Recommending XML Physical Designs for XML Databases

Iman Elghandour · Ashraf Aboulnaga · Daniel C. Zilio · Calisto Zuzarte

Abstract Database systems employ physical structures such as indexes and materialized views to improve query performance, potentially by orders of magnitude. It is therefore important for a database administrator to choose the appropriate configuration of these physical structures for a given database. XML database systems are increasingly being used to manage semi-structured data, and XML support has been added to commercial database systems. In this paper, we address the problem of automatic physical design for XML databases, which is the process of automatically selecting the best set of physical structures for a database and a query workload. We focus on recommending two types of physical structures: XML indexes and relational materialized views of XML data. We present a design advisor for recommending XML indexes, one for recommending materialized views, and an integrated design advisor that recommends both indexes and materialized views. A key characteristic of our advisors is that they are tightly coupled with the query optimizer of the database system, and they rely on the optimizer for enumerating and evaluating physical designs. We have implemented our advisors in a prototype version of IBM DB2 V9, and we experimentally demonstrate the effectiveness of their recommendations using this implementation.

Keywords Database physical design · XML database · Design advisor · XMLTable views · XML indexes

Iman Elghandour (✉)
Alexandria University, E-mail: ielghand@alexu.edu.eg
(Work done while the author was at the University of Waterloo)

Ashraf Aboulnaga
University of Waterloo, E-mail: ashraf@uwaterloo.ca

Daniel C. Zilio, Calisto Zuzarte
IBM Toronto Lab, E-mail: {zilio, calisto}@ca.ibm.com

1 Introduction

Recently, an increasing amount of data is exchanged, processed and stored in XML format. In addition, XML is now commonly used in many applications to represent and exchange semi-structured data. This has led to an increased focus on XML data management. There are three main approaches for storing and managing XML data: (1) native XML databases, (2) shredding XML data into relational databases, and (3) XML column type. In this paper, we focus on XML data that is natively stored in a column of type XML in a table in a relational database. This approach is now supported by most commercial database systems [5, 30].

Database systems introduce several physical structures to improve the performance of query execution. Examples of physical structures that relational database systems support and XML database systems fully or partially support are indexes, materialized views, and partitioning. For XML databases, the performance improvements provided by these physical structures stem primarily from: (1) direct access to parts of the data in the XML documents without needing to scan them (e.g., indexes), (2) grouping parts of the data into one logical unit that can be scanned independently of other such units (e.g., materialized views and partitioning), and (3) rewriting the query for a smaller part of the data (e.g., materialized views).

The various XML physical structures can potentially improve the performance of XML database systems by orders of magnitude. Users of these systems now face the problem of deciding on the best set of physical structures to create for a given XML database and query workload. Automatic physical database design has been studied extensively in the context of relational databases, and most commercial database systems now include *Design Advisors* that automatically recommend various physical structures [1, 7, 34, 36].

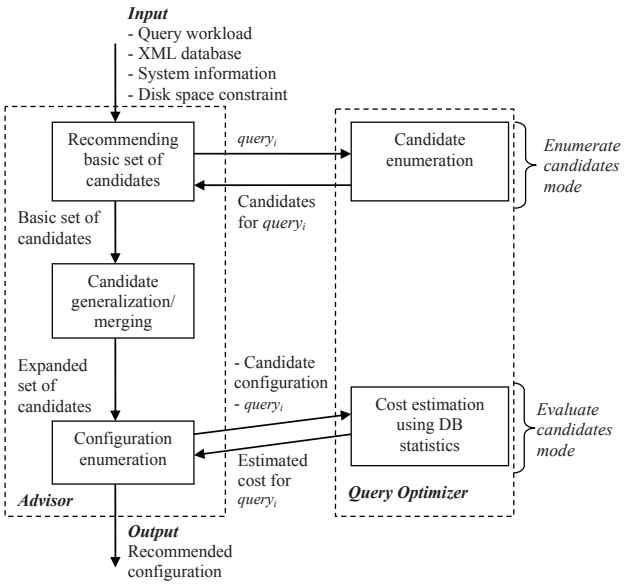


Fig. 1 General architecture of a physical design advisor.

However, automatic physical design for XML databases has not been studied as extensively in the database literature.

In this paper, we study automating the physical design of XML databases and build a system to recommend physical structures that are useful for a given XML database and query workload. We study the automatic recommendation of two physical structures: XML indexes and XML materialized views. A well-established architecture for physical design recommendation has been developed in the context of relational physical design advisors. A design advisor needs to address four questions: (1) how to determine the candidate structures that would be useful for a query workload, (2) how to expand the set of candidates with more general candidates, (3) how to estimate the benefit of a physical design configuration (i.e., a set of physical structures), and (4) how to search all the possible configurations for the best configuration. The recommendation process is divided into several phases where each phase addresses one of these questions. Figure 1 shows the general architecture of relational database design advisors, which we follow in our proposed XML design advisors. We extend the query optimizer with operation modes that allows us to: (1) recommend physical structures that can be useful for a query and (2) estimate the cost of a query while assuming the existence of some physical structures.

XML databases have unique characteristics, and so their physical structures are also different from the ones that are defined for relational database systems. This introduces unique challenges, such as identifying the patterns occurring in an XML database that can be indexed, finding general forms of the identified patterns, and consequently, searching a large number of candidate patterns. These unique challenges make automatic physical design for XML databases more difficult than that for relational databases and lead to the details of the physical design procedure being significantly different. Also, the physical structures for XML databases are not yet well established, and so there is an opportunity for research on automatic physical design to impact the definition of the physical structures being recommended. For example, a wide variety of XML indexes have been explored in the literature [13, 28, 30]. On the other hand, XML materialized views of various types are still being investigated in research [2, 15, 18, 30]. In this paper, we explore using the result of the XMLTable functions [18, 30] as relational-structured materialized views to speed up answering XQuery queries, and we develop an advisor that recommends XMLTable views for a given workload of XQuery queries.

Our focus in this paper is on developing techniques and algorithms to automate the recommendation of XML indexes and XMLTable materialized views for a given XML database and XML query workload. We present two end-to-end advisors: an XML Index Advisor and an XMLTable View Advisor. We then incorporate these two advisors into one Integrated Index-View Advisor that recommends both XML indexes and XMLTable materialized views for an XML database. An earlier version of the XML Index Advisor appeared in [10] and was demonstrated in [9], and the client-side XML Index Advisor application is available for download from the IBM developerWorks web site. In this paper, we present a new technique to deal with database maintenance statements (update, delete, and insert) in the input workload. An earlier version of the XMLTable View Advisor appeared in [11]. We expand that work by presenting the query optimizer architecture that can rewrite XQuery queries as needed by the View Advisor, and presenting the details of the query translation algorithm. The Integrated Index-View Advisor is a new research contribution beyond what appears in [9, 10, 11].

In the rest of the paper, we present our contributions, which can be summarized as follows: (1) the algorithms needed for building an XML Index Advisor (Section 3), (2) an XMLTable View Advisor (Section 4), (3) a combination of our index and view advisors proposed in Sections 3 and 4 to build an Integrated Index-View Advisor (Section 5), and (4) an implementation of our advisors on top of IBM DB2 V9 and an experimental study that uses this implementation (Section 6).

2 Background and Related Work

2.1 XML Physical Structures

Several types of physical structures can be used to improve the performance of query execution. In this paper, we focus on XML indexes and XMLTable views, which we briefly describe in this section.

2.1.1 XML Indexes

XML query languages (for example, XQuery and SQL/XML) use XPath path expressions to represent elements to be retrieved from the data. The retrieval of elements from the XML data can be helped by the presence of an XML index, and there have been many proposals for XML indexes over the past few years [13, 28, 30]. XML indexes can be categorized into *structural indexes* that speed up navigation through the hierarchical structure of the XML data (e.g., [13]), and *value indexes* that help in retrieving XML elements based on some condition on the values they contain (e.g., [28, 30]). A structural index can help in answering an XPath query such as `/Security/Symbol` (find all security symbols), while a value index can help in answering an XPath query like `/Security[Yield >= 4.5]` (find all securities with a yield greater than 4.5).[1]

Covering indexes (for example, DataGuide [13]) can grow as large as the data that they index [20], so they might not improve query execution time and are harder to maintain compared to smaller ones. However, *partial indexes*, which include only the XML elements that are reachable via specific *index patterns* [5, 30, 32] can improve the speed of index maintenance and lookups. These index patterns are typically specified as linear XPath expressions that do not include predicates. For example, an index that includes only XML elements that are reachable by the pattern `/Security/*` (i.e., immediate children of `/Security`) would be useful for answering queries such as the example queries above, but it would not be useful in answering queries on, say, `/Security/SecInfo//Sector`.

2.1.2 XMLTable Views

On the physical design level, materialized views of XML data can be in one of the following forms: (1) both the view and query language are XQuery [2, 29]. In this case, the main research issue is to check XQuery result containment to decide whether a view can be used to answer a query; (2) the view language is XPath and the query language is XQuery [4, 35]. In this approach, the XML views are defined using XPath path expressions and are similar to the XML indexes described in Section 2.1.1; and (3) the view language is a combination of XPath and SQL, and the query language is XQuery. This approach has some similarities to shredding the XML data into relational tables [6, 33]. In this paper, we adopt the third approach, which we can call *selective shredding*, and we elaborate on it in Section 4.

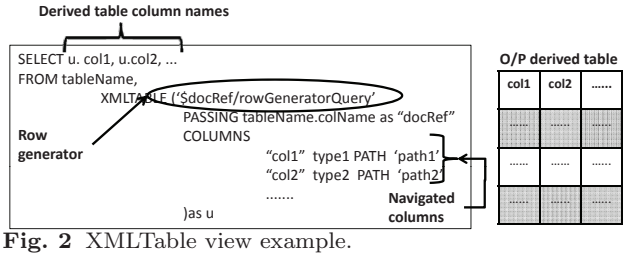


Fig. 2 XMLTable view example.

Using relational materialized views for XML data and queries provides a simple and effective way to improve the performance of XML query workloads by leveraging the existing rich and mature infrastructure for these views built into many database systems. However, 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 [15], the XML Wrapper of IBM DB2 [19] is used to perform 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 [12, 18, 30]. XMLTable is an SQL table function that creates a virtual derived relational table based on XML data. The virtual table can then be queried using SQL or materialized as a relational view [12]. An example of using the XMLTable function to create indexes is described in [24]. The XMLTable function is executed on a table with an XML-typed column. The XMLTable function allows us to include parts of the XML data that is frequently accessed by queries in the workload in a relational table format for faster retrieval. By doing so, we selectively shred the XML data into relational views. Figure 2 illustrates an example SQL query with an XMLTable function. The main parts of the syntax of the XMLTable function are as follows:

– A *row generator* XQuery string, which is an XQuery (or XPath) expression. The XMLTable function iterates through the results of the XQuery (or XPath) expression in the row generator and generates a tuple in the derived table for every result.

