Parameterized XPath Views

Timo Böhme and Erhard Rahm

Database Group
University of Leipzig
{boehme,rahm}@informatik.uni-leipzig.de

Abstract. We present a new approach for accelerating the execution of XPath expressions using parameterized materialized XPath views (PXV). While the approach is generic we show how it can be utilized in an XML extension for relational database systems. Furthermore we discuss an algorithm for automatically determining the best PXV candidates to materialize based on a given workload. We evaluate our approach and show the superiority of our cost based algorithm for determining PXV candidates over frequent pattern based algorithms.

1 Introduction

With XML as the lingua franca for data exchange and an increasingly popular storage format for structured data there is a growing demand for natively storing and querying of XML. Consequently native XML database systems evolve and relational database systems have been augmented with XML support. Query optimization is a main challenge for these systems due to the high flexibility and ordered structure of XML and the complexity of its query languages.

XPath is a crucial component of XML query languages such as XQuery or XSLT and thus has been an essential part for improving query performance. Work on this topic ranges from indexing techniques [15, 21, 22], structural join algorithms [3, 12], containment, equivalence and intersection of XPath expressions [10, 13] to cardinality estimation [20, 25]. Despite the large amount of work on XPath processing, running complex queries on large XML data sets is still a challenge. Moreover, several of the proposed algorithms are not applicable in certain environments like implementations using a relational database system (RDBS) back-end for storing the XML tree structure.

It was shown that caching techniques [4, 14] and materialized views [1, 18, 23] could be used to address these performance problems. However we found that the proposed solutions were not flexible enough to adapt to specific workloads. We therefore propose to enhance the materialized view approach in two directions. First, we parameterize the view definition in order to use materialized views for queries with different comparison values. Second, our views contain extra information to efficiently use them as a replacement for query fragments which do not start at the query root.

With the enhanced flexibility of our views a manual selection of the most profitable views to materialize for a given workload, database and space constraint is not feasible. We therefore developed a method to automate this important decision

R. Cooper and J. Kennedy (Eds.): BNCOD 2007, LNCS 4587, pp. 125–137, 2007.

process and show how this can be implemented in an XML extension for RDBS called XMLRDB.

The rest of the paper is organized as follows. Next we discuss related work. Section 0 details our enhancements to materialized views called PXV. The integration of PXVs in XMLRDB is described in Section 4. In Section 5 we present our method for automatically determining the most valuable view candidates to materialize. Section 6 evaluates experimentally the performance gains obtained by employing PXVs. Finally Section 7 concludes the paper.

2 Related Work

Grust et al. proposed efficient implementations for XPath [7] and XQuery [8] based on a RDBS with a generic storage of XML data. The most efficient variant utilized a specific numbering scheme as well as a special join operator. A general problem of RDBS usage is that they need many expensive join operations for complex XPath expressions (see Section 0). This also holds for the related work on XQuery-to-SQL translation. The optimizations proposed in the present paper can complement these previous approaches.

A general framework for materialized XPath views is described in [1]. The views may contain XML fragments, typed data values, references to nodes in the actual data and full paths. The paper covers the XPath query rewriting process. Our work differs from this paper in the following points. We enhanced the view concept by parameterizing comparison values and added information for simplified view application. Furthermore we propose an algorithm to automatically determine valuable views to materialize.

[14] creates materialized views on the fly for query caching. The sampled workload is parameterized on comparison values. Each view stores its data as an XML fragment. Only the information which parameter values were used to build the fragment are kept. Therefore if comparison predicates are used in a query which should be answered by a view, the view must be pruned by a compensation query. Since the view only contains the result of its defining query q_v it is only possible to restrict on predicates of the last step in q_v because the corresponding node is the root node of the stored XML fragment.

Materialized XML views are used to improve performance of an XML interface of a RDBS in [18]. Instead of translating each query to SQL and transforming the relational result to XML it caches frequently accessed data as materialized XML views. This approach differs greatly from ours, as it is based on relational data whereas we depend on XML node based storage.

Query rewriting using views has been extensively discussed for RDBS [9]. Later this problem was studied for semistructured data [6, 17] and recently it was examined for the XML domain with the specialities of the XML data model and XML query languages. [11, 23] focus on subsets of XPath for polynomial time algorithms. [16] covers query rewriting using XQuery based views. In our approach we focus on a query rewriting to find an identical match (cf. Fig. 1) of the view definition within the query which can be achieved in $O(\#_{steps}(q) \cdot \#_{steps}(q_v))$ time complexity.

