other case when there is no matching materialized view in the central server, the following equation is used to estimate the query execution time:

Total query execution time = the query execution time at the data sources + the time for transmitting the results back to the central server + the query execution time at the central server

We estimate the query execution time at a data source using the following formula:

The query execution time at the source = (the size of the query results at the source [in bytes]) / (the source's estimated query processing rate [in bytes per second])

Similarly, we estimate the transmission time required to send the query results from a data source to the central server using the following formula:

The time of transmitting the results back to the central server = the size of the query results at the source [in bytes]) / (the source's estimated transmission rate [in bytes per second])

5.3.3 Workload Optimization

After determining candidate materialized views, this step aims to find the optimal set of these views to be recommended for materialization.

We model the problem of selecting materialized views as a Knapsack problem and use a greedy algorithm for solving it. In the Knapsack problem, each object has an associated "benefit" and "cost", and the objective is to include objects in the knapsack such that: (1) the sum of the "benefit" of the included objects is maximized, and (2) the sum of the "cost" of the included objects does not exceed a specified threshold (the size of the knapsack).

In modelling the materialized view selection problem as a knapsack problem, the objects to be included are the candidate views suggested in the first step. The benefit for a view is the improvement in the estimated execution time of all queries in the workload that would result when the view is materialized. The cost value assigned to a materialized view is the size of the view, in bytes. The size of the knapsack is the maximum space available for the materialized views in the central server. Thus, the objective function is to maximize the benefit of all views in the knapsack subject to the constraint on the total available space.

The following defines more formally the benefit of a materialized view. The benefit of a materialized view is the sum

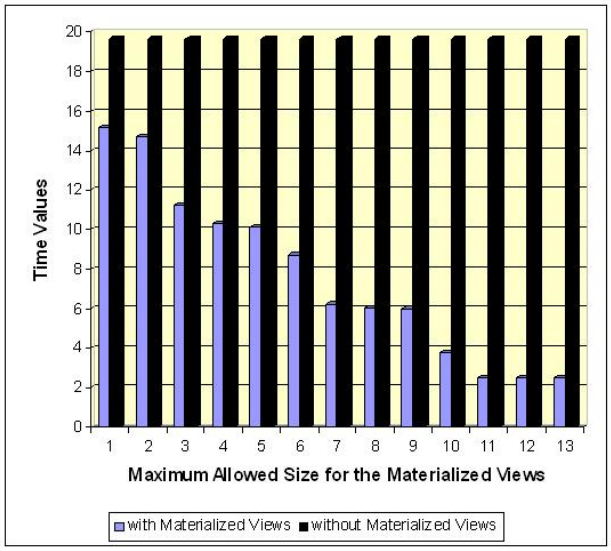

Figure 7: Elapsed time of executing the workload with and without materialized views

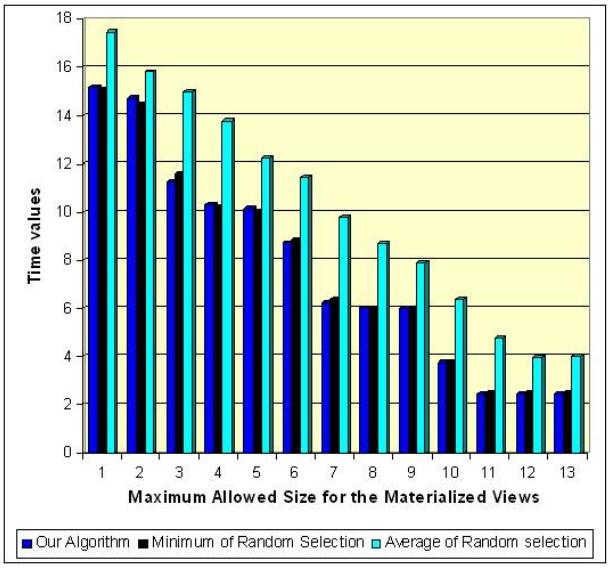

Figure 8: Comparison of our greedy approach with random selection among candidates

of the benefits of using the view in answering all queries in the workload. The benefit of using a materialized view V in answering a query Q is defined as follows:

$$\begin{cases} 0 & \text{if } V \text{ does not match } Q, \\ net\_execution\_time(V, Q) & \text{otherwise.} \end{cases}$$

The function $net\_execution\_time(V, Q)$ is defined as follows:

(The total execution time of Q without using a materialized view − the total execution time of Q using the view V) = (The execution time of Q at the data sources + the time for transmitting the results of Q back to the central server − the execution time of Q at the central server using V)

Note that the execution time of Q at the central server using the materialized view V can be obtained by invoking the central server's optimizer.