[1] Throughout this paper, we use examples from the TPoX benchmark [27].

– *Column navigators* are XPath navigation expressions. Their execution results are used to populate the columns of the derived table to be created by the XMLTable function.

Using the XMLTable function to create materialized relational views of the XML data allows us to benefit from both the mature relational view matching [14] and also XPath view matching [4, 35]. The XMLTable is defined in the FROM clause of a SELECT statement which allows two levels of matching of queries with views. The query optimizer matches queries that contain XMLTable functions with XMLTable views. Next, XPath matching can be used for the XMLTable definitions of the query and view to find the necessary compensation and so to rewrite the query to use the contents of the view. A discussion of the possible techniques and issues related to matching and rewriting SQL/XML queries with XMLTable functions to use XMLTable views is presented in [12]. That work focuses on describing the matching and rewriting rules needed by a query optimizer to use XMLTable views.

The XQuery Update Facility (XQUF), which is a W3C Recommendation that provides a declarative approach for updating XML, has been shown to effectively update XML data and XMLTable indexes [22, 24]. That work proves the effectiveness of using XMLTable views as physical structures for workloads that contain both queries and database maintenance statements.

2.2 Related Work on Automatic Physical Design for XML

Two works have attempted to tackle the index recommendation problem for XML databases [17, 31]. They both suffer from having rudimentary techniques for candidate generation, cost estimation, and configuration enumeration. Furthermore, the index advisors proposed in these works are independent of the database system query optimizer, so there is no guarantee that the recommended indexes will be of use to the optimizer, and no guarantee that the estimated benefits of candidate index configurations are accurate.

In [31], a tool is proposed for selecting indexes for an XML database system. The main focus of the work is to find a good cost model for selecting the best set of indexes for a query workload, making use of structural information and data statistics. In our work, we adopt a simple and powerful solution to the cost estimation problem by leveraging the query optimizer cost model.

Another index recommender for XML is presented in [16, 17]. This index recommender analyzes the workload periodically and creates or drops XML indexes on the fly. As in [31], the cost model used is independent of the query optimizer and hence likely to be inaccurate. For configuration enumeration, [16] proposes using either a greedy search, which can be inaccurate, or an exhaustive search, which is slow. The configuration enumeration step in [16, 17] also ignores the penalty for updates, deletes, and inserts.

In this paper, we also consider recommending relational materialized views for XML data. Relational and XML data reside side by side in current database systems [5]. Query execution cost depends on the storage mode of the data, and so there are situations where it is efficient to use a relational representation of the data and others where it is more efficient to use an XML representation. The work in [21, 26] discusses the factors affecting the choice of using a relational or XML representation to store data and attempts to find a logical design for a database given the characteristics of the data to be stored in it.

Application access patterns of the data can also help in choosing how to store this data. These alternative access patterns can be exploited to add materialized views to the database to enhance query execution performance [14]. To incorporate both relational and XML data models in the same database system, several hybrid XML-relational architectures are presented in [15]. In Section 4, we study building relational materialized views as an alternative access pattern for XML data.

Using relational materialized views to answer XQuery queries requires translating XQuery queries on the XML data to SQL queries on the materialized views. In the literature, translating XQuery to SQL has traditionally taken place at the application level, where the XQuery string gets translated into an SQL string before it is sent to the database server [33]. In comparison, XQuery native compilation, described in [23] takes place inside the database server. During XQuery native compilation, an XQuery query is compiled into the server internal data structures which are shared between XQuery and SQL queries.

The main focus in [23] is to rewrite XQuery queries into SQL queries using the SQL/XML extensions provided by the Oracle DBMS. This rewriting is done during query compilation to take advantage of the powerful capabilities of the full-fledged relational query optimizer. The first phase of XQuery compilation is to parse the query into the XQueryX [25] representation. Next, static type checking, which is important for XQuery optimization, is performed. Finally, the XQuery query is rewritten to an SQL/XML query. To rewrite XQuery to SQL/XML, each XQuery expression is converted into an SQL operator or operator tree or a sub-query block. In some cases, when native compilation is not possi-

ble (i.e., a mapping between XQuery and SQL/XML is not available), a hybrid approach is taken, and a co-processor is used to handle these parts of the XQuery query. In Section 4, we take a similar but simpler approach for XQuery to SQL/XML translation. We limit ourself to a subset of XQuery that can be mapped to SQL/XML with XMLTable functions.

3 Recommending XML Indexes

Partial XML indexes are supported by commercial database systems such as DB2 and Oracle [28, 30]. Recall that a partial index is an index on parts of an XML document that match index patterns specified by the user (Section 2.1.1). Partial XML indexing leads to smaller indexes that include only the paths in a document that are relevant to user queries. This makes index maintenance on database updates more efficient and significantly improves index lookup performance over indexes that include all the paths in a document [3]. The large number of partial indexes that a user can choose makes the decision of which ones to build more difficult. In the rest of this section, we present an XML Index Advisor that automatically recommends the best set of partial XML index patterns for a given database and query workload, while taking into account the cost of updating the index on data modification.

3.1 Overview and Architecture

The architecture of the XML Index Advisor is the same as the general architecture illustrated in Figure 1. The high-level framework of the index recommendation process is as follows. First, for every query in the workload, we rely on the query optimizer to enumerate a set of candidate indexes that would be useful for it. Next, we expand the enumerated set of candidate indexes to include more general indexes, each of which can potentially benefit multiple queries from the current workload or from future, yet-unseen but related workloads. Finally, we search the space of possible index configurations to find the optimal configuration, which maximizes the performance benefit to the workload.

Much of the functionality of the advisor is implemented in a client-side application. However, we use the query optimizer for index recommendation by extending it with two new *query optimizer modes*: (1) *Enumerate XML Indexes*, in which the optimizer enumerates the indexes that can be of benefit to one input query, hence allowing us to start with a useful basic set of candidate indexes and (2) *Evaluate XML Indexes*, in which

Q1: Return a security having a specified Symbol ("BCIIPRC").

```
for $sec in SECURITY('SDOC')/Security
where $sec/Symbol= "BCIIPRC"
return $sec
```

Fig. 3 Query Q1.

Q2: List securities in a particular sector ("Energy") given a yield range (>4.5).

```
for $sec in SECURITY('SDOC')/Security[Yield>4.5]
where $sec/SecInfo/*/Sector= "Energy"
return <Security>{$sec/Name}</Security>
```

Fig. 4 Query Q2.

the optimizer simulates an index configuration and estimates the cost of a query under this configuration. These optimizer modes are the only server-side extensions required for the XML Index Advisor. They allow us to tightly couple the index recommendation process with the query optimizer, and they eliminate the need to replicate any functionality that is already available in the optimizer. Moreover, the XML Index Advisor client application is now useful for any database system that supports XML indexes, and whose optimizer is extended with our proposed modes.

In the new modes, the optimizer needs to work with hypothetical indexes that do not exist but are still needed to identify candidate indexes or evaluate their cost. To enable this, we modify the query optimizer to allow it to create *virtual indexes* that can then be used during query optimization. These virtual indexes are added to the database catalog and to all the internal data structures of the optimizer, but they are not physically created on disk and no data is inserted into them, and therefore, they cannot be used for query execution. Virtual indexes are used in relational index advisors to enable the optimizer to estimate the cost of candidate index configurations [7, 34]. In our XML Index Advisor, we use virtual indexes for cost estimation, but a novel feature of our work is that we also use them for enumerating candidate indexes for workload queries.

Next, we describe the details of the XML Index Advisor phases for recommending partial XML indexes. We use a workload consisting of the two queries Q1 and Q2 on the TPoX database, which are shown in Figures 3 and 4, respectively, as a running example.

3.2 Basic Candidate Set

XQuery and SQL/XML are complex languages. In these languages, XML patterns can appear in various parts of a query, but indexes are not useful for some of the XML

C1	/Security/Symbol	string
C2	/Security/SecInfo/*/Sector	string
C3	/Security/Yield	numerical
C4	/Security//*	string
C5	/Security/*	numerical

Table 1 Basic and general index candidates for Q1 and Q2.

patterns that appear in the query (e.g., patterns that appear in the return clause [3]). In addition, the process of deciding which indexes can benefit which patterns in a query is dependent on the XML query optimizer implementation. To obtain the basic candidate set of indexes that are useful to a given query, we tightly couple the process of generating candidate indexes in the XML Index Advisor with the process of *index matching* in the optimizer. Index matching is a fundamental process performed by query optimizers. In this process, the optimizer decides which of the available indexes can be used by the query being optimized, and how they can be used (e.g., for which predicates in the query) [4, 35].

Coupling candidate enumeration with index matching allows us to leverage the fairly elaborate query parsing, index matching, type checking, and query rewriting functionality of the query optimizer, without the need to replicate this functionality. In addition, we can support any type checks or type casts that the optimizer performs when using an index, and we can enumerate indexes that are only exposed by query rewrites in the optimizer. Moreover, we are assured that the candidate indexes considered by the Index Advisor can actually be matched and used by the optimizer.

To leverage the index matching capability of the query optimizer for enumerating candidate XML indexes, we modify the optimizer with a special Enumerate XML Indexes query optimizer mode. In this mode, we create a *virtual universal index* over the XML data, which is a virtual index whose index pattern is `//*`. This `//*` *virtual index* (virtually) indexes all elements in the document and hence can be matched with any XPath pattern in the query that can be answered using an index. Next, the query optimizer optimizes the workload query with the `//*` virtual index in place. After the index matching step of the optimizer, the optimizer returns to the user all the index patterns in the query that were matched with the `//*` virtual index.

The candidate index patterns enumerated by the optimizer take predicates into account and include indexes that are only exposed by query rewrites. For example, C1, C2, and C3 in Table 1 are the patterns enumerated by the DB2 optimizer for Q1 and Q2.

3.3 Candidate Generalization

The XML Index Advisor optimizes each workload query in Enumerate XML Indexes mode. The resulting candidate index patterns of all queries are considered as a basic candidate set. Thus, the optimizer helps us identify index patterns specific to each query. However, it is unable to identify index patterns that can benefit multiple queries in the current workload and also future queries with similar patterns. We assume that the queries that we have not seen in the input workload and would like to answer in the future have XPath expressions that are slight variants of the XPath expressions that appear in the queries of the input workload.

