Recommending XMLTable Views for XQuery Workloads

Iman Elghandour[1] *, Ashraf Aboulnaga[1], Daniel C. Zilio[2], and Calisto Zuzarte[2]

[1] University of Waterloo
[2] IBM Toronto Lab

Abstract. Physical structures, for example indexes and materialized views, can improve query execution performance by orders of magnitude. Hence, it is important to choose the right configuration of these physical structures for a given database. In this paper, we discuss the types of materialized views that are suitable for an XML database. We then focus on XMLTable materialized views and present a procedure to recommend them given an XML database and a workload of XQuery queries. We have implemented our XMLTable View Advisor in a prototype version based on IBM® DB2® V9.7, which supports both relational and XML data, and we experimentally demonstrate the effectiveness of our advisor's recommendations.

1 Introduction

XML is becoming widely adopted as a data storage and representation format. In addition to native XML database systems, most commercial database systems now support an XML column type and have query optimizers that can handle XML data and queries [6, 21, 22]. Furthermore, these database systems allow creating physical structures such as indexes and materialized views to improve the query execution performance of XML queries. For large databases and complex query workloads, it is challenging to choose the right configuration of physical structures that also have a reasonable disk usage.

Recommending indexes and materialized views as part of the physical database design process has previously been studied extensively in the context of relational databases, and most commercial database systems now include *Design Advisors* that automatically recommend various physical structures [2, 23]. The high-level outline of the recommendation process for XML databases is similar to that for relational databases. However, recommending indexes and materialized views for XML databases presents some unique challenges that make the problem more difficult than the relational case, and that lead to the details of the solutions being significantly different.

*Supported by an IBM PhD Fellowship. Also affiliated with Alexandria University, Alexandria, Egypt.

There are currently several types of materialized views for XML data. Different proposals have defined different view languages for XML data and have studied matching these views with XML queries. In this paper, we discuss these different approaches and we then focus on one of them, namely XMLTable materialized views. We discuss the advantages of using XMLTable materialized views, which are relational in structure, to improve the performance of XQuery workloads. Next, we present a physical design advisor that recommends XMLTable materialized views for XQuery workloads. We present an experimental study of the the effectiveness of this XMLTable View Advisor.

The main issues that we address when recommending materialized views are: (1) determining the candidate physical structures (materialized views) that would be useful for a query or a workload consisting of a set of queries, (2) expanding the set of candidates by adding new ones that are useful for multiple queries in the workload, and (3) searching the space of possible materialized view configurations for the optimal configuration that provides the maximum benefit to the workload while satisfying disk, schema, and other system constraints. In this paper, we present novel techniques to address each of these challenges. We have implemented our XMLTable View Advisor in a prototype version of DB2 V9.7, which supports both relational and XML databases, and we have used this implementation to verify the efficiency of our proposed advisor and the high quality of the view configurations that it recommends.

The rest of the paper is organized as follows. We present related work in Section 2. Next, we present our contributions, which can be summarized as follows:

- A brief discussion of the existing materialized view languages for XML data (Section 3).
- We propose an end to end solution for an XMLTable View Advisor that recommends relational materialized views that are constructed using the SQL XMLTable function (Section 4). Within our solution for the XMLTable View Advisor we make the following contributions: (1) a technique for enumerating XMLTable views that are useful for an XQuery query (Section 4.2), (2) an algorithm that translates XQuery queries into relational queries that use XMLTable views (Section 4.3), (3) a generalization algorithm that generates new XMLTable views that are useful for multiple queries in the current workload (Section 5), and (4) a search algorithm that extends the heuristic algorithm introduced in [10] to address the interaction between views (Section 6).
- An implementation of the XMLTable View Advisor in a prototype version of DB2 and an experimental study using the TPoX [19] benchmark (Section 7).

2 Related Work

In the past few years, there has been a considerable amount of work on automatic physical design for relational databases [2, 23]. Unfortunately, none of these works extend directly to XML databases. The XML Index Advisor proposed in [10] recommends XML indexes for an XML database given a workload of XML queries. Our XML View Advisor expands on the Index Advisor by recommending XMLTable views, which are more complex than the partial XML indexes recommended by the Index Advisor. In this section, we first discuss existing approaches that decide on how to store the data based on its characteristics. Next, we present previous cost based approaches that are used to recommend materialized views for XML databases.