Finding frequent XML query patterns as candidates for caching or materialization is targeted in [24]. The proposed algorithm FastXMiner finds frequent query patterns

of a set of XPath queries. Its limitations however are that it does not support predicates and that the root of a query pattern has to be the root of a query. So it cannot find frequent patterns starting either at the second or a later step of an XPath query or within predicates. FastXMiner as well as other work on mining frequent query patterns from trees [5] consider only workload data but do not rank the patterns according to real or estimated query costs. [14] considers only complete XPath queries from a given workload as view candidates. For each query a template is created by parameterizing all constants. Queries with the same template and templates which contain each other are grouped together.

3 Parameterized XPath Views

We first define some terms to be used in the sequel. A *node* is short for XML node and describes an XML element, attribute or another XML node type. It is part of an XML document stored in the database and is the smallest unit which can be accessed. A *node reference* is a link pointing to a node within the database. Typically a node reference will be a value which uniquely identifies a node. Modelling an XML document as a tree of nodes $t_d\langle V_d,E_d,r_d\rangle$ with V_d the set of nodes, E_d the edges between the nodes and $r_d \in V_d$ the document element, we can define an *XML fragment* as a subtree $t_f\langle V_f,E_f,r_f\rangle$ with $r_f \in V_d$ and $V_f \subseteq V_d$ the descendant nodes of r_f and $E_f \subseteq E_d$ the edges between nodes of V_f.

We will first describe materialized XPath views as found in the literature. Afterwards we discuss two limitations of them and our solution. Materialized XPath views contain precomputed query results and can thus be used to quickly answer queries without the need to query the actual data. A materialized view v can be described by $v\langle q_v, R_v\rangle$ with q_v being the query the view represents, and R_v the query result. A view v can be used to answer query q if the result of q can be obtained by executing a so-called compensation query c on R_v that is $q=c\circ q_v$. Following [1] R_v may contain XML fragments, typed data values, node references or a combination thereof. If it contains XML fragments the compensation query has to be based on the fragment data since only the result nodes and their descendants can be accessed. When only typed data values are stored in the view no compensation (besides restricting the value range) is possible. Storing node references in R_v represents the most flexible variant for compensation and postprocessing the results. Here R_v can be seen as the set of context nodes to run a further XPath expression on.

Examining this standard view concept we found two deficiencies which limit the envisioned flexible applicability of views. To overcome these limitations we propose two enhancements which will be described in the following: support for parameterized comparison values and support for inner query fragments.

3.1 Parameterized Views

The standard definition of view queries assumes fixed specification of XPath expressions. This makes it difficult to efficiently support queries with comparison predicates like */world/country[@name='Germany']/history/entry*. For such a query

we would like to utilize a view of history entries of all countries. It would be possible to define a view */world/country/history/entry* which contains node references and rewrite the query to use a compensation *ancestor::country[@name='Germany']*. However this postprocessing, especially for queries with more complex predicates, largely reduces the utility of the materialized view.

To overcome this problem we propose the use of an enhanced view definition supporting parameterized view queries. The main focus is on equality expressions since they occur very frequently. For example, in our workloads we observe common patterns of queries which only differ in a constant value like in

/world/country[@name='Germany']/history/entry
and */world/country[@name='France']/history/entry*.

To generalize a view we allow constants within predicates containing only an equality expression to be replaced by a parameter. So in our example we would define the view query as */world/country[@name=$1]/history/entry*.

The materialized view contains for each result node reference all parameter assignments yielding this node. Since the number of possible assignments per node could become quite large the constants to be replaced by parameters should be selected carefully.

Parameterizing of constants in view definition queries was also proposed in [14]. Unlike our approach the parameters can only take values from a fixed set taken from the workload. Furthermore the parameter assignments are only used when materializing the view. Query rewritings cannot use the parameters to restrict the view result.

3.2 Support for Inner Query Fragments

The standard views are primarily tuned for queries which exhibit a similar query prefix as the view query. Otherwise the compensation operations, if possible, would be quite costly. To extend the applicability of materialized views we also want to utilize a materialization of "inner" query fragments which occur after a certain query prefix[1]. This extension is motivated by the observation that different complex queries often use the same inner fragments for different query prefixes. Hence optimizing the execution of such fragments by materialized views is likely to be very effective as it can reduce the number of query steps to be processed. Such a step reduction will improve query performance especially in systems with a relational backend like ours (cf. Section 0) where each step results in an extra join operation.