For example, our basic candidate set for Q1 and Q2 includes: `/Security/Symbol` and `/Security/SecInfo/*/Sector`. Therefore, the set of candidates can be expanded to include the more general pattern `/Security//*`. This new path expression covers the two original path expressions as well as other path expressions that could potentially exist in the data, such as `/Security//Industry`. Our Index Advisor can now recommend the new general index instead of the two original candidate indexes. This new candidate index will generally have a size that is greater than or equal to the total size of the two original candidate indexes, since it potentially covers more elements in the data than they do. However, it has the advantage that it can answer more queries than the two original indexes and so it can potentially be useful for queries beyond the training workload.

The candidate generalization algorithm attempts to find more generalized index patterns by iteratively applying several generalization rules to each pair of basic candidate indexes and to the resulting generalized indexes. The process continues until no new generalized XPath expressions can be found. The rules consider two XML index patterns concurrently and try to find common path nodes (representing common subexpressions) between these two patterns, which is captured in a new generalized XPath expression. We add this newly formed XPath expression to our set of candidate index patterns. Before attempting to generalize two patterns together, we check their compatibility under any other constraints, such as data type and namespace.

We represent path expression patterns as linked lists in which each node represents a path step. To generalize a pair of XML patterns, we start at the head nodes of the linked lists representing the path expressions and perform a synchronized traversal of the two lists. We examine each navigational step in the two patterns, and if a match is found, we add a matching step in the generated pattern. If an immediate match is not found,

we skip steps looking for a match and this is reflected in the generated pattern by adding * steps. We continue this procedure until we reach the indexed nodes. The details of the algorithm are presented in [8, 10].

For example, this matching process extends the candidates for Q1 and Q2 to include candidate C4 in Table 1. Candidate C3 cannot be generalized with either C1 or C2 because it is of a different data type. Therefore, we propose a heuristic approach that generalizes index patterns in the basic candidate set individually by predicting the existence of other expressions similar to a candidate. This heuristic technique replaces the last non-* navigation step in the candidate path with a * navigation step. For example, candidate C3 is generalized to C5 in Table 1 using the proposed heuristic.

3.4 Estimating the Benefit of XML Indexes

After the candidate enumeration and generalization steps, we have in hand an expanded set of candidate indexes. To find the best index configuration from these candidates, the XML Index Advisor needs to be able to estimate the benefit of an index or a set of indexes to a given workload, which we describe in this section. We also describe how we account for maintenance (update, delete, and insert) statements in the workload when estimating this benefit.

Relational index advisors leverage the query optimizer to estimate the benefit to a query workload of having a particular index configuration [7, 34]. Similarly, we employ a new query optimizer mode that we call the Evaluate XML Indexes mode. This mode relies on creating virtual indexes and estimating the cost of workload queries with these virtual indexes in place. However, we first need to collect statistics on the XML data populated in the database (e.g., using the RUNSTATS command in DB2). The optimizer in Evaluate XML Indexes mode uses these data statistics to estimate for the virtual indexes the index statistics that are necessary for the optimizer cost model (e.g., the number of leaf nodes in a B-tree). The details of the index statistics that are needed depend on the implementation of the query optimizer. The optimizer can then include the virtual indexes with other existing real indexes when performing index matching to find the possible indexes to be used in a query, and when determining a query execution plan for this query. After optimizing a query in Evaluate XML Indexes mode, the optimizer returns the set of indexes that were used, plus their index statistics and the new cost information of the evaluated query. This information is used by our index advisor to determine the benefit of using an index or a configuration consisting of multiple indexes.

3.4.1 Estimating the Benefit of an Index Configuration

In the XML Index Advisor client-side application, the benefit of using an index is estimated as the reduction in query execution cost when the index is created. If the initial cost of query q is $C_{old}(q)$ (i.e., the cost of the query when any existing indexes are in place) and the cost of the same query after creating index x is $C_{new}(q)$ (i.e., the cost of the query when the index is added to the existing configuration), the benefit of index x to query q is calculated as $Benefit(x;q) = C_{old}(q) - C_{new}(q)$. We use the Evaluate XML Indexes mode to evaluate the cost of a query when an index is in place without actually creating the index.

To evaluate the benefit of an index for a workload of queries W, we generalize the above calculation to: $Benefit(x;W) = \sum_{q \in W}(C_{old(q)} - C_{new}(q))$. Furthermore, to calculate the benefit of an index configuration, we create all the indexes in the configuration as virtual indexes and then optimize all queries in the workload in Evaluate XML Indexes mode to estimate their new costs. Thus, we have: $Benefit(x_1, x_2, \ldots, x_n; W) = \sum_{q \in W}(C_{old}(q) - C_{new}(q))$.

3.4.2 Estimating the Cost of Update, Delete, and Insert Statements and the Benefit that they Derive

Our workloads may contain update, delete, and insert (UDI) statements in addition to queries. Any index that we recommend must be maintained for each of the UDI statements in the workload. At the same time, update and delete statements may benefit from an index that helps them identify the data that needs to be updated or deleted. Such benefit is estimated just like the benefit of indexes for queries. In some database systems, such as DB2, the optimizer cost estimates do not include the cost of updating indexes because updating the indexes is an operation that has to be performed regardless of the chosen query execution plan, so ignoring the cost of this operation will not affect the plan chosen by the query optimizer. Therefore, we develop special techniques in our client-side application to estimate the maintenance cost of indexes under UDI statements.

To estimate the maintenance cost for an index x_i due to a UDI statement s, we use the data statistics to estimate the number of XML documents that have changed because of this statement, *docChanged(s)*, and the total number of elements included in this index *numElement(x_i)*. We make two simplifying assumption: (1) the number of indexed XML elements from all documents is the same and (2) all the index entries corresponding to these XML elements will need to be updated. Given the total number of XML documents in

the database *numDocs*, we can estimate the number of XML elements that the statement will affect in the index as follows:

$$elementsUpdated(x_i, s) = numElements(x_i) \times docsChanged(s)/numDocs$$

The maintenance cost mc of an index x_i because of a UDI statement s is calculated as a function of $elementsUpdated(x_i, s)$ in a way that depends on how the database system implements index updates. To account for the index maintenance cost in the benefit calculation, we subtract from the calculated benefit the maintenance cost (mc) of all indexes in the configuration. Thus, for indexes $x_1, \ldots, x_n$ and workload W that contains queries $q_1, \ldots, q_l$ and maintenance statements $s_1, \ldots, s_k$:

$$Benefit_{UDI}(x_1, \ldots, x_n; W) = \sum_{q \in W} (C_{old}(q) - C_{new}(q)) + \sum_{s \in W} ((C_{old}(s) - C_{new}(s)) - \sum_{i=1}^{n} mc(x_i, s))$$

3.4.3 Efficient Evaluation of Index Configurations

To evaluate the benefit of an index configuration, we can simply estimate the benefit of the individual indexes independently and add up these estimated benefits. However, this method ignores the *interaction* between indexes. The benefit of an index will change depending on what other indexes are available because the query optimizer can use multiple indexes in its plans. We can take index interaction into account by simply evaluating the entire workload with all indexes in the configuration created as virtual indexes. Since we evaluate the benefit of index configurations repeatedly during our search for the optimal index configuration, we have developed a more efficient approach that reduces the number of calls to the optimizer while taking index interaction into account. This approach is inspired by the atomic configuration concept described in [7].

During the generation of candidate indexes, we keep track of which workload statements produced each index x. These are the statements that can benefit from x, and we call them the *affected set* of x. To evaluate the benefit of a configuration, we only need to call the optimizer for the union of the affected sets of its indexes. Furthermore, we divide a configuration into smaller sub-configurations, where each sub-configuration includes indexes that may interact with each other, which are indexes that have overlapping affected sets. We maintain a cache of previously evaluated sub-configurations and we only evaluate a sub-configuration if it is not found in this cache. To create the set of sub-configurations for a given configuration, we start with a sub-configuration for each index, and we iteratively merge the sub-configurations whose affected sets overlap, until there can be no more merging.

For example, to evaluate the benefit of the index configuration containing C1, C2 and C3 from Table 1, we initially have each one of them in a separate sub-configuration. Because C2 and C3 are enumerated from the same query Q2, we merge their sub-configurations into {C2, C3}. To evaluate the configuration {C1}, we only need to optimize Q1 while C1 is created as virtual index. Similarly, to evaluate the configuration {C2, C3} we only need to optimize Q2 while C2 and C3 are created as virtual indexes. The benefit of the configuration {C1, C2, C3} will be the sum of the individual benefits of {C1} and {C2, C3}. When evaluating a configuration of, say, {C1, C2, C5}, we split it into the two sub-configurations, {C1} and {C2, C5}. Since {C1} was evaluated in the previous step, we only need to evaluate {C2, C5}.

3.5 Searching for the Optimal Configuration

3.5.1 Problem Definition

The XML Index Advisor needs to search the space of possible index configurations consisting of indexes from the candidate set including basic and generalized candidates to find the index configuration with the maximum benefit, subject to a constraint specified by the user on the disk space available for the chosen index configuration. This combinatorial search problem can be modeled as a 0/1 knapsack problem [34], which is NP-complete. The size of the knapsack is the disk space budget specified by the user. Each candidate index, which is an "item" that can be placed in the knapsack, has a *cost*, which is its estimated size on disk, and a *benefit*. Given that p is an index configuration in the set of candidate configurations P, W is the workload, and x is an index in p, the objective of the search problem can be described as:

$$maximize_{p \in P}\{Benefit(W, p)\}$$
$$\text{such that} \sum_{x \in p} Size(x) \leq DiskBudget$$

Modeling the index search as a 0/1 knapsack problem gives us a spectrum of solutions that ranges from greedy approximation to dynamic programming. When considering the right algorithm for the search problem, we also need to take index interaction into account. The simplest approach to solving the 0/1 knapsack problem is to use a *greedy search* that ignores index interaction. To take index interaction into account, we have added

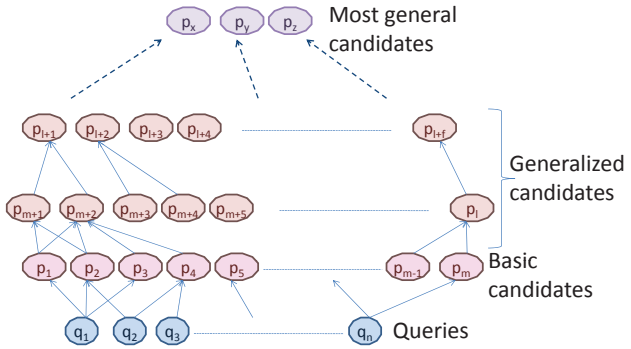


Fig. 5 Relationship between workload queries and candidate XML patterns.

some *heuristics* to the greedy search to ensure that we use as many indexes with high benefit as we can, and that they are all actually used in optimizer plans. We also propose a *top down* search that chooses as many general indexes as it can fit into the disk budget. The goals of the greedy search with heuristics and the top down search are fundamentally different: The greedy search with heuristics attempts to find the best possible set of indexes for the given workload, without any consideration for the generality of these indexes, while the top down search attempts to find configurations that are as general as possible so that they can benefit not only the given workload but also any similar future workloads.