Our approach relies on recommending relational materialized views for XML queries. Relational and XML data reside side by side in current database systems [6]. Query execution cost depends on the storage mode of the data, and so there are situations where it is appropriate to use a relational representation of the data and others where it is appropriate to use an XML representation. A discussion of the factors affecting the choice of using a relational or XML representation to store data is presented in [14, 18]. The proposed solution is to find a logical design for a database given the characteristics of the data to be stored in it. However, application access patterns of the data are also important. These access patterns can be exploited to add materialized views to the database to enhance performance [12]. To incorporate both relational and XML data models in the same database system, several hybrid XML-relational architectures are presented in [13].

Another area where relational and XML data coexist is publishing relational data as XML, an area that has been extensively studied in the last few years. In these systems, data is stored in relational stores and published as an XML schema, which requires translating XQuery queries into SQL queries, and translating relational data into XML data that satisfies the published XML schema. Most publishing techniques have one fixed way to translate the relational data into XML based on the XML schema. However, some research projects attempt using a cost based analysis for choosing the best translation [7, 9].

In MARS [9], the data is originally stored in relational and XML format, in addition to partial views of the data that are of relational and XML types. In that work, one virtual XML view is published and the incoming queries are translated according to the source that is chosen to answer them. A cost based analysis to choose the best query translation is proposed.

In LegoDB [7], the mapping between XML and relational views of the data is also chosen according to a cost based approach. The application is represented by a workload of queries and data statistics. A subset of the XML schema, called *p-schema*, is used to describe the data. P-schema has the advantage that it can be directly mapped to relational data, and also it is annotated by statistics information. Initially, different candidate p-schemas are enumerated. Then, a greedy heuristic search is used to find the best schema. The cost of a schema is estimated by performing the mapping between the XML data and the relational

storage, translating the XML workload according to this mapping, importing the XML statistics into the new relations, and finally, using a relational optimizer to estimate the cost of the workload.

An attempt to partially automate the logical design of a hybrid (Relational-XML) database system is presented in [18]. The input to the proposed *Schema Advisor* is an annotated information model that is considered as a conceptual design for the database. Based on this annotated model, the schema advisor analyzes different storage alternatives and chooses the best of them according to a scoring function. Users of the system can also give their input to the tool to guide the advisor process.

Another cost based approach for automating the logical design of XML databases is proposed in the ULoad project [4]. That work uses the XML Access Modules (XAMs) algebraic formalism to represent the data and its storage structures. ULoad uses a fixed set of designs to choose from, but the users can expand them with their own persistent data structures using the same graphical language. A structural summary of the data is then used to estimate the cost of answering a workload of queries given a configuration of XAMs.

3 Materialized Views for XML Data

Creating views of relational and XML data can take place on either the logical or physical level or both. On the logical design level, data can be XML and be published as relational views [13, 17], or data can be relational and be published as XML views [9, 17]. Queries are written according to the published schema, so if, for example, the published schema is XML and the data is stored in relational format, we need to (1) translate the XML queries to SQL queries according to the stored schema, and (2) transform the XML data to relational to be stored in the relational store, and vice versa for query answers.

On the physical design level, materialized views of XML data can be in one of the following forms:

1. Views of XML data fragments that are defined by XQuery queries [3, 20]. The queries written against the views are also in XQuery. Result containment is checked to decide if a view can answer a query.
2. Views of XML data fragments that are defined by XPath path expressions [5, 16]. Queries can be either XPath or full XQuery. In the latter case, indexes containing fragments of the data constitute the XML views.
3. Views of XML data elements and their values that are defined by XPath path expressions and stored in relational tables. XQuery queries are then translated into SQL queries to be executed on these materialized relational views. This approach is close to shredding the XML data into relational tables [7, 8]. We adopt this approach in this paper and elaborate on it next.