Example Consider the following two queries

/world/country/history/entry[@year=1990]/text()
//town[@name='Leipzig']/../history/entry[@year=1990]/author

Both share the inner fragment *history/entry[@year=1990]* which could be materialized as *//history/entry[@year=1990]*. Using this view would need a possibly costly

[1] We define as an XPath query fragment each continuous sequence of steps from the query. Even steps within a predicate can make up a query fragment.

compensation check for each view entry for both queries. [1] proposes to remedy this problem by storing full paths[2] in the view with each node reference, so that no data has to be accessed for the prefix check. However this will not work in cases like query two since the full path can only be used to test for ancestor element nodes.

To support materialization of inner fragments views where the definition starts with '//' are extended by storing the references of the starting nodes, i.e. the nodes identified by the first step after "//". To restrict the number of possible starting node references the first step should have a name test. In the above example with the view definition *//history/entry[@year=1990],* all *history* elements within the stored XML document are starting nodes. The materialized view not only contains references to the result nodes, i.e. *entry* elements which have *history* elements as parent and a *year* attribute with value '1990' but also a reference to the *history* parent for each of these *entry* result references. Depending on the view definition a result node may have several starting nodes.

$$
\begin{aligned}
q &= /A_1/../A_t/b_1[p_{1,1}]..[p_{1,n}][p_{1,n+1}]..[p_{1,m}]/B_2/../B_p/b_q[p_{q,1}]..[p_{q,r}][p_{q,r+1}]..[p_{q,s}]/C_1/../C_u \\
q_v &= //b_1[p_{1,1}]..[p_{1,n}]/B_2/../B_p/b_q[p_{q,1}]..[p_{q,r}] \\
q_{rw} &= /A_1/../A_t/b_1[p_{1,n+1}]..[p_{1,m}]/v_{q_v}[p_{q,r+1}]..[p_{q,s}]/C_1/../C_u.
\end{aligned}
$$

Fig. 1. XPath query rewriting using PXV with q – source query, q_v – query defining view, q_{rw} – rewritten query using view v_{q_v}

The query *//town[@name='Leipzig']/../history/entry[@year=1990]/author* can now be answered using the materialized view. Its result nodes are selected by the constraint that starting nodes must be contained in the set defined by *//town[@name='Leipzig']/../history*. So we can now treat the view as a special XPath step which replaces a fragment within an XPath query. It takes the context nodes from the previous step, generates the intersection with its starting node references and produces a new set of context nodes from its result node references.

We can define this process more formally as shown in Fig. 1. A_x, B_y, C_z are complete steps comprised of axis, node test and predicates and b_1, b_q are complete steps without predicates. It is depicted that a query fragment q_f can be replaced by a view v if q_f and q_v have identical steps whereas q_f may have further predicates in its first and last step. Steps are identical if they either exactly match or the step from the view definition contains a parameter whereas the other step has a constant. Furthermore the sequence of predicates may be different between two identical steps if no positional predicates are involved.

The application of our view concept which we call PXV (parameterized XPath views) is described in the next two sections. First we show its implementation in an XML extension for RDBS. This is followed by a proposal to automatically determine a reasonable set of PXVs based on a given workload.

[2] A *full path* is the sequence of element nodes from the document element to the actual view result.

4 Implementation of PXV

We have implemented PXVs in an XML database system named XMLRDB. We developed this system as an XML extension for RDBS to evaluate schema-independent and document-centric XML processing. XMLRDB stores XML documents generically in a RDBS and translates XPath expressions into SQL. This translation leads to complex SQL statements with a join operation for each XPath step and subexpression. However, this results in performance problems since even relational optimizers of commercial DBMS reach their limit with queries containing many join operations (>10). Hence reducing the number of joins is key to good query performance. Since most proposed XPath processing algorithms depend on fast navigations within the XML tree they are not an option for this kind of system where each navigational step has to be translated and executed as an SQL query. Path oriented index structures are of limited use as well for XPath expressions with predicates. Hence we mainly rely on PXVs to materialize hot spots in our query workload and thus reduce the number of joins. We first briefly introduce XMLRDB and discuss PXV implementation later on.