In our search algorithms, we model the relationship between queries in the workload, the extracted XML patterns, and the generalized candidates as *directed acyclic graph (DAG)*. Figure 5 presents an example of such a DAG. For queries $q_1, q_2, \ldots q_n$ we enumerate a basic set of candidates $p_1, p_2, \ldots p_m$ as described in Section 3.2. Each query and basic candidate is represented as a node in the DAG. One basic candidate can be enumerated because of more than one query, and one query can produce more than one candidate, so we associate with each candidate the set of queries that produced it via a set of edges in the DAG. We build the next levels in the DAG by generalizing the basic candidates using the algorithm in Section 3.3, and we continue until we reach the most general candidates as shown in the figure. For each new candidate created during candidate generalization, we associate with it the set of XML patterns that were the cause of generating it through a set of edges in the DAG. Hence, by following these edges, we will have for any candidate index pattern a list of all candidates in the subtree rooted at this pattern, which we call the *coverage list*. The leaves of the subtree are the queries that can benefit from this candidate index pattern, which we call the *affected queries*. The list of affected queries of a generalized pattern is the concatenation of the lists of affected queries of its children. For example, the coverage list of C4 in Table 1 is {C1, C2, C4}, and its list of affected queries is {Q1, Q2}. Next, we present our two search algorithms.

3.5.2 Greedy Search with Heuristics

The greedy approximation of the 0/1 knapsack problem was not effective for our XML Index Advisor. The benefit of an index is highly influenced by the existence of other indexes in the configuration that can be used to answer the same query. Moreover, the greedy search can select general indexes that can be used for path expressions already covered by other indexes in the configuration. However, the query optimizer can use only one of these indexes in its query execution plan. A possible solution to this problem is to compile all workload queries after the indexes in the configuration are selected and then to eliminate indexes that are never used. The problem with this solution is that we free up extra disk space at the end of the index selection process that we never use again for adding more indexes, even though this space could be very useful. A similar approach is used for searching relational indexes in [34]. In our proposed solution for searching through candidate XML indexes, we adopt a different approach.

To address the index redundancy problem described above, we add one more objective to our search problem: maximizing the number of workload XPath expressions that use indexes in the selected configuration. Maximizing the workload benefit remains the primary objective of the greedy search algorithm that we use. Heuristics are added to the greedy search to attempt to enforce the new objective in a best effort manner.

Algorithm 1 outlines the search algorithm with the added heuristic rules. The high-level outline of the algorithm is as follows. First, we estimate the size of each candidate index and the total benefit of this index for the workload. We then sort the candidate indexes 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 (which we state later in the section), starting with the highest ratio, and we continue until the available disk space budget is exhausted. We refer to the coverage of a candidate index (*cand*) or a group of indexes (*config*) as *cand.coverage* and *config.coverage*, respectively. We also refer to the size of a candidate index (*cand*) or a group of indexes (*config*) as *cand.size* and *config.size*, respectively. We use the following functions to perform the search and apply the heuristics:

Algorithm 1 heuristicSearch(*candidates*, *diskSize*)

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

- *benefit*(*config*) returns the estimated benefit of the workload when this configuration of indexes is created as described in Section 3.4.
- *addCandIfSpaceAvail*(*cand*, *config*) adds *cand* to *config* if *cand.size* + *config.size* ≤ *diskSize*. If the condition holds, *addCandIfSpaceAvail* also updates the *size* and *coverage* of *config*.
- *replaceCandIfSpaceAvail*(*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 increase in size is below a threshold β. This is the heuristic that we add to the greedy search to deal with index interactions. The value β specifies how much increase in size we are willing to allow. We have found $\beta = 10\%$ to work well in our experiments. If the condition holds and there is enough disk space to do the replacement, *size* and *coverage* of *config* are updated.
- *overlapCoverage*(*cand*, *config*) scans *config* and returns the maximal set of candidates *overlapConfig* that has the index coverage of *cand* or part of it.

3.5.3 Top Down Search

The greedy search with heuristics attempts to recommend a configuration with the highest benefit that fits the given workload. Because of that, it can be viewed as *over-fitting* the given workload. If the workload changes even slightly, the recommended configuration may not be of use. This is acceptable if the DBA knows that the workload is not likely to change. For example, this might occur if the workload is all the queries in a particular application. However, another likely scenario is that the DBA has assembled a representative training workload, but the actual workload may be a variation on this training workload. This is true for relational data, but it is of added importance for XML, because the rich structure of XML allows users to pose queries that retrieve different paths of the data with slight variations. If this is the case, and the workload presented to the Index Advisor is a representative of a larger class of possible workloads, then we posit that the goal of the Index Advisor should be to choose a set of indexes that is as general as possible, while still benefiting the workload queries. We have developed a *top down search* algorithm to achieve this goal.

In the top down search, we use the DAG constructed during candidate generalization (e.g., the DAG shown in Figure 5). The roots of the DAG are the most general indexes that can be obtained from the workload. We start with these roots of the DAG as our current configuration. Since general indexes are typically large in size, this starting configuration is likely to exceed the available disk space budget, but it likely has a higher benefit compared to specific indexes. General indexes can have zero or negative benefit for two reasons: (1) high maintenance cost because of update, delete, and insert statements in the workload, and (2) not being used in optimizer plans. To handle this, we add a preprocessing phase to remove any indexes with zero or negative benefit from our search space. Next, we iteratively replace a general index from the current configuration with its specific (and smaller) child indexes, and we repeat this step until the configuration that we have fits within the disk space budget.

To choose the general index to replace, we introduce two metrics ΔB and ΔC. Assume that candidates $x_1, \ldots, x_n$ are generalized to a candidate $x_{general}$. There will be nodes in the DAG for each of these candidates, and $x_{general}$ will be a parent of $x_1, \ldots, x_n$. We define ΔB and ΔC as follows:

$$\Delta B = IB(x_{general}) - IB(x_1, \ldots, x_n)$$
$$\Delta C = Size(x_{general}) - \Sigma_{0 \le i \le n} Size(x_i)$$

In the previous equation, we define $IB(X)$, the *improved benefit* of the set of indexes X, as the benefit of the recommended index configuration built to this point when X is added to it.

Since our goal is to obtain the maximum total benefit for the workload by choosing the most general configuration that fits in the disk space budget, we iteratively choose the general index with the smallest $\Delta B/\Delta C$ ratio, and we replace it with its (more specific) children in the DAG. That is, we replace general indexes whose additional benefit per unit cost over their children is lowest. In case of ties, we select the index with the largest

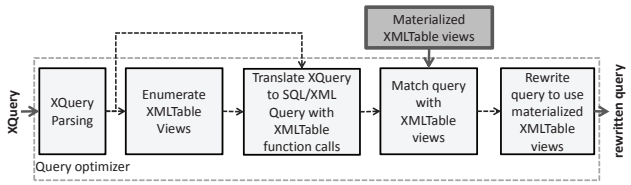


Fig. 6 Rewriting XQuery queries to use XMLTable materialized views.

ΔC. If we run out of general candidates to replace and do not yet meet the disk space budget, we use greedy search. Note that in this case, we do not need to apply our heuristics since none of the indexes we are searching through is general.

4 Recommending XML Views

In this section, we focus on enumerating and recommending XMLTable materialized views for a workload of XQuery queries.

4.1 Overview of XMLTable View Recommendation

4.1.1 Query Optimizer Architecture for Rewriting XQuery Queries to Use XMLTable Views

Our advisor recommends XMLTable materialized views for improving the performance of XQuery queries. To rewrite XQuery queries at run time to use the recommended XMLTable views, the query optimizer of a database system using our approach needs to be extended with the ability to translate XQuery queries into SQL/XML queries that use XMLTable functions. The query optimizer rewriting XQuery queries to use materialized XMLTable views runs through the following steps (Figure 6):

1. **XQuery parsing.** The XQuery query is parsed into its XML representation (XQueryX [25]) to help the query optimizer analyze the clauses of the query.
2. **XMLTable view enumeration.** We examine all the clauses in the XQueryX representation of the input query and enumerate the possible XMLTable views that include the XPath expressions that are referenced in the input query (Section 4.2).
3. **Generating a SQL/XML query that uses the enumerated views.** A new SQL/XML query that has the XMLTable views enumerated in the previous step in its FROM clause as sub-queries is created (Section 4.6).
4. **Selecting the best materialized XMLTable views that match the query.** The optimizer matches the translated SQL/XML query with all the XMLTable views materialized in the database and applies a cost-based function to select the best set of views to rewrite the query.
5. **Rewriting the query to use the selected views.** Finally, the query optimizer rewrites the query to use the set of matched XMLTable views.