3.1 XMLTable Views of XML Data

Using relational materialized views for XML data and queries allows us to benefit from the rich and mature infrastructure for these views built into many database

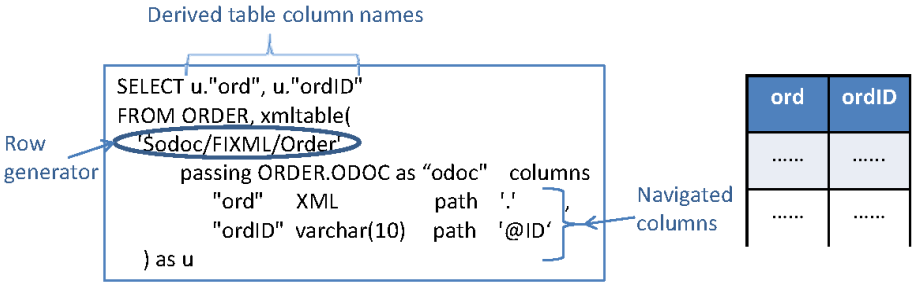


Fig. 1. XMLTable view example.

systems. Using these views provides a simple and effective way to improve the performance of XML workloads by leveraging existing infrastructure. Building relational views of XML data requires a mechanism that maps between XML elements and their corresponding column names in the relational views. For example, in ROX [13], the XML Wrapper of IBM DB2 [15] is used to do this mapping. The XML Wrapper allows CREATE NICKNAME statements that include nicknames for XPath expressions in the XML document.

A new approach for creating relational views for XML data is to use the XMLTable function [1, 21]. XMLTable is a SQL table function that creates a derived table based on XML data. The XMLTable function is applied on a table with an XML-type column. Each row of the table has an XML document in this XML-typed column, and the XMLTable functions maps elements occurring in these XML documents to columns in the derived table generated by the XMLTable function. The parameters of an XMLTable function are: (1) A row generator, which is a path expression. Each element reachable by this path expression corresponds to a tuple in the derived table. (2) Column navigators, which are Xpath navigation patterns. Each column navigator is used to populate a column in the derived table. The row generator specifies the rows in the derived table generated by the XMLTable function, and the column navigators specify the columns of these rows. Figure 1 illustrates an example SQL query with an XMLTable function.

Using the XMLTable function to create relational views of the XML data allows us to benefit from both the mature relational view matching [12] and also XPath view matching [5, 16]. The XMLTable is defined in the FROM clause of a SELECT statement which allows two levels of matching of queries with views. The optimizer matches queries that contain XMLTable functions with XMLTable views. Next, XMLTable definitions of the query and view can use XPath matching to find the needed compensation and so to rewrite the query to use the view contents. A discussion of the possible techniques and issues related to matching and rewriting queries to use XMLTable views is presented in [11].

In this paper, our goal is to recommend XMLTable materialized views that benefit a workload of XQuery queries on data that is stored in an XML-typed column of a table. This requires: (1) enumerating XMLTable views for an XQuery query and translating the query to use the views, (2) expanding the set of can-

didate views, and (3) choosing the best set of views given a disk space budget. We elaborate on these three steps in Sections 4-6, respectively.

4 View Enumeration Process

4.1 Types of XMLTable Views

We employ a cost based analysis to choose the views that would benefit the queries in the workload the most. The high level architecture of the XMLTable View Advisor is as follows. First, we analyze each query in the workload and enumerate its possible XMLTable view candidates. The set of XMLTable views enumerated for all queries in the workload constitutes our basic set of candidate views. Next, we expand the set of candidate views by recommending more general views that can answer more queries in the workload. Then, for each candidate view, we invoke the query optimizer in a special mode to estimate the benefit of the view to the queries in the workload. Finally, we search the space of candidates to find the best configuration of views that has the highest benefit to the workload and fits into the given disk space budget. Our advisor architecture is similar to that of the XML Index Advisor described in [10]. The proposed advisor is based on employing common access patterns of XQuery queries to decide on the views that are useful for them. For example, if a query frequently accesses an element's value in the XML data (an ID for instance), then it is beneficial to extract it as a separate column in the XMLTable view.