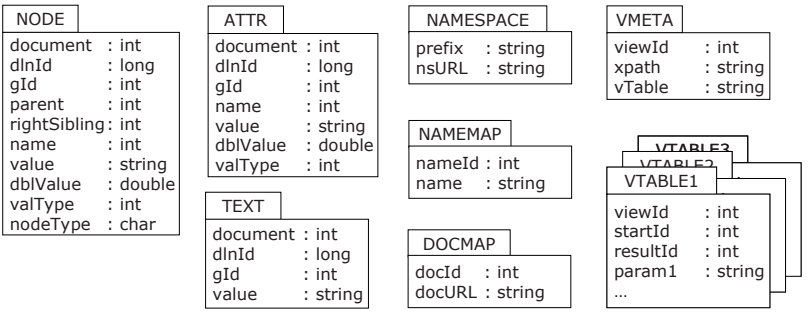

Fig. 2. Relational schema of XMLRDB with PXV tables

4.1 XMLRDB

With XMLRDB we want to evaluate how existing relational database systems without vendor specific XML extensions can be used for XML data processing. We therefore developed an XML layer which transforms XML documents and queries into their relational counterparts and vice versa. The XPath query transformation module employs multiple stages for query optimization like XPath transformation or query rewriting using XML index and views which are managed by the XML layer. We use a generic structure oriented mapping to transform the XML tree structure into predefined relations (see Fig. 2). This kind of mapping was best suited to meet the goals: (1) independence of application-specific XML schemas, (2) support for all kinds of XML documents especially document-centric ones and (3) efficient insert, update and query operations using XML interfaces. In order to support efficient query operations

we use an improved version of the node labelling scheme DLN introduced in [2]. It supports efficient evaluation of XPath axes and allows for fast document reconstruction and insert operations.

```
SELECT DISTINCT x3.docId, x3.dlnId
FROM node x1, attr a1, node x2, node x3
WHERE   x1.name='country' AND
        a1.name='name' AND
        a1.gid=x1.gid AND
        a1.value='Germany' AND
        x2.name='history' AND
        x2.parent=x1.gid AND
        x3.name='entry' AND
        x3.parent=x2.gid
ORDER BY x3.doc,x3.dlnId
```

Fig. 3. Generated SQL for *//country[@name='Germany']/history/entry*

```
INSERT INTO vtable1
SELECT DISTINCT x1.gid, a1.value, x3.gid
FROM node x1, attr a1, node x2, node x3
WHERE   x1.name='country' AND
        a1.name='name' AND
        a1.gid=x1.gid AND
        x2.name='history' AND
        x2.parent=x1.gid AND
        x3.name='entry' AND
        x3.parent=x2.gid
```

Fig. 4. Generated SQL to materialize *//country[@name=$p1]/history/entry*

4.2 Integration of PXVs in XMLRDB

Making XMLRDB PXV-aware we had to implement a PXV management component and to enhance the XML query processor. The management component stores the materialized views and their metadata within the relational database and uses them during query translation. Table *vmeta* contains the view id, the parameterized XPath view definition and the materialization table names. While it is possible to materialize views with the same number of parameters within the same table it is not advisable. The reason is that views will have different ratios of starting nodes and result nodes. Thus the sampling algorithms of the relational optimizers to gather statistical data typically yield a wrong picture of the distribution of a specific view.

When a view has to be materialized by the management component it can reuse the standard XPath-to-SQL transformation component (XtoS) with only small changes. For an XPath expression, XtoS creates a single SQL query with join operations for each step, even for nested predicates. To generate the view data we only have to specify additional returning node ids from the first step and parameter values. This is illustrated in Fig. 3 and Fig. 4. Fig. 3 shows the generated SQL for a standard XPath query returning node references using the DLN labeling scheme. For enhanced readability we provide real element and attribute names instead of ids here. Fig. 4 shows how the same query is translated for view materialization with the constant value replaced by a parameter. It was generated by the same XtoS component changing only the select clause to return database-wide unique node ids for starting node, result node and value assignments for the parameters. Storing further attributes like DLN id or node value of the result nodes within the view can additionally reduce the number of joins in queries using this view.