After assigning benefit and cost values to each candidate materialized view, a greedy solution to the Knapsack problem can be obtained as follows. At each stage, select the view which has the maximum benefit-to-cost ratio and keep accepting views until the knapsack is full.

6. EXPERIMENTAL RESULTS

We designed the experiments with the aim of showing:

- The benefit of using materialized views to answer queries.
- The effectiveness of our greedy algorithm in finding the optimal set of materialized views.
- The advantage of applying transformations suggested by [17] to the original workload in order to minimize the overlapping between the materialized views.

6.1 Experimental setup

We used Microsoft Visual Studio .net as our programming environment because of its extensive XML and XPath support. We used C♯ as our programming language.

With the consideration of the time limit for this project, we used relatively small data sources and workload as compared to a real world scenario. So the experiments are illustrative rather than exhaustive.

6.2 Benefit of Materialized Views

The goal of this experiment is to show the decrease in the elapsed time of executing the workload with the materialized views. The results are as expected and shown in Figure 7. As we increase the storage limit for the materialized views, the elapsed time of answering queries decreases. The reason is that as more materialized views are stored in our central server, more queries can be answered by using these materialized views reducing the need for going to the data source where the data originally sit.

6.3 Effectiveness of Our Greedy Approach

Remember that we take the original workload with the frequency information of the queries and apply the transformations suggested by [17]. Then we have our candidate materialized views. The goal of this set of experiments is to evaluate the effectiveness of our greedy algorithm in selecting the optimal set of materialized views.

We evaluate the benefit of each candidate materialized view by considering:

- The frequency of the queries for which this candidate can be used.
- The size of the results of the queries for which this candidate can be used.

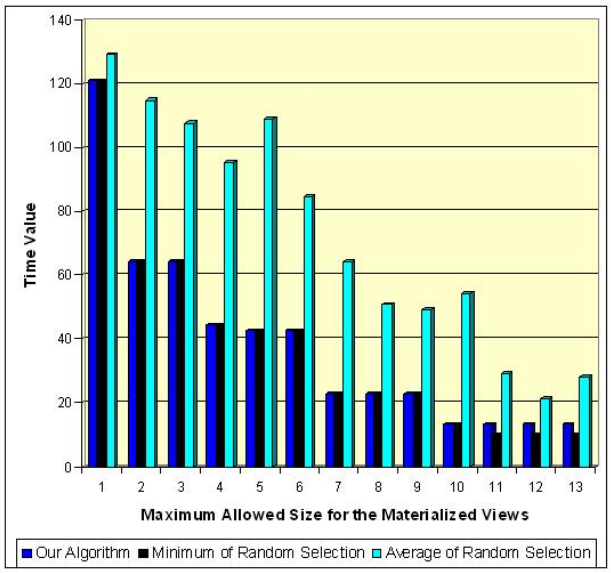

Figure 9: Comparison of our greedy approach with random selection among candidates