The class of XQuery queries that we support includes queries with FOR, LET, WHERE, and RETURN clauses. The RETURN clause can have either a simple or a constructed expression. The general form of a query that we support is as follows:

GQ

```
for $forVar in (ColumnName)/forExpr[forPredicate]
let $letVar := aggFn(letExpr)
where wherePredicate
return returnExpr
```

We use the following query *Q1* on the TPoX [19] benchmark database as a running example:

Q1: For every customer whose age is greater than 50 and has an ID greater than 9000, return her name and the number of accounts she has.

```
for $cust in ("CUSTACC.CADOC")/Customer[@id > 9000]
let $accounts := count($cust/Accounts/Account)
where $cust/age > 50
return
    <print>
        <name>$cust/name</name>
        <accounts_number>$accounts</accounts_number>
    </print>
```

4.2 Enumerating Candidate Views

To enumerate candidate views for an XQuery, we parse the query and break it down into its FOR, LET, WHERE, and RETURN clauses. Then, for each one of these clauses we further break it into its components. We describe next how we handle each clause in the candidate enumeration process (Algorithm 1).

Algorithm 1 enumerateCandidates(*xquery*)

```
1: for clause ∈ xquery do
2:   if clause is forClause then
3:     create a new view view and associate it with the variable $forVar
4:     set the row generator of view to be forExpr
5:     for p ∈ forPredicate do
6:       add p to view as a column navigator
7:     end for
8:   else if clause is letClause then
9:     create a new view view and associate it with the variable $letVar
10:    if letExpr references an existing refView then
11:      resolvedLetExpr ← append the row generator of refView and letExpr
12:      set the row generator of view to be resolvedLetExpr
13:      add column "." to refView and a backward navigation path to view (these
         columns are used to join the two views view and refView)
14:    else
15:      set the row generator of view to be letExpr
16:    end if
17:    if clause has aggFn then
18:      add a SQL GROUP BY clause to view with all columns except the expres-
         sion that appears in the aggFn
19:    end if
20:  else if clause is whereClause then
21:    for p ∈ wherePredicate do
22:      find refView that is referenced in p
23:      add p to refView as a column navigator
24:    end for
25:  else if clause is returnClause then
26:    for expr ∈ returnExpr do
27:      find refView that is referenced in expr
28:      add expr to refView as a column navigator
29:    end for
30:  end if
31: end for
```

FOR Clause. We divide the FOR clause into a variable, a path expression, and its optional predicates. For every FOR clause: (1) we create a new view and assign its row generator to be the path expression extractor in the FOR clause (i.e. the path expression after removing any predicate values from it, *forExpr* in GQ), (2) we record the variable name and the created view so we can add any expression that references it to the view as a column, and (3) finally, for every

path appearing in a predicate, we create a navigation path and add it to the view. For example, when we parse the FOR clause of *Q1*, we create a new view *V1* that has the row generator `/Customer` and the column `@id`:

V1:
```
select u.cx0 from CUSTACC, xmltable(
    '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
     columns
            cx0  double path '@id') as u
```

LET Clause. Similar to the FOR clause, we parse the LET clause to find the clause variable (*letVar* in *GQ*) and its binding expression (*letExpr* in *GQ*). In addition, a LET clause might have an optional aggregation function that we only take into account when we rewrite the XQuery to use the view and a binding expression that references a previously bound variable (`$cust/Accounts/Account` in Q1). For an expression with a reference variable, we look up the expression referenced by this variable (`/Customer` in this example) and concatenate it with the rest of the expression to form the path expression we use for this clause. We then create a new view with that new path expression as a row generator. We add a column in each of the newly created view and the old one to be used for joining them together in the translated query. The updated version of *V1* and the newly created *V2* will be as follows:

V1:
```
select u.cx0, u.cx1 from CUSTACC, xmltable(
    '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
     columns
            cx0  double path '@id',
            cx1  xml path '.') as u
```

V2:
```
select count(u.cy0) as ACc1, u.cy1 from CUSTACC, xmltable(
    '$cadoc/Customer/Accounts/Account' passing CUSTACC.CADOC as "cadoc"
     columns
            cy0  xml path '.',
            cy1  double path 'parent::Accounts/parent::Customer') as u
group by cy1
```

WHERE Clause. For every predicate appearing in a WHERE clause, we handle each predicate expression by finding the view referenced by the variable that appears in this expression, and adding a column to that view to correspond to this navigation. To account for the predicate on `age` in Q1, view *V1* is now written as follows:

V1:
```
select u.cx0, u.cx1, u.cx2 from CUSTACC, xmltable(
    '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
     columns
            cx0  double path '@id',
            cx1  xml path '.',
            cx2  double path 'age') as u
```