We extended the XMLRDB query optimizer to utilize PXVs for rewriting XPath queries. For a given query we first try to apply usable PXVs before considering XMLRDB-maintained indices[3]. Normally the algorithms for finding suitable views for query rewriting exhibit a high complexity [1, 23]. With the PXV concept of replacing query fragments and the parameterization of the views we can greatly simplify the search for relevant views to exact matches since the compensation is given by the remaining part of the original query.

The query rewriting algorithm for PXVs works as follows. Take the first PXV from the list of available views and try to find a query rewriting according to Fig. 1. If it was successful this can be repeated for the remaining fragment of the query. Now repeat these steps for each remaining PXV using the rewritten or, in case no replacement was possible, the unchanged query. So with each iteration more fragments may get replaced by PXVs. Potentially a query can have several different rewritings depending on the order query fragments are replaced by PXVs. In order to ensure that our algorithm finds a good rewriting the list of available views is ordered according to the complexity of the view definition (e.g. number of steps, with steps in predicates counted as well). Thus replacing a small fragment of a query will not prevent replacing a more rewarding, larger one. Generally the view definitions should not overlap to a great extent[4]. The proposed algorithm for automatically generating PXVs (cf. Section 5) respects this property. Alternatively the view list could be sorted according to potential savings of using the views determined during view creation (cf. Section 5.2).

PXV support also required extending the XPath-to-SQL transformation component. Whenever it encounters a special view step, which was inserted during the query rewriting phase, it inserts an equijoin with the table containing the materialized view using the starting node reference attribute. The parameter values given in the view step are added as selection predicates. The result node reference attribute is used to add further steps. Fig. 5 shows the SQL generated for the query

/world/country[@name='Germany']/history/entry/@year

which was rewritten using view

//country[@name=$p1]/history/entry.

The view contains not only the global id for the result nodes but document id and DLN id as well. So we save an additional join with the *node* table. Compared to the SQL expression resulting from the original query we reduced the number of joins from 6 to 4.

```
SELECT DISTINCT a1.value
FROM   node x1, node x2, vtable1 v1, attr a1
WHERE  x1.name='world' AND x1.parent IS NULL AND
       x2.name='country' AND  x2.parent=x1.gid AND
       v1.startId=x2.gid AND v1.p1='Germany' AND
       a1.name='year' AND a1.gid=v1.resId
ORDER BY v1.resDoc,v1.resDlnId
```

Fig. 5. Generated SQL for */world/country/view::v1[@p1='Germany']/@year*

[3] According to [1] most of these index structures can also be seen as a kind of materialized views.

[4] Two view definitions q_{v1} and q_{v2} overlap if they share at least one common XPath step.

5 Automated PXV Creation

While PXVs can be manually created it is a challenging task to find a nearly optimal set of PXVs for a given workload over a database and a maximum space constraint. Therefore we have developed a PXV wizard which suggests a ranked list of PXVs for a given database and workload. Given a constraint on the maximum storage space for materialized views the wizard automatically determines the most promising PXVs for improving query performance.

There exist some previous work on mining frequent query patterns in tree-like structures [5, 24]. However these algorithms consider only the workload data but not the processing costs of the individual patterns. Hence good materialization candidates with high savings on accumulated query time may be missed when their pattern is less frequent than other patterns. Furthermore most of these algorithms are applicable only for a subset of XPath. We therefore implemented our own algorithm which uses a cost estimation to find rewarding view candidates. We will first describe the general idea and discuss later on how we can obtain a good cost estimation in XMLRDB.

5.1 General Approach

The formal notation of the following description is shown in Fig. 6. We assume the workload to be optimized consists of unique queries which may be weighted according to their execution frequency. For each query we generate successively all possible fragments. Since view definitions should exhibit some complexity in order to be relevant simple fragments are filtered out. Per query we now determine the cost saving potential $sav_{f/q}$ for each fragment if it would be materialized. This involves using a cost model and depends on the implementation. We will show this for XMLRDB in the next section.