Q3: 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_num>$accounts</accounts_num>
  </print>
```

Fig. 7 Query Q3.

4.1.2 XMLTable View Advisor Architecture

Our view advisor architecture follows the same general architecture described in Section 1 (Figure 1). At a high level, the goal of the XMLTable View Advisor is to identify common access patterns in the input XQuery workload and to extract the XML data accessed by these patterns into XMLTable views. For example, if the queries in the input workload frequently access the value of an element in the XML data (an ID element for instance), we extract this element as a separate column in an XMLTable view.

The class of XQuery queries that our advisor supports includes queries with FOR, LET, WHERE, and RETURN clauses. The RETURN clause can have either a simple or a constructed expression. Multiple FOR and LET clauses can occur in the query. Expressions that appear in the FOR, LET, WHERE, and RETURN clauses can have any number of predicates.

In the rest of this section, we describe the phases of the view recommendation process in detail. We use Q3, shown in Figure 7, as a running example.

4.2 Enumerating Candidate Views

XMLTable views are more complex physical structures than XML indexes, and therefore, there is no simple equivalent to the `//*` index, described in Section 3, that can be used to enumerate candidate materialized views. Because of that, we decided to develop a process for enumerating candidate XMLTable materialized views that does not rely on the query optimizer. We describe the XQuery-to-SQL/XML translation algorithm that we use to enumerate candidate views in this section. In

Algorithm 2 enumerateCandidates(*xquery*)

```
1: for clause ∈ xquery do
2:    if clause is forClause then
3:       break forClause into forVarName and forExpr
4:       view ←
         createViewFromExpr(forVarName, forExpr)
5:    else if clause is letClause then
6:       break letClause into letVarName and letExpr
7:       view ← createViewFromExpr(letVarName, letExpr)
8:       process any aggFn in letClause
9:    else if clause is whereClause then
10:      for comparisonExpr found in whereClause do
11:         for pathExpr found in comparisonExpr do
12:            find refView associated with varRef
13:            add pathExpr to refView as a column navigator
14:         end for
15:      end for
16:   else if clause is returnClause then
17:      for pathExpr found in returnClause do
18:         find refView associated with varRef
19:         add pathExpr to refView as a column navigator
20:      end for
21:   end if
22: end for
```

Algorithm 3 createViewFromExpr(*varName*, *expr*)

```
1: create a new view view and associate it with the variable
   name varName
2: break expr into pathExpr and predicateList
3: if pathExpr has a variable reference varRef then
4:    find refView associated with varRef
5:    view.rowGen ← refView.rowGen + pathExpr
6:    add column "." to refView and a column with back-
      ward navigation path "refCol" to view
7: else
8:    set the row generator of view to be pathExpr
9: end if
10: for p ∈ predicateList do
11:    for pathExpr found in p do
12:       add pathExpr to view as a column navigator
13:    end for
14: end for
    return view
```

Section 4.6, we revise this algorithm to enable translating XQuery queries into SQL/XMLTable queries that use the enumerated materialized views. The class of XQuery queries that our advisor support is described in [8].

To enumerate candidate views for an XQuery query, we parse the query and break it down into its FOR, LET, WHERE, and RETURN clauses. We further break each of these clauses into its components. The FOR and LET clauses in an XQuery query are used to produce a tuple stream in which each tuple consists of one or more bound variables. This behavior resembles the row generator in the XMLTable function (recall Section 2.1.2). Therefore, for every FOR or LET clause in the input XQuery, we create a new candidate XMLTable view. We describe next how we handle each clause in the candidate enumeration process (Algorithm 2 and the helper function described in Algorithm 3).

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
```

Fig. 8 Final version of V1.

FOR Clause. We divide the FOR clause into a variable, the path expression associated with the variable (the binding sequence for that variable), and the optional predicates (Algorithm 2, Line 3). A FOR clause produces a tuple stream for every variable and iterates over the binding sequence of that variable, which resembles the functionality of the row generator of the XMLTable function. Therefore, for every FOR clause: (1) we create a new candidate materialized view and assign its row generator to be the binding sequence in the FOR clause (i.e., the path expression after removing any predicate values from it, for example, `/Customer` in the FOR clause of Q3), (2) we record the variable name and the created view, and finally (3) for every predicate expression appearing in the binding sequence of the FOR clause, we add it as a column navigator path expression to the view. For example, when we parse the FOR clause of Q3, we create a view V1 (Figure 8) that has the row generator `/Customer` and the column `@id`. Algorithm 3 illustrates the procedure of creating a view from the path expression that appears in a FOR clause.

LET Clause. Similar to the FOR clause, a LET clause produces a tuple stream for every variable declared in it. Unlike the FOR clause, a LET clause binds each variable declared in it to the result of its associated expression without iteration and hence we need to compensate for this behavior. First, we create a new candidate XMLTable view with the binding expression of the LET clause after removing any predicates from it as its row generator. Next, to compensate for the non-iterative behavior of the LET clause, we add column navigator with the `"."` expression to the generated view to represent all the tuples generated by the row generator of the view and then group all of these tuples using a GROUP BY clause (Algorithm 2, Lines 6–8).

For a binding sequence that references another variable (e.g., the expression `$cust/Accounts/Account` in Q3), we look up the expression referenced by this variable (`$cust` references `/Customer` in the FOR clause, which is also associated with the already generated view V1) and concatenate it with the rest of the ex-

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  xml path 'parent::Accounts/parent::Customer') as u
group by cy1
```

Fig. 9 View V2 after parsing the LET clause in Q3.

pression to form the path expression that we use as a row generator when creating the XMLTable view (`/Customer/Accounts/Account` is used as the row generator for V2, the materialized view generated from the LET clause of Q3 in this example). We then add a column in each of the views: (1) a column in the newly generated view (V2) to backward navigate the row generator of the view that represents the referenced variable in the binding sequence (the column `parent::Accounts/parent::Customer` in V2 references `/Customer` in V1), and (2) a `"."` column in the referenced view (V1) (Figure 8). These columns are used for joining the two views in the translated query. Additionally, a LET clause might have an optional aggregation function that we handle by adding the aggregation of the `"."` column to the SELECT clause of the XMLTable view (`count(u.cy0)` in V2). The generated view V2 is shown in Figure 9.

WHERE Clause. For every predicate appearing in a WHERE clause, we extract the XPath expressions appearing in this predicate. For each XPath expression, we look up the view (*refView*) associated with the referenced variable (*varRef*) in this expression and add a column to that view to correspond to this navigation (Algorithm 2, Lines 10–15). For example, to account for the predicate on `age` in Q3, we add a column navigator in view V1 (Figure 8).

RETURN Clause. For all the XPath path expressions that appear in the RETURN clause, we find all the views that are associated with the reference variables that appear in these expressions and we then add a column for each expression to the corresponding view (Algorithm 2, Lines 17–20). For example, the expression `$accounts` in the RETURN clause of Q3 references an existing column in V2 and hence no change is needed to the view. However, for the expression `$cust/name`, we add the column `name` to V1. The final version of view V1 is shown in Figure 8.

4.3 Generalizing the Set of Enumerated Views

Recall that the XML Index Advisor generalizes the index patterns to make them useful for queries not seen in the input workload that is used for recommendation. Similarly, creating XMLTable views that answer multiple queries in the workload and potential unseen queries can increase the usefulness of our recommendations. Since our proposed view definition involves both XPath expressions and SQL query definitions, generalization can benefit from the index generalization techniques proposed in Section 3 and the query merging techniques proposed in [36]. We describe the forms of query generalization that we use in our View Advisor in this section. The XMLTable View Advisor applies these generalization rules to the basic set of candidate views to generate an expanded set of candidate views.

V4.
```
select u.cx0, u.cx1 from CUSTACC, xmltable(
  '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
  columns
    cx0  double path '@id' ,
    cx1  varchar(100) path 'occupation') as u
```

Fig. 10 View V4.

4.3.1 Generalizing Column Navigators to Include Subtrees

Most of the XMLTable views that are generated in the candidate enumeration phase are a normalization (flattening) of values that are accessed in the workload queries. An alternative approach is to recommend views that store sub-trees of the data as XML columns. For example, we can generalize V1 (Figure 8) into V3 that has `/Customer` as a row generator and `"."` as the only column navigator. The `"."` column navigator means that all the subtrees reachable by the row generator are stored in the materialized view. This approach is useful when the query requires reconstructing the XML tree.

4.3.2 Merging Views

A common generalization approach used in relational advisors is view merging [36]. For XMLTable views, we merge views that have the same row generator to produce a new view that has as its column navigators 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 without affecting performance by combining views. This approach is a special case of the approach discussed in Section 4.3.1, since we keep the normalization state (flat or nested) of the column navigator. For example, view V5 (Figure 11) is a merging of V1 (Figure 8) and V4 (Figure 10).