RETURN Clause. For all the expressions that appear in the RETURN clause, we find all the variables that reference views and we find the views that they reference. We add a column for each variable to the corresponding view. View *V1* can be updated now to have `name` as a column:

V1:

```
select u.cx0, u.cx1, u.cx2, u.cx3 from CUSTACC, xmltable(
    '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
     columns
            cx0  double path '@id',
            cx1  xml path '.',
            cx2  double path 'age',
            cx3  varchar(100) path 'name') as u
```

4.3 Translating XQuery Queries into SQL Queries that Use XMLTable Views

Current XML query optimizers lack the infrastructure to perform the matching of XQuery queries with relational (XMLTable) views. Existing matching algorithms match queries with XMLTable function to XMLTable views [11]. Therefore, we outline in this section a procedure to translate XQuery queries into SQL queries with XMLTable functions. The translation involves using views that are similar to the ones being recommended, and hence we perform the candidate enumeration step described in the previous section to find the best suitable view for a query. Next, we use these recommended views to rewrite the query.

We examine the parsed XQuery, and then construct an SQL query based on this information. We add all the recommended views to the FROM clause of the SQL query. We then use the column names in the views in the SELECT clause and WHERE clause according to the binding of variables and how they appear in the original query. We also add joins between the views that are used to rewrite the query when needed.

For example, we have recommended two views *V1* and *V2* for *Q1* and we can now construct the FROM clause as `FROM V1, V2`. Next, we examine the return clause and construct the SELECT clause of the rewritten query. If the return value is a simple XPath expression, then the corresponding column name is used, otherwise if an XML fragment is constructed, an XQuery construction is done using the XMLELEMENT function. Finally, we construct the WHERE clause as a conjunction of all the predicates that appear in the XQuery and those that correspond to joins between views. The final rewritten query for Q1 is as follows:

Rewritten Query: RQ1

```
select  XMLELEMENT( NAME  "print" , XMLELEMENT( NAME  "name" ,
     Vv1.cx3) , XMLELEMENT( NAME "accounts_number" , Vv2.ACc1))
from  (..same as V1..) as Vv1, (..same as V2.. ) as Vv2
where  ( Vv1.cx2  >  50 )  and  ( Vv1.cx0  >  9000 )
    and ( Vv2.cy1  =  Vv1.cx1 )
```

5 Expanding the set of Enumerated Views

In the XML Index Advisor [10], we have found that generalizing the index patterns makes them useful for queries not seen in the workload that is used for the recommendation. Similarly, creating views that answer multiple queries in the workload and potential unseen queries can increase the usefulness of our recommendations. Since our proposed view definition encapsulates both XPath expressions and SQL query definitions, generalization can benefit from the index generalization techniques we proposed in [10] and the query merging techniques proposed in [23]. The possible generalization techniques include generalizing the row generator or the column navigator of the view, and merging views. In addition, it is possible to use relational indexes on XMLTable views to increase their benefit. We describe some of the possible query generalization forms that we have explored in this section.

Generalizing Column Navigators to Include Subtrees. Most of the XMLTable views that we recommend in the enumeration phase are a normalization (flattening) of all the values that are being accessed in the workload queries. An alternative approach is to recommend views that store sub-trees of the data as XML columns. A recommended XMLTable view can now have the XPath path expression to reach the data as the row generator and one column with a "." path expression to represent all the subtrees reachable by that row generator. For example, *V3* (below) is a generalization of *V1*. This approach is useful when the query requires reconstructing the XML tree. This general view requires that the matching infrastructure allows matching multiple columns in the query with one column in the view and is also capable of performing XPath compensation. For example matching view *V3* with *Vv0* in query *RQ1* means matching columns *cx0*, *cx1*, *cx2* and *cx3* in *Vv0* with *cx0* in *V3* and requires navigating for *@id*, *age*, and *name*, respectively, in the rewritten query that uses the view. Instead of replacing the columns of a view with a "." column, a less aggressive approach for generalizing column navigators is to consider pairs of views that share the same row generator and consider pairs of columns, one from each view, and generalize these columns together using index generalization algorithms that we propose in [10].

V3:

```
select u.cx0 from CUSTACC, xmltable(
    '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
     columns
            cx0  int path '.') as u
```

Merging Views. A common generalization approach used in relational advisors is view merging [23]. For XMLTable views, we merge views that have the same row generator to produce a new view that has the set of column navigators that appear in the merged views after removing duplicates. The goal of this approach is to decrease the disk space required for views by removing duplicate columns from the merged views, while still achieving the same performance.