We use a hash table *frags* to maintain the parameterized query fragments (cf. Section 3) together with their parameter values, query ids and potential savings multiplied by the query weights. If a fragment already exists in *frags* only the query id and potential saving multiplied by the query weight are added. Furthermore it is recorded if parameter values differ. After all queries and their fragments have been processed we check each parameter if only the same values were assigned to it. In this case and if the corresponding fragment was contained in at least two queries the parameter is replaced back by the constant value. Thus we only keep the required parameters. Now a list

```
FOREACH q ∈ workload {
  F ← fragments(q)
  F ← removeSimpleFrag(F)
  FOREACH f ∈ F {
    s ← getSaving(f, q) * weight(q)
    if contains(frags, f)
      e ← getEntry(frags, f)
      addQuery(e, q, s)
    else addEntry(frags, f, q, s)
} }
adjustParameters(frags)
filterMinSupport(frags)
rankedFrag ← descSort(frags)
FOREACH e ∈ rankedFrags {
  FOREACH q ∈ e {
    FOREACH e_r ∈ rankedFrag\e {
      removeQuery(e_r, q)
      if queryCount(e_r) = 0
        removeEntry(rankedFrag, e_r)
  } }
  descSort(rankedFrag)
}
```

Fig. 6. Algorithm to create view candidate list

rankedFrag with all entries from *frags* sorted by their potential savings in descending order is built.

To obtain a practically reasonable ranking we need to adjust the potential savings in *rankedFrag*. At the current stage we may have several top-ranked fragments of the same costly query. However since they typically will overlap it makes little sense to materialize all of them. We rather assume heuristically that only one view will be used for query rewriting. Thus we adjust the potential saving with the following algorithm. From the top entry in *rankedFrag* each query id is checked whether it occurs in the other list entries. For each entry containing such a query id this id is removed and the potential savings are reduced according to the share the query had. After all entries are processed *rankedFrag* is sorted again and the algorithm starts over with the next entry. In the end, the top listed entries are the best candidates for materialization under the assumption that in most cases only one view will be used for query rewriting.

The algorithm described so far has the possible limitation that the fragments of the top view candidates may be helpful for only a single query. Thus materializing such a candidate could benefit only a relatively small number of queries. To circumvent this we additionally require that the support of a query fragment *f*, *supp(f)*, should exceed a threshold *minSupp* [5]. Here, *supp(f)* is simply the number of workload queries containing *f* divided by the absolute number of workload queries ($0 < supp(f) \leq 1$). The *minSupp* filter restriction has to be applied to *rankedFrag* before the potential savings are adjusted.

5.2 Determining Savings in XMLRDB

In the previous section we argued that the potential saving of a materialized fragment for a query depends on the implementation and its cost model. We will now discuss the approach we use in XMLRDB. From a series of experiments we learned that a system independent, general relational cost model does not work because different relational databases may produce highly varying query plans.

Since an external cost model was not an option as explained before, we decided to utilize the *explain* facility of the relational database system. We only had to provide realistic queries to receive suitable cost estimations. Temporarily materializing all query fragments as views was not an option because of the large fragment number. Therefore we materialized dummy views with different cardinality and different ratios between the number of start nodes and result nodes. To calculate the potential saving $sav_{f/q}$ for a materialized fragment *f* and a query *q* we replace *f* by a corresponding dummy view v_d. The decision which dummy view will be used is based on a cardinality estimation component. This component maintains statistical data about the stored XML documents like child count per element type, minimum and maximum height within the document tree etc. For the fragment to be replaced we can now retrieve the estimated input and output cardinality and choose an appropriate dummy view. The query *q* as well as the rewritten query q_{rw} are translated to SQL. Using the explain facility we can calculate the potential saving as $sav_{f/q} = explain(toSQL(q)) - explain(toSQL(q_{rw}))$.

6 Evaluation

We evaluated the introduced PXV concept with our prototype XMLRDB in comparison with the standard configuration which only uses relational index structures but no special XML access structures. Furthermore we wanted to assess the quality of our automated PXV creation algorithm. We were especially interested in the benefit we can gain from using the cost based approach in comparison to a simple frequent pattern matching approach like [24].

Our test environment consists of a computer with 1 GB of main memory and a 2.4 GHz Pentium IV processor. We used the data set from XMark benchmark [19] with a scaling factor of 1 resulting in a raw XML document size of ca. 110 MB. It contains 2,8 million text and element nodes and 380,000 XML attributes. In order to create a reasonable workload we first translated the XMark set of queries which is formulated in XQuery into XPath as far as possible. Additional, more complex queries were generated by an XPath creation tool. It traverses the document and generates queries with multiple and recursive predicates as well as value comparisons. The complete workload consists of 50 queries with a maximum of 14 steps and a mean of 7 steps.