```
V5.
select u.cx0, u.cx1, u.cx2, u.cx3 , u.cx4 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',
    cx4  varchar(100) path 'occupation') as u
```

Fig. 11 Generalized view V5.

4.4 Relational Indexes on XMLTable Views

One approach to make XMLTable views more useful is to build relational indexes on their columns. This is possible since the XMLTable views are regular relational tables with indexable columns that happen to originate from XML data. There can be many possible indexes that can be built on the 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 appear in a predicate in the XQuery that caused this view to be recommended. This guarantees 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 build for view V1 is index I1 (`create index I1 on V1(cx0, cx2)`). For every candidate view, we add to the search space another alternative physical structure that consists of the view with a relational index on its columns.

4.5 Searching for the Optimal View Configuration

After applying the generalization rules on the basic candidate views generated from the input queries, we obtain an expanded set of candidates. To choose some of these enumerated XMLTable views (a view configuration) to recommend for a workload, we search the space of enumerated candidate views to find the best set of views that fits in a given disk space budget. We use the same search algorithms presented in Section 3.5. The top down search algorithm described in Section 3.5.3 can be used without any changes for searching the candidate XMLTable views. However, we made minor changes to the greedy search with heuristics algorithm described in Section 3.5.2.

XMLTable views can interact with each other in ways that affect their total benefit for a query workload. 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 and therefore subsume them. These interactions are similar to the ones encountered when searching the space of XML indexes. We use the search algorithm in Section 3.5.2 unmodified except for the definition of candidate coverage. We define the *view coverage* of a view as its view ID as well as the IDs of the views that it subsumes (i.e., the views that it was generated from using the generalization rules, and the views that have the same row generator and column navigators that were enumerated for other queries). The coverage of a configuration of views is defined as the union of the view coverage of its constituent views. For example, if V5 is generated by merging V1 and V4, then the coverage of V5 is the set {V1, V4, V5}.

4.6 Translating XQuery Queries into SQL Queries that Use XMLTable Views

At run time, the query optimizer needs to translate input XQuery queries to SQL queries with XMLTable functions to be able to match these queries with the XMLTable views. Translation of XQuery queries to SQL queries with XMLTable functions during query compilation is studied in [23]. We adopt a similar approach that we describe in this section. The translation involves using XMLTable views that are similar to the ones being enumerated for the XQuery queries using Algorithm 2 (Section 4.2). This ensures that the XMLTable views in the translated XQuery queries will match the recommended XMLTable views.

During the XQuery to SQL translation, we examine the parsed XQuery, generate XMLTable views that encapsulate all referenced XPath expressions in the query, and then construct an SQL query based on this information. We add all the generated views to the FROM clause of the SQL query. We then construct the SELECT and WHERE clauses in the translated query by referring to the columns of the views to reflect how their associated expressions appear in the original query. We also add joins between the views that are used to rewrite the query when needed. These joins are needed to link two FOR or LET clauses where one references the other to make sure that the data referenced by both clauses in any iteration is the same. For example, the binding sequence of the LET clause in Q3 (`$cust/Accounts/Account`) references the binding sequence of the FOR clause (`/Customer`). Therefore, we add an equality predicate (i.e., a join) for the expressions represented by `$cust` referenced in the FOR and LET clauses to make sure that the XML data is the same in any iteration (i.e., we are aggregating the accounts of the same customer in any iteration).

Table 2 lists the lines that we add to Algorithms 2 and 3 in order to get the algorithms for

translateXQuery(*xquery*)	
lineNo	Added code
L0a	*selectElementsList* $\leftarrow \phi$
L0b	*fromViewsList* $\leftarrow \phi$
L0c	*wherePredicatesList* $\leftarrow \phi$
L4a	add *view* to *fromViewsList*
L8a	add *view* to *fromViewsList*
L14a	add *comparisonExpr* to *wherePredicatesList*
L20a	construct return value *returnVal*
L20b	add *returnVal* to *selectElementsList*
L22a	generateQuery(*selectElementsList*, *fromViewsList*, *wherePredicatesList*)
translateXQueryAndCreateViewFromExpr(*varName*, *expr*)	
lineNo	Added code
L6a	construct predicate *joinPred* to join columns "." in *refView* and "*refCol*" in *view*
L6b	add predicate *joinPred* to *wherePredicatesList*
L13a	add predicate *p* to *wherePredicatesList*

Table 2 Extensions made to Algorithms 2 and 3 for XQuery translation.

translating XQuery queries. The new versions of the algorithms are translateXQuery(*xquery*) and translateXQueryAndCreateViewFromExpr(*varName*, *expr*). The main goal of these extensions is to build the three lists *selectElementsList*, *fromViewsList*, and *wherePredicatesList* that we use to construct the translated query. First, we modify Algorithm 2 by inserting the code listed in the three rows L0a, L0b, and L0c of the table before Line 1 of Algorithm 2 to initialize the three lists. For every FOR or LET clause in the query, we record the views that we create by adding them to *fromViewsList*. Thus, we insert entries L4a and L8a of the table after Lines 4 and 8 in Algorithm 2, respectively. For every predicate, we encounter during the parsing either in an expression appearing in a FOR or LET clause or in a WHERE clause, we add a reference to it in the *wherePredicatesList* (Table entry L13a to be inserted after Line 13 in Algorithm 3, and Table entry L14a to be inserted after Line 14 in Algorithm 2). When a binding sequence references a previously defined variable, we interpret this occurrence as a join between the referenced view and the new view. The columns needed for this join are illustrated in entries L6a and L6b, which we insert after Line 6 in Algorithm 3. Finally, we call the function *generateQuery* to construct the translated query from the three lists *selectElementsList*, *fromViewsList*, and *wherePredicatesList* (We insert table entry L22a in Algorithm 2). The *generateQuery* function uses a template of an SQL query with SELECT, FROM, and WHERE clauses to construct the translated query as follows: (1) simple elements or XML constructs in *selectElementsList* are added to the SELECT clause of the query, (2) references to views in *fromViewsList* are added to the FROM clause of the query, and (3) all predicates in *wherePredicatesList* are added to the WHERE clause. If the return value is a simple XPath expression, then the corresponding column name is used, otherwise we use the XMLElement SQL function to construct an XML fragment.

Translated Query: SQ3.

```
select XMLElement( NAME "print" ,
         XMLElement( NAME "name", Vv0.c3) ,
         XMLElement( NAME "accounts_num", Vv1.ACc1))
from
 (select v0.c0, v0.c1, v0.c2, v0.c3
  from CUSTACC, xmltable(
     '$rowVar/Customer' passing CUSTACC as "rowVar"
  columns
     c0 double path '@id' ,
     c1 xml path '.' ,
     c2 double path 'age',
     c3 varchar(100) path 'name') as v0 ) as Vv0,
 (select count(v1.c0) as ACc1 , v1.c1
  from CUSTACC, xmltable(
     '$rowVar/Customer/Accounts/Account'
      passing CUSTACC as "rowVar"
  columns
     c0 xml path '.',
     c1 xml path 'parent::Accounts/parent::Customer') as v1
  group by v1.c1 ) as Vv1
where ( Vv0.c0 > 9000 ) and ( Vv1.c1 = Vv0.c1 ) and ( Vv0.c2 > 50 )
```

Fig. 12 Translated query SQ3.

To illustrate our translation process, we show the final translated query for Q3 (Figure 7) in Figure 12. The two views V1 (Figure 8) and V2 (Figure 9) are recommended for query Q3, so we construct the FROM clause in the translated query as `from V1, V2`. Next, we examine the return clause and construct the SELECT clause of the rewritten query. 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.

5 Integrated Recommendation of Indexes and Materialized Views

In this section, we integrate the index and view advisors described in Sections 3 and 4 into one Integrated Index-View Advisor that recommends both XML indexes and XMLTable views for a workload of XQuery queries. The Integrated Index-View Advisor ensures that the recommended configuration satisfies the given disk space constraint.

5.1 Motivation: The Need for an Integrated Index-View Advisor

XMLTable materialized views are considered alternative relational access paths to the XML data in the

Q4: Return order IDs whose OrdStatus is equal to P.

```
for $ord in doc("ORDER.ODOC")
      /Order[OrdStatus = "P"]
return $ord/@ID
```

Fig. 13 Query Q4.

V6: View on the ORDER table that contains the ID and OrdStatus values for all the order documents stored in the table.

```
select u.cx0, u.cx1
from ORDER, xmltable(
  '$odoc/Order' passing ORDER.ODOC as "odoc"
  columns
    cx0  varchar(100) path '@ID',
    cx1  varchar(100) path 'OrdStatus') as u
```

Fig. 14 View V6 that can be used to answer Q4.

RQ4: A rewritten version of query Q4 that uses view V6.

```
select V1.cx0
from V1
where V1.cx1 = "P"
```

Fig. 15 Rewritten query Q4 that uses view V6.

database. It has been shown in the literature that XMLTable materialized views can reduce the execution time of queries [24] and also database maintenance statements [22]. However, they can grow as large as the data, and the query execution plans that use them might not be better than the query execution plans without them. In contrast, partial XML indexes are usually smaller in size and can drastically reduce the execution time of queries. In this section, we show that XMLTable views are especially useful for certain types of queries. We also show that XML indexes are not useful all the time. Therefore, it is beneficial to include both XML indexes and XMLTable views in one unified search space when recommending physical designs, as our Integrated Index-View Advisor does. We study the usefulness of XML indexes and XMLTable views to query execution plans by comparing these different plans. We highlight three usage patterns for indexes and views: pre-navigation, joining tables, and aggregation.

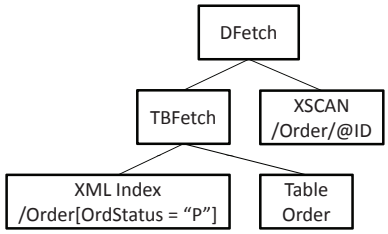
Pre-navigation: Pre-navigation to the XML elements that are needed during query execution and storing them in a format that is easily accessible can save a huge amount of query execution time. The XMLTable function allows pre-navigation and stores the resulting pre-navigated values in a relational table format. By using XMLTable functions, we create new relational views of some of the fragments of the XML data that are accessed by the queries in the workload. Therefore, we can now translate complex XQuery queries into simple select statements. XML indexes are also useful in navigating to the nodes (or their values) referenced in the query. To evaluate the benefit of pre-navigation, we compare three optimizer query plan alternatives for query Q4 (Figure 13): (1) the execution plan when indexes are used, (2) the execution plan when XMLTable views are used, and (3) the execution plan when XMLTable views and relational indexes on them are used. In these plans, we use the following abbreviations: (1) **DFetch:** refers to fetching a document from an XML column, (2) **XSCAN:** refers to scanning an XML document, which consequently means parsing or navigating an XML document depending on how the XML data are stored in the database, (3) **TBFetch:** refers to fetching specific rows in the table, and (4) **TBSCAN:** refers to scanning an entire table to examine its rows.

Figure 16 shows three possible query execution plans for Q4.[2] In a typical query execution plan when no physical structures are used, all the documents in the table are read and scanned to find the qualifying predicate(s) and the return value(s). The total cost of this plan equals the cost of navigating all documents in the table. To reduce the execution cost, there are three alternatives:

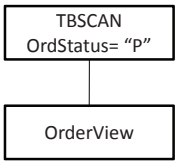
1. When we use an XML index (for example, an index that includes the XML nodes that are reachable by the XPath expression `/Order/OrdStatus`) to select the XML subtrees rooted by nodes that satisfy the predicate(s) in the query, we need to navigate to these subtrees to find the return value(s). In this execution plan, the execution cost is equal to the sum of the index navigation cost and the navigation cost of the selected documents (Figure 16(a)).
2. When we use an XMLTABLE view such as V6 (Figure 14), the execution plan for the rewritten query that uses this view (RQ4 shown in Figure 15) includes scanning all the rows of the view to find qualifying tuples. The cost of this execution plan is equal to the cost of scanning the entire view (Figure 16(b)).
3. When we use an XMLTABLE view and a relational index on the columns that represent all predicates in the original XQuery (for example, an index on the column `cx1` in view V6), the cost of executing the plan is equal to the sum of the index navigation cost and the cost of fetching the qualified tuples (Figure 16(c)).

Depending on the structure of the XML documents and the selectivity of the predicates in the query, various situations will lead to different possible plans having the lowest cost. Therefore, we rely on estimated

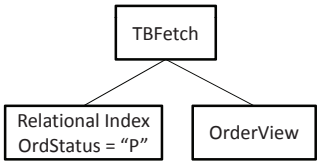
[2] We generated these query execution plans using DB2. XQuery queries used as examples in this section are simple queries, and most database systems would generate similar execution plans for them.



(a) XML indexes.

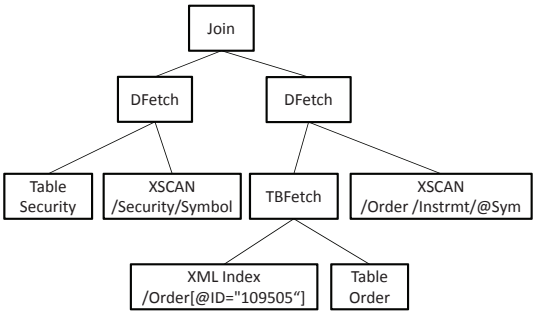


(b) XMLTABLE views.

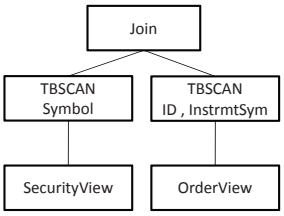


(c) XMLTABLE views and relational indexes on them.

Fig. 16 Query execution plans for query Q4.



(a) XML indexes.



(b) XMLTABLE views.

Fig. 17 Query execution plans for query Q5.

Q5: Return current open price of a particular order.