Indexes on XMLTable Views. One approach to make XMLTable views more useful is to build relational indexes on their columns and hence improve query performance. This is possible since the XMLTable views are defined in the form of SQL statements that produce relations. There can be many possible indexes that can be built on the different columns of an XMLTable view to help the view perform better. In this paper, we use a heuristic approach to select only one index for each view. The chosen index has all the columns of the view that have originally participated in a predicate in the XQuery that caused this view to be recommended. This way we guarantee that these columns have relational values that are used for lookup in the query. The index follows the same order of the columns in the view. For example, the index that we recommend for view *V1* is "`create index index1 on V1(cx0, cx2)`". For every candidate view, we add to the search space another alternative structure which is composed of the view with a relational index over its columns.

6 Searching for the Optimal View Configuration

To recommend a set of XMLTable views (a view configuration) for a workload, we need to search the space of candidate views to find the best set of views that fits into a given disk space budget. We generalize the search algorithms in [10] to be able to search any physical structure (indexes, views, views with indexes on them, etc.). The search problem can be modeled as a 0/1 knapsack problem. The size of the knapsack is the disk space budget specified by the user. Each candidate physical structure – which is an "item" that can be placed in the knapsack – has a *cost*, which is its estimated size, and a *benefit*. We compute the benefit of a physical structure as the difference between the workload cost as estimated by the query optimizer before and after creating this structure.

XMLTable views can interact with each other in ways that reduce their total benefit for a query workload. Our search algorithm takes such interactions into consideration. The main types of interaction affecting the selection of views are: (1) views that can be used together to rewrite a query, and (2) views that are generated by merging other views. These interaction factors are similar to the ones encountered when searching the space of XML indexes, so we use a greedy search algorithm as in [10], but we modify the heuristic rules used in this search to deal with interactions so that they suit the view search problem.

The high level outline of the greedy search algorithm is as follows. First, we estimate the size of each candidate view, and the total benefit of this view for the workload. We then sort the candidate views according to their benefit/size ratio. Finally, we add candidates to the output configuration in sorted order of benefit/size ratio if they agree with the heuristic rules, starting with the highest ratio, and we continue until the available disk space budget is exhausted. In [10] we proposed heuristic rules that are based on *index coverage*. We define the *view coverage* of a view as its view ID as well as the ID of the views that it subsumes. Subsequently, the coverage of a configuration of views is the combination of the view coverage of its constituent views. For example, if *V3* is generated by

merging *V1* and *V2*, then the coverage of *V3* is the set of {1, 2, 3}. We refer to the coverage of a candidate view (*cand*) or a group of views (*config*) as *cand.coverage* and *config.coverage* respectively. We also refer to the size of a candidate view (*cand*) as *cand.size*. Algorithm 2 outlines the the search algorithm. We use the following functions to apply the heuristics and perform the search:

- *benefit*(*config*) returns the estimated benefit of the workload when this configuration of views (or views with relational indexes on them) is created. It is based on calling the query optimizer with and without the views in place and computing the reduction in the optimizer's estimated cost when the views are in place.
- *addCandIfSpaceAvl*(*cand*, *config*) adds the candidate (*cand*) to the configuration (*config*) if the $cand.size + config.size \leq diskConstraint$. In addition, if the condition holds, *addCandIfSpaceAvl* updates the *size* and *coverage* of *config*.
- *replaceCandIfSpaceAvl*(*cand*, *subConfig*, *config*) replaces the *subConfig* in *config* with *cand* if the new configuration after performing the replacement *newConfig* has a higher benefit than *config* and the added size is below a threshold β. This is the heuristic that we add to the greedy search to deal with view interactions. The value β is a threshold that specifies how much increase in size we are willing to allow. We have found $\beta = 10\%$ to work well in our experiments. Finally, if the condition holds and there is enough disk space to do the replacement, *replaceCandIfSpaceAvl* updates the *size* and *coverage* of *config*.
- *overlapConfig*(*config1*, *config2*) scans a *config2* and returns the minimal *subConfig* configuration that has the view coverage of *config1*.