Fig. 7 and Fig. 8 show the execution times for the whole workload. The given values only contain the query execution time without materialization of the results. First (*NoPXV*) we run all workload queries without using PXVs. For the next run (*PXV_cost*) we run our automated PXV creation algorithm (took 6 minutes) and materialized the first 10 view candidates resulting in approx. 200,000 tuples in view tables which were created in 113 seconds. We choose the first 10 candidates because the potential savings of the following candidates were two orders of magnitude lower. Two additional runs were conducted to evaluate a purely frequency-based view selection. For these runs we ignored the cost estimations and sorted the view candidates according to their frequency in the workload queries. Thus we modelled a pure frequent pattern based approach. For *PXV_Pattern_10* we materialized the top 10 view candidates as we did it for the cost based variant. Since the number of tuples materialized were only a third in comparison to *PXV_cost* we materialized further view candidates until we reached the same number. *PXV_Pattern_14* denotes this configuration utilizing 14 materialized views.

Fig. 7 shows that the overall execution time improved by an order of magnitude using our proposed PXV concept. The pattern based candidate selection approach is 5 times slower. Note that adding more materialized views does not need to improve query time. Looking at the ignored saving values from the candidates we could see an estimated negative impact. Fig. 8 shows mean and maximum execution time of single queries within the workload. Here again we can see that the PXVs selected by our proposed algorithm can decrease the maximum execution time by an order of magnitude while the PXVs selected by the pattern based approach have no real impact on maximum execution time. The rewriting of the workload queries took typically less than a millisecond and is thus negligible compared to the query execution time.

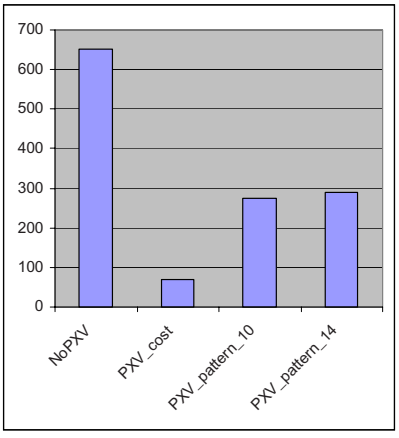

Fig. 7. Execution time in seconds for a workload of 50 queries with and without PXVs

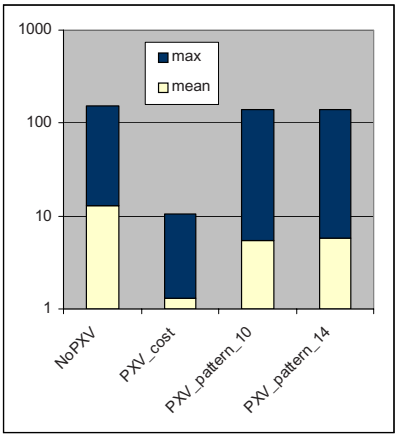

Fig. 8. Maximum and mean execution time in seconds for a workload of 50 queries with and without PXVs

7 Conclusion

We have introduced parameterized XPath views, PXVs, as a new concept for utilizing materialized views for efficient XML query processing. With its parameterization it enables to utilize a view for a broader range of similar queries. The additional information of starting node references stored within the view simplifies the adoption in queries without a costly calculation of compensations. We further showed how PXVs can be implemented in an XML database system like our XMLRDB prototype. Creating a rewarding set of materialized views is a complex task which is hardly feasible to do manually. Therefore we discussed an algorithm for automating it. Unlike other approaches which only take workload data into account for finding common query patterns we base our solution on a cost model and utilize the idea of materialized dummy views. With our evaluation we could verify that the PXV concept can be used to improve execution time of complex XPath queries considerably. Furthermore we showed that our cost based algorithm to automatically create PXVs achieves far better results than a pure workload pattern based approach.

Further work may address the view update problem and study the applicability of proposed solutions for the PXV concept.