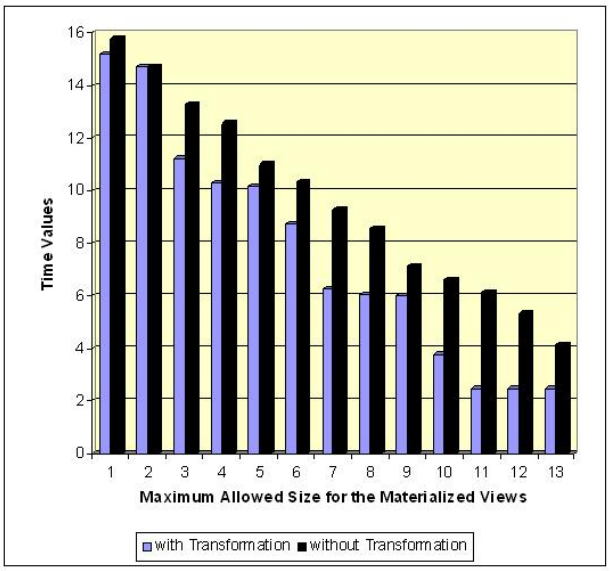

Figure 10: Elapsed time with and without transformations

- The different characteristics of the data sources (We assume that we are aware of the characteristics of the data sources such as communication time with the sources, query processing time of the sources, cost of using the network and accessing the sources).

We determine the sizes of the candidate views by directly querying the sources. Actually, we tried to develop some techniques for the estimation of the sizes of the candidates by using the original workload, but then our greedy algorithm generally finds sub-optimal solutions due to the imprecise estimations.

By using the benefit and size of each candidate materialized view, we find an optimal set of candidates with a greedy approach. Figure 8 is a comparison of our algorithm with the random selection among these candidates. What we mean with random selection is that we arbitrarily select some set of candidates among the available candidates. We repeat this random selection process for a number of times. In the figure, averages of these are shown. In the figure, also the minimums among these are shown. As can be seen, for most of the cases, our algorithm finds the optimal set.

Figure 9 shows the results for the same workload and data sources but with different data source characteristics. In this case, the random selection gives worse results than the previous case.

6.4 Advantage of Transformations

We apply the transformations suggested by [17] in order to minimize the overlapping of data between the candidate materialized views. Thereby, we avoid storing the same data more than once, and provide space for storing more materialized views. Figures 10 and 11 justify our claim. Figure 10 compares our algorithm with the one which our cost model and greedy approach are directly applied on the original workload (without applying the transformations). Figure 11 compares our algorithm with the one which random selection is done on the original workload (without applying the transformations). We run the random selection algorithm for a number of times and the results are the minimums and averages of these. There are significant differences between our approach and the other approaches which show the value of applying these transformations.

Figure 12 shows the results with another workload which have more intersecting queries (with intersecting queries, of course we mean the results of the queries are intersecting). As can be seen, as the intersections between the queries increase, the value of transformations also increases.

6.5 Discussion

The experiments justify our two important contributions by showing:

- The value of the transformations.
- The effectiveness in selecting the optimal set of materialized views.

As shown, transformations make the workload run faster by providing space for more materialized views. As the intersections between the queries in the workload increase, the benefit of applying the transformations increases. Also, our greedy algorithm, with our cost model, is able to find the optimal set of materialized views for nearly all of the cases.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we present an algorithm for recommending materialized XPath views for XML Web query systems. The