```
for $ord in doc("ORDER.ODOC")/Order[@ID="109505"]
for $sec in doc("SECURITY.SDOC")
    /Security[Symbol=$ord/Instrmt/@Sym]
return <ret> {...} <ret>
```

Fig. 18 Query Q5.

costs to decide which of the physical structures can best benefit a specific workload of queries and XML data.

Joining Tables: Joining tables is a common and important operation in XQuery queries. The execution cost of XQuery queries with table joins can be reduced by pre-navigating the values to be joined, storing them in relational tables, and then joining these relational tables. For example, we show in Figure 17 the execution plans for Query Q5 (Figure 18), which has a join between tables `ORDER` and `SECURITY`. Figures 17(a) and 17(b) show the execution plans for query Q5 using XML indexes and XMLTABLE views, respectively. The number of elements in the join operator's two inputs is the same in both execution plans. The total execution cost can be lower in the execution plan with XMLTABLE views because of the following: (1) the relational optimizer can use a larger variety of join methods, hash joins for example and (2) the table scan of the XMLTable materialized view is cheaper than scanning a table with XML documents stored in one of its columns. In the latter case, it is necessary to parse and navigate the XML documents during the scan. To demonstrate this, we executed Q5 after rewriting it to use XML indexes and XMLTable materialized views as illustrated in the execution plans shown in Figure 17. The execution time of Q5 when rewritten to use XMLTable views was 17 times faster than its execution time when rewritten to use XML indexes.

Aggregation: Another type of queries that can benefit from using XMLTable views are queries with grouping and aggregation functions. In addition to the benefit of pre-navigation, pre-grouping and/or pre-aggregating the data in an XMLTable view reduces query execution time. This can be done only with views and not with indexes.

Having qualitatively described and contrasted the benefit of XML indexes and XMLTable views, we now compare the execution time of queries when creating the recommendations of the XML Index Advisor and the XMLTable View Advisor for a large space budget (2 GB). The setup for this experiment is described in Section 6.1. Figure 19 shows the estimated execution times of queries in the TPoX workload for the following three cases: (1) no physical structures are used, (2) XML indexes recommended by the XML Index Advisor

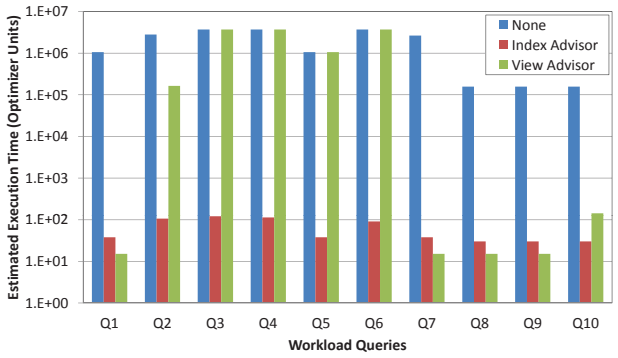


Fig. 19 Estimated execution time per query for advisor recommendations.

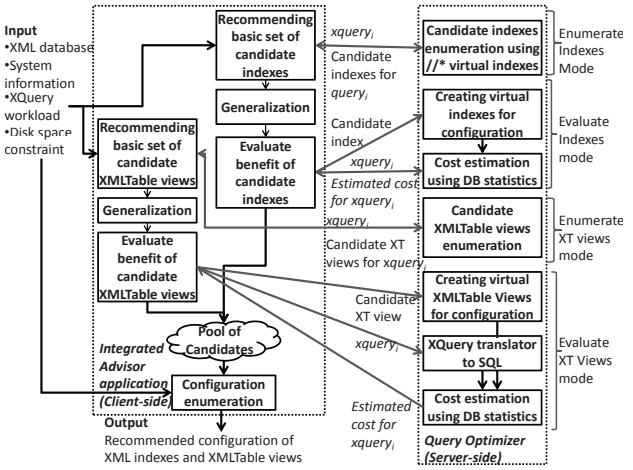


Fig. 20 The Integrated Index-View Advisor architecture.

are created, and (3) XMLTABLE views recommended by the XMLTable View Advisor are created. In each case, the configuration recommended by the advisor is created, and then, the optimizer is invoked in a special mode to estimate the execution time of the queries in the workload. We note that we could not use views with four TPoX queries, Q3-Q6 (see Section 6 for details). We observe that the execution times of four out of the remaining six queries of the TPoX workload when rewritten to use XMLTable views are less than the execution times of these queries when rewritten to use XML indexes. However, the benefit/size ratios of the XMLTable views used in rewriting these queries are less than the benefit/size ratios of indexes used for rewriting the same queries because views usually have a larger size compared to indexes. We also note that the execution times of queries Q7-Q9, which extract data from the three different tables of the TPoX database, can be reduced by half through the use of XMLTable views instead of XML indexes.

From the above comparison, we conclude that both XML indexes and XMLTable views are useful for different queries. It would be difficult to inspect each query to decide whether to recommend XML indexes or XMLTable views for it. Furthermore, some queries can benefit from both indexes and views, and our rewriting algorithms restrict us to using one type of physical structure for each query. More complications arise when searching the space of candidate XML indexes and XMLTable views due to considering relational indexes on XMLTable views. Therefore, it is beneficial to consider XML indexes and XMLTable views together as one search space when recommending a physical design for an XML workload. In the rest of this section, we present an Integrated Index-View Advisor that recommends the best configuration of XML indexes and XMLTable views for a given XML database and XQuery workload.

5.2 Design of the Integrated Index-View Advisor

Figure 20 shows the architecture of our integrated advisor. Since the candidate enumeration processes for XML indexes and XMLTable materialized views are different, we enumerate and generalize candidates of each type separately using the candidate enumeration and generalization algorithms described in Sections 3 and 4. This results in candidates of three types: XML indexes, XMLTable materialized views, and XMLTable materialized views with relational indexes on them. We combine all these candidates into one pool of candidates, and we search for the best configuration among all these candidates. The search algorithm is different from the search algorithms in Sections 3 and 4 because the space of candidates contains different types of physical structures, which introduces new types of interactions. In the next section, we generalize the search algorithms described in Sections 3 and 4 to an algorithm that considers different types of interactions between different physical structures.

5.3 Searching Indexes and Views Together

The search algorithms that we have proposed in Sections 3 and 4 take into account two types of interactions between candidates: (1) interaction between candidates that can be used to rewrite the same query and (2) candidates where one is a general form of the other. While the former type of interaction affects the benefit of candidates due to the existence of other candidates, the latter type of interaction poses a restriction that at most one candidate is to be chosen. When searching the combined space of XML indexes and XMLTable views, we also consider that a query can either be rewritten to use XML indexes or XMLTable views, but not both,

Algorithm 4 integratedSearch(*candidates*, *diskSize*)