References

[1] Balmin, A., Özcan, F., Beyer, K.S., Cochrane, R.J., Pirahesh, H.: A Framework for Using Materialized XPath Views in XML Query Processing. In: Proc. 30th VLDB Conf 2004 (2004)

[2] Böhme, T., Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In: Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb) 2004 (2004)

[3] Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., Candan, K.S.: Twig2Stack: Bottom-up Processing of Generalized-Tree-Pattern Queries over XML Documents. In: Proc. 32nd VLDB Conf., 2006 (2006)
[4] Chen, L., Rundensteiner, E.A.: ACE-XQ: A Cache-aware XQuery Answering System. In: Proc. 5th Int. Workshop on the Web and Databases (WebDB) (2002)
[5] Feng, J., Qian, Q., Wang, J., Zhou, L.: Exploit sequencing to accelerate hot XML query pattern mining. In: Proc. ACM Symposium on Applied Computing (SAC) (2006)
[6] Grahne, G., Thomo, A.: Query Containment and Rewriting Using Views for Regular Path Queries under Constraints. In: Proc. 22nd ACM Symposium on PODS (2003)
[7] Grust, T., van Keulen, M., Teubner, J.: Accelerating XPath evaluation in any RDBMS. ACM Trans. Database Syst. 29, 91–131 (2004)
[8] Grust, T., Sakr, S., Teubner, J.: XQuery on SQL Hosts. In: Proc. 30th VLDB Conf., 2004 (2004)
[9] Halevy, A.Y.: Answering Queries Using Views: A Survey. VLDB Journal, 10(4) (2001)
[10] Hammerschmidt, B.C., Kempa, M., Linnemann, V.: On the Intersection of XPath Expressions. In: Proc. 9th Int. Database Eng. and App. Symposium (IDEAS) (2005)
[11] Lakshmanan, L.V.S., Wang, H., Zhao, Z.: Answering Tree Pattern Queries Using Views. In: Proc. 32nd VLDB Conf., 2006 (2006)
[12] Mathis, C., Härder, T.: Hash-Based Structural Join Algorithms. In: Proc. 2nd Int. Workshop on Database Techn. for Handling XML Inform. on the Web (DataX) (2006)
[13] Miklau, G., Suciu, D.: Containment and Equivalence for a Fragment of XPath. ACM Journal 51(1) (2004)
[14] Mandhani, B., Suciu, D.: Query caching and view selection for xml databases. In: Proc. 31st VLDB Conf., 2005 (2005)
[15] O'Connor, M., Bellahsene, Z., Roantree, M.: An Extended Preorder Index for Optimising XPath Expressions. In: Bressan, S., Ceri, S., Hunt, E., Ives, Z.G., Bellahsène, Z., Rys, M., Unland, R. (eds.) XSym 2005. LNCS, vol. 3671, Springer, Heidelberg (2005)
[16] Onose, N., Deutsch, A., Papakonstantinou, Y., Curtmola, E.: Rewriting Nested XML Queries Using Nested Views. In: Proc.ACM SIGMOD Int. Conf. Mgmt. of Data. 2006 (2006)
[17] Papakonstantinou, Y., Vassalos, V.: Query Rewriting for Semistructured Data. In: Proc. ACM SIGMOD Conf., 1999 (1999)
[18] Shah, A., Chirkova, R.: Improving Query Performance Using Materialized XML Views: A Learning-Based Approach. In: Jeusfeld, M.A., Pastor, Ó. (eds.) Conceptual Modeling for Novel Application Domains. LNCS, vol. 2814, Springer, Heidelberg (2003)
[19] Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: Proc. 28th VLDB Conf., 2002 (2002)
[20] Wang, W., Jiang, H., Lu, H., Yu, J.X.: Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In: Proc. 30th VLDB Conf., 2004 (2004)
[21] Wang, H., Park, S., Fan, W., Yu, P.S.: ViST: a dynamic index method for querying XML data by tree structures. In: Proc. ACM SIGMOD Conf., 2003 (2003)
[22] Wang, W., Wang, H., Lu, H., Jiang, H., Lin, X., Li, J.: Efficient Processing of XML Path Queries Using the Disk-based F&B Index. In: Proc. 31st VLDB Conf., 2005 (2005)
[23] Xu, W., Özsoyoglu, Z.M.: Rewriting XPath Queries Using Materialized Views. In: Proc. 31st VLDB Conf., 2005 (2005)
[24] Yang, L.H., Lee, M.L., Hsu, W.: Efficient Mining of XML Query Patterns for Caching. In: Proc. 29th VLDB Conf., 2003 (2003)
[25] Zhang, N., Özsu, T., Aboulnaga, A., Ilyas, I.: XSeed: Accurate and Fast Cardinality Estimation for XPath Queries. In: Proc. 22nd Int. Conf. on Data Engin (ICDE) (2006)