Algorithm 2 heuristicSearch(*candidates*, *diskConstraint*)

```
1: sort candidates according to their benefit(cand)/cand.size ratio
2: recommended ← ∅, recommended.size ← 0, recommended.coverage ← ∅
3: while recommended.size < diskConstraint do
4:    bestCand ← pick the next best cand in candidates
5:    if recommended.coverage = ∅ or recommended.coverage∩bestCand.coverage = φ
      then
6:       addCandIfSpaceAvl(bestCand,recommended)
7:    else if recommended.coverage ≥ best.coverage then
8:       replaceCandIfSpaceAvl (bestCand,recommended,recommended)
9:    else
10:      overlapConfig ← overlapCoverage(bestCand, recommended)
11:      replaceCandIfSpaceAvl (bestCand,overlapConfig,recommended)
12:   end if
13: end while
14: return recommended
```

7 Experiments

7.1 Experimental Setup

Since V9.1, DB2 supports both relational and XML data [6]. We have used an initial prototype version of IBM DB2 V9.7 that was modified to support creating materialized views using the XMLTable function [1]. The client side of the XMLTable View Advisor is implemented in Java 1.6, and communicates with the prototype server via JDBC. We have conducted our experiments on a Dell PowerEdge 2850 server with two Intel Xeon 2.8GHz CPUs (with hyperthreading) and 4GB of memory running SuSE Linux 10. The database is stored on a 146GB 10K RPM SCSI drive.

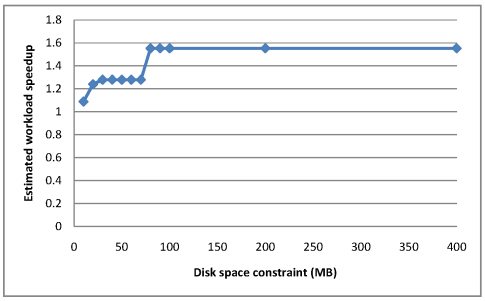
We used the TPoX [19] benchmark in our experiments. We generate the data using a scale factor of 1GB. We evaluate our advisor on the standard 10 queries that are part of the benchmark specification. We have made minor changes to the workload queries to account for some implementation limitations.

Our XMLTable View Advisor implementation has some limitations due to the existing DB2 prototype infrastructure. These limitations make our advisor unable to recommend views for certain XQuery query types. We can only use SQL data types for columns that appear in the XMLTable functions, since casting XML data into their corresponding relational data types fails in some cases. In addition, columns in XMLTable functions can only be elements; hence, subtrees reachable by an XPath expression, or linear expressions that select several elements will be concatenated into one large string value. This is not the correct approach when executing XQuery queries. Moreover, our implementation does not support more than two joins per query. We have also left adding support for structured queries, which are XQuery queries with a sub-query in the return clause, for future work. However, these limitations have not prevented us from verifying the usefulness of XMLTable views to answer XQuery queries.

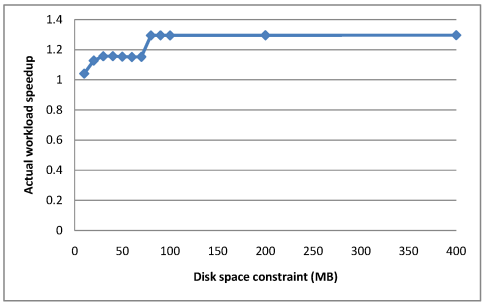
7.2 Effectiveness of the XMLTable View Advisor Recommendations

Figure 2 shows the estimated (based on query optimizer estimates) and actual (based on measured execution time) speedups for the TPoX workload. Speedup is defined as the execution time (estimated or actual) of the workload when no XML physical structures are created in the database divided by the execution time of the workload with the view configuration recommended by our advisor is in place. Both figures show that a maximum ratio of 1.6 (for the estimated workload execution speedup) and 1.3 (for the actual workload execution speedup) is achieved when we create the recommended views. Since some queries in the workload did not benefit from views, we also show the estimated and actual execution time of each query in Figure 3. Figures 3(a) and 3(b) show the estimated and actual execution time per query for a configuration with no views and view configurations of different sizes. Queries Q1, Q2, Q7, Q8, Q9, and Q10, which range from simple navigation to join queries, have benefited from the recommended XMLTable views. The actual speedup exceeded 3000 for some queries,

for example Q1 and Q7. The configuration which consists of all useful views has a size of 115 MB, which also helped us to achieve an average speedup per query of 639 (the speedup of queries that did not benefit from views is 1). Even for a configuration size of 9.8 MB, the average speedup per query is 134 which proves that XMLTable views can be useful for many query types.