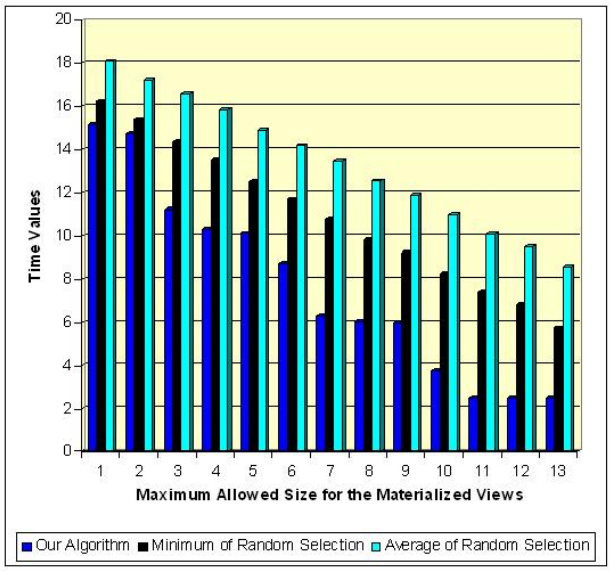

Figure 11: Elapsed time with and without transformations

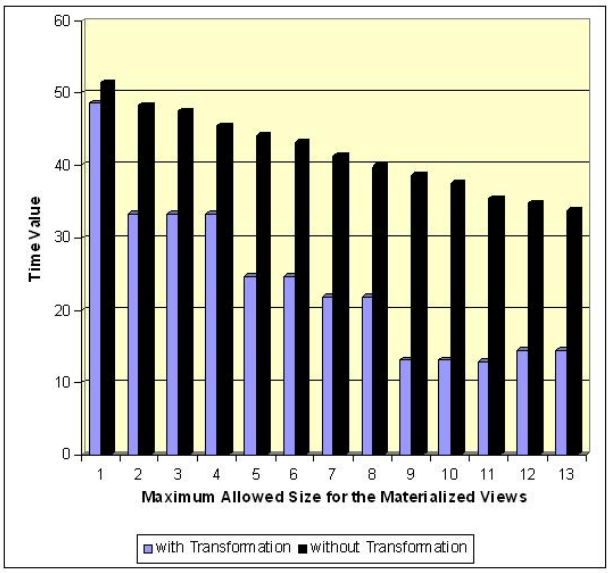

Figure 12: Elapsed time with and without transformations

algorithm selects views that have minimum overlapping providing for a better utilization of the available space in the central server. We use a simple greedy algorithm for enumerating the candidate views and choosing the optimal view set. Our experiments show that materialized views recommended by our algorithm outperform other alternatives.

In the future, we plan to extend our work to handle more complexities faced in XML Web query systems. Mainly, we aim to address the view selection problem under less-restricted assumptions. These include the following:

1. Adding support for dynamic data sources where updates are possible. In this case, costs incurred in updating the materialized views must also be taken into account.

2. Adding support for dynamic workloads. Thus, when the workload changes, the algorithm should adaptively recommend a new set of materialized views.

3. Supporting more complex cost models. One aspect of a cost model relates to the accuracy of statistics on the data sources and the communication network. A future work should take the fuzziness of these statistics over the Web into account. Another aspect of the cost model is the cooperation of data sources. As optimizers for XML databases become more mature, query execution time at the data sources can be obtained by invoking their optimizers.

8. REFERENCES

[1] XML Path Language (XPath) Version 2.0-W3C Working Draft, April 2005. http://www.w3.org/TR/xpath20/.

[2] XML Path Language (XPath) Version 1.0-W3C Recommendation, November 1999. http://www.w3.org/TR/xpath.

[3] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 591–600, 2001.

[4] A. Aboulnaga and J. F. Naughton. Building XML statistics for the hidden web. In *Proceedings of the 12th International Conference on Information and Knowledge Management*, pages 358–365, 2003.

[5] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 496–505, 2000.

[6] N. Ashish, C. A. Knoblock, and C. Shahabi. Selectively materializing data in mediators by analyzing user queries. In *Proceedings of the 4th International Conference on Cooperative Information Systems*, pages 256–266, 1999.

[7] A. Balmin, F. Ozcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 60–71, 2004.

[8] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: making XML count. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 181–191, 2002.

[9] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the 6th International Conference on Database Theory*, pages 98–112, 1997.

[10] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proceeding of the 7th International Conference on Database Theory*, pages 453–470, 1999.

[11] H. Karloff and M. Mihail. On the complexity of the view-selection problem. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 167–173, 1999.

[12] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. XPathLearner: An on-line self-tuning Markov histogram for XML path selectivity estimation. In *Proceedings of the 28th Conference on Very Large Data Bases*, pages 442–453, 2002.

[13] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 315–326, 1999.

[14] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proceedings of the 21st Symposium on Principles of Database Systems*, pages 65–76, 2002.

[15] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.

[16] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[17] K. Tajima and Y. Fukui. Answering XPath queries over networks by sending minimal views. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 48–59, 2004.

[18] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of the 19th International Conference on Data Engineering*, pages 443–454, 2003.

[19] L. Xyleme. A dynamic warehouse for XML data of the Web. *IEEE Data Engineering Bulletin*, 24(2):40–47, 2001.

[20] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 136–145, 1997.