```
1: sort candidates according to their benefit/size ratio
2: recommended ← ∅, recommended.size ← 0,
   recommended.coverage ← ∅
3: while recommended.size < diskSize do
4:    bestCand ← pick the next best cand in candidates
5:    if recommended.coverage ∩ bestCand.coverage = ∅
      then
6:       addCandIfSpaceAvail(bestCand,recommended)
7:    else if recommended.coverage ∩ bestCand.coverage
      ≠ ∅ then
8:       overlapConfig ←
         overlapQCoverage(bestCand, recommended)
9:       if bestCand is XINDEX then
10:         replaceConfig ← {cand | cand ∈ overlapConfig
            and (isGeneral(bestCand, cand) or cand is
            XVIEW or cand is XVIEW_RINDEX)}
11:      else if bestCand is XVIEW then
12:         replaceConfig ← {cand | cand ∈ overlapConfig
            and (isGeneral(bestCand, cand) or cand is
            XINDEX or (cand is XVIEW_RINDEX and
            (isGeneral(bestCand, cand.view) or bestCand =
            cand.view)))}
13:      else if bestCand is XVIEW_RINDEX then
14:         replaceConfig ← {cand | cand ∈ overlapConfig
            and (isGeneral(bestCand, cand) or cand
            is XINDEX or (cand is XVIEW
            and (isGeneral(bestCand.view, cand) or
            bestCand.view = cand)))}
15:      end if
16:      if replaceConfig = ∅ then
17:         addCandIfSpaceAvail(bestCand, recommended)
18:      else
19:         replaceCandIfSpaceAvail(bestCand,
            replaceConfig, recommended)
20:      end if
21:   end if
22: end while
```

because of their different rewriting algorithms. The notion of candidate coverage is not valid any more, as there is no clear relation between the XML indexes and the XMLTable views that can be used for the same query, and we also want to consider using either type of structure for each query. We choose to define the coverage in the integrated search algorithm based on query coverage, and we introduce new rules to handle special cases.

The high-level outline of the search algorithm is similar to the algorithm we use to search the space of indexes (Section 3.5.2) and the algorithm we use to search the space of views (Section 4.5), with different rules for the various types of candidates. Algorithm 4 presents the integrated search algorithm. The first step of the search algorithm is to sort all of the physical structures according to their benefit/size ratio. We then iteratively consider candidate physical structures: XML indexes (XINDEX), XMLTable views (XVIEW), and XMLTable views with relational indexes on them (XVIEW_RINDEX) and try to add them to the set of recommended structures (*recommended*). In every iteration, if the recommended set of candidates is empty or the candidate that we are considering in this iteration (*bestCand*) adds new coverage (i.e., it helps a query in the workload that is not yet helped by any of the structures already selected by the advisor), we add *bestCand* to our set of recommended physical structures if enough disk space is available. Otherwise, if there is overlap between the queries that are already covered by physical structures in the *recommended* configuration and the coverage of *bestCand*, we apply the heuristic rules that we describe next to decide whether to add *bestCand* to our set of recommended physical structures or not. First, we use the helper function *overlapQCoverage* to find the set of physical structures in the *recommended* configuration that help some or all of the queries that are covered by *bestCand*. We call this set of candidates the *overlapConfig*. We then apply the following rules depending on the type of *bestCand*:

1. *bestCand* **is an XML index:** We build an alternate configuration (*replaceConfig*) consisting of the set of physical structures {*cand*} that belong to the set *overlapConfig* and that satisfy one of the following conditions:
   (a) *cand* is an XML index, and *bestCand* is a general form of it. In this case, *bestCand* can replace *cand* in its query execution plans.
   (b) *cand* is an XMLTable view or XMLTable view with a relational index on it. In this case, choosing an XMLTable view to answer a query in the workload means that we cannot use XML indexes for rewriting it, because the query rewriting algorithm can use either XML indexes or XMLTable views to rewrite a given query, but not both.
2. *bestCand* **is an XMLTable view:** We build an alternate configuration (*replaceConfig*) consisting of the set of physical structures {*cand*} that belong to the set *overlapConfig* and that satisfy one of the following conditions:
   (a) *cand* is an XMLTable view, and *bestCand* is a general form of it. In this case, *bestCand* can replace *cand* in its query execution plans.
   (b) *cand* is an XML index. Hence, we either choose the XML index that is already selected (*cand*) or the new XMLTable view that we are currently considering (*bestCand*).
   (c) *cand* is an XMLTable view with a relational index on it. Whether *bestCand* is the same as *cand.view* or is a general form of it, we add *cand* to *replaceConfig*.

3. *bestCand* **is an XMLTable view with a relational index on it:** We build an alternate configuration (*replaceConfig*) consisting of the set of physical structures {*cand*} that belong to the set *overlapConfig* and that satisfy one of the following conditions:
   (a) *cand* is an XMLTable view with a relational index on it, and *bestCand.view* is a general form of *cand.view*. In this case, *bestCand* can replace *cand* in its query execution plans.
   (b) *cand* is an XML index. Hence, we either choose the XML index that is already selected (*cand*) or the new XMLTable view with a relational index on it that we are currently considering (*bestCand*).
   (c) *cand* is an XMLTable view. Whether *bestCand.view* is the same as *cand* or is a general form of it, we add *cand* to *replaceConfig*.

The next step in the algorithm is to check the alternate configuration *replaceConfig*. If it is empty, this means that *bestCand* can be used together with already selected physical structures to answer queries in the workload and that we can safely add *bestCand* to the *recommended* configuration if there is enough disk space. Otherwise, we check the following two configurations: (1) $bestCand \cup (recommended - replaceConfig)$: the configuration that includes *bestCand* in addition to the structures that we have already selected after removing the ones in *replaceConfig* from it, and (2) *recommended*. If the new configuration ($bestCand \cup (recommended - replaceConfig)$) has a higher benefit and its size does not exceed the disk space constraint, we make it the *recommended* configuration.

In Algorithm 4, we use the following helper functions:

- *isGeneral*(*cand1*, *cand2*): returns *true* if *cand1* is of the same type as *cand2* and is a general form of *cand2*. If *cand1* and *cand2* are XMLTable views with relational indexes on them, we compare *cand1.view* and *cand2.view*.
- *addCandIfSpaceAvail*(*cand*,*config*): adds the candidate physical structure *cand* to the configuration of physical structures *config* if the size of the new configuration is within the disk space constraint.
- *replaceCandIfSpaceAvail*(*cand*,*replaceConfig*,*config*): replaces the set of structures in *replaceConfig*, which is a subset of the configuration *config*, with the candidate structure *cand* if the new configuration has a higher benefit than the old configuration, and the size of the new configuration does not exceed the disk space constraint. The new configuration is: $cand \cup (config - replaceConfig)$.

6 Experimental Evaluation

6.1 Experimental Setup

We conducted our experiments on a Dell PowerEdge 2850 server with two Intel Xeon 2.8 GHz CPUs (with hyperthreading) and 4 GB of memory running SuSE Linux 10. The database is stored on a 146 GB 10 K RPM SCSI drive. We implemented our advisors in a prototype version of IBM DB2 V9.7, which we use for our experimental evaluation.

We use the TPoX [27] benchmark in our experiments. TPoX is an XML benchmark based on a financial application. We run the experiments on two TPoX data sets generated using scale factors of 1 and 10 GB. We present experiments run on the 10 GB database unless otherwise stated. We evaluate our XML Index Advisor on the standard queries that are part of the benchmark specification, 11 XQuery queries. To illustrate the effectiveness of our generalization algorithm, we also use synthetic queries on the TPoX data in Section 6.2.2.

Our metric for evaluating the recommendations of the XML advisors is *estimated speedup*: The estimated execution time by the query optimizer of the workload with no XML physical structures divided by the estimated execution time of the workload with the configuration of physical structures recommended by the XML advisors. In Sections 6.3 and 6.4, we report the estimated query execution time in optimizer units (called *timerons* in DB2). Optimizer units give a better presentation of the differences between the advisor recommendations in these sections. We first present an evaluation of the XML Index Advisor, then the XML View Advisor, and finally the Integrated Index-View Advisor. More experimental results can be found in [8].

6.2 Effectiveness of the XML Index Advisor

In this section, we illustrate that our XML Index Advisor makes good index recommendations that effectively use the available disk space budget and that are useful beyond the training workload.

6.2.1 Effectiveness of the Advisor Recommendations

We have implemented four different combinatorial search strategies in our Index Advisor: (1) greedy search (without heuristics), (2) greedy search with heuristics, (3) top down, and (5) dynamic programming. With the exception of greedy search and dynamic programming, which are standard combinatorial search techniques, these strategies are described in Section 3.5. In our first

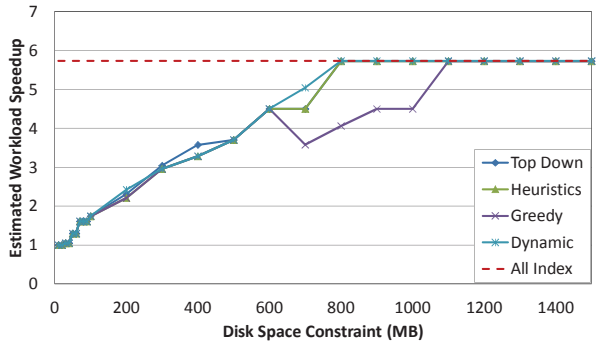


Fig. 21 Estimated workload speedup with index recommendations.

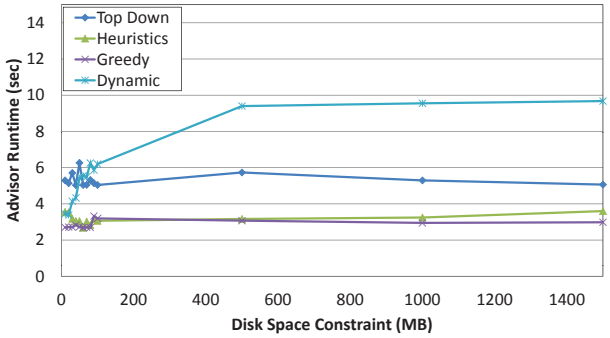


Fig. 22 Index Advisor run time with different search strategies.

experiment, we compare the index recommendations of these four strategies. In this experiment, we only vary the search algorithm used by the XML Index Advisor, all the algorithms used for the other recommendation phases are the same. In our implementation, we allow the user to input a hint to the advisor to replace XML patterns that contain `//*`, which might be undesired patterns, with their children during the search for the best configuration.

Figure 21 shows the estimated speedup for the search strategies when varying disk space budgets. The workload that we use for this experiment consists of the 11 queries of the TPoX benchmark and one more query. The added query is similar to one of the TPoX queries that joins customers and orders after varying the XPath expressions in its predicates to allow more general indexes to be generated. The best speedup that can be achieved in this experiment is 5.7. This speedup is achieved by the *All Index* configuration shown in the figure, which has an index for every indexable XPath expression in the query workload. The size of this configuration is 880 MB. In this experiment, we use the benchmark queries for recommending the indexes and also for evaluating the recommendations. Every run, we create the recommended configuration of indexes as virtual indexes. We then use the explain mode of the optimizer to estimate the execution time of the queries of the workload with these indexes in place.

Figure 21 shows that our XML Index Advisor is able to recommend indexes that speed up workload execution for the TPoX workload at all disk space budgets. As expected, speedup increases as we increase the available disk space budget, until it reaches the best possible speedup of the All Index configuration. Greedy search requires significantly more disk space than the All Index configuration to match its performance. The reason is that greedy search often chooses multiple indexes that answer the same query, thereby wasting some of the available disk space budget without gaining any benefit. The heuristics we use with greedy search are designed to avoid such errors, and their effectiveness can be seen from the figure. The top down, greedy with heuristics, and dynamic programming search algorithms result in similar speedups except for small variations that are due to selecting different indexes with similar benefits. However, these algorithms take different times to find the recommended configuration as we show next.

Figure 22 shows the run time of the Index Advisor with different search strategies for various disk space budgets. Top down search takes up to 2 times longer than greedy search with heuristics. However, the run time of top down search improves as the available disk space increases because it needs to explore fewer nodes in the DAG of candidate indexes before arriving at a configuration that fits within the disk space budget. The run time of greedy search is lower than all the other search algorithms and is not affected by changing the disk budget because it checks every candidate index at most once. Adding heuristic rules to the greedy search does not have a significant effect on the run time of the advisor. The run time of dynamic programming increases exponentially with increasing the disk space budget. Thus, we can see that the recommendations of the dynamic programming search algorithm, which are sometimes better than the recommendations of other approximate search algorithms, come at a significantly high cost.

6.2.2 Recommending Generalized Indexes

In this section, we demonstrate that our Index Advisor can recommend indexes that are more general than the candidates generated from the workload, and that these indexes can benefit future queries different from those in the training workload. This is a key feature of our Index Advisor.

The first question we address is how many generalized indexes can potentially be found in a workload. In

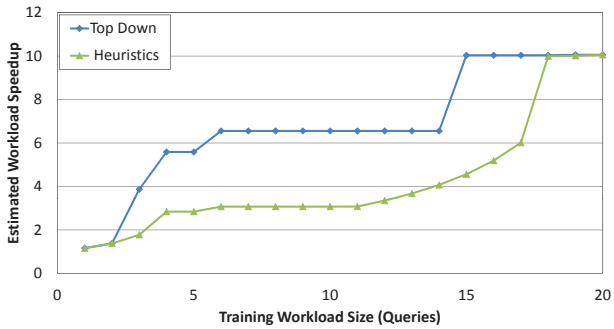


Fig. 23 Index Advisor generalization to unseen queries.