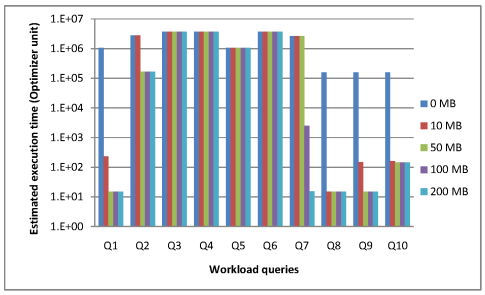


(a) Estimated speedup.

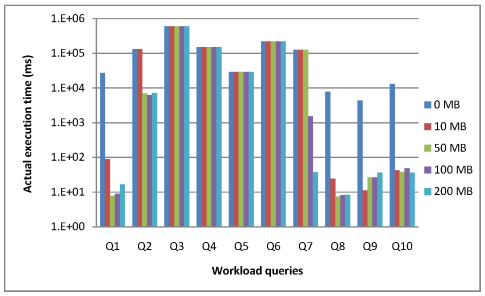


(b) Actual speedup.

Fig. 2. Workload speedup for the recommended XMLTABLE views.



(a) Estimated execution time.



(b) Actual execution time.

Fig. 3. Query execution time per query for the recommended XMLTABLE views.

8 Conclusions

In this paper, we have presented an XMLTable View Advisor. This is a new approach for building relational materialized views for XQuery workloads. Our XMLTable View Advisor recommends relational views that are in the form of XMLTable views. These views are useful in pre-navigating to queried values that appear in the data. In addition, XMLTable view matching is based on relational view matching and XPath matching, and hence we benefit from leveraging the already existing infrastructure of many database system query optimizers. We have implemented our advisor in a prototype version of DB2, and our experiments with this implementation show that our advisor can effectively recommend views that result in orders of magnitude performance improvement for some queries.

References

1. XMLTABLE overview, 2006. Available at: http://publib.boulder.ibm.com/infocenter/db2luw/v9/.
2. S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *VLDB*, 2004.
3. A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.
4. A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: choosing the right storage for your XML application. In *VLDB*, 2005.
5. A. Balmin, F. Özcan, K. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.
6. K. Beyer et al. DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal*, 45(2), 2006.
7. P. Bohannon, J. Freire, J. R. Haritsa, and M. Ramanath. LegoDB: Customizing relational storage for XML documents. In *VLDB*, 2002.
8. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.
9. A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.
10. I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. Beyer, and C. Zuzarte. XML index recommendation with tight optimizer coupling. In *ICDE*, 2008.
11. P. Godfrey, J. Gryz, A. Hoppe, W. Ma, and C. Zuzarte. Query rewrites with views for XML in DB2. In *ICDE*, 2009.
12. A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.
13. A. Halverson, V. Josifovski, G. M. Lohman, H. Pirahesh, and M. Mörschel. ROX: Relational over XML. In *VLDB*, 2004.
14. Comparing XML and relational storage: A best practices guide. IBM: Storage best practices, 2005.
15. V. Josifovski, S. Massmann, and F. Naumann. Super-Fast XML wrapper generation in DB2: A demonstration. In *ICDE*, 2003.
16. B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.
17. I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB*, 2001.
18. M. M. Moro, L. Lim, and Y.-C. Chang. Schema advisor for hybrid relational-XML DBMS. In *SIGMOD*, 2007.
19. M. Nicola, I. Kogan, and B. Schiefer. An XML transaction processing benchmark. In *SIGMOD*, 2007. Benchmark Available at: https://sourceforge.net/projects/tpox/.
20. N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.
21. Oracle Corp. *Oracle Database 11g Release 1 XML DB Developer's Guide*, 2007. Available at: http://www.oracle.com/pls/db111/.
22. M. Rys. XML and relational database management systems: Inside Microsoft SQL Server 2005. In *SIGMOD*, 2005.
23. D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *VLDB*, 2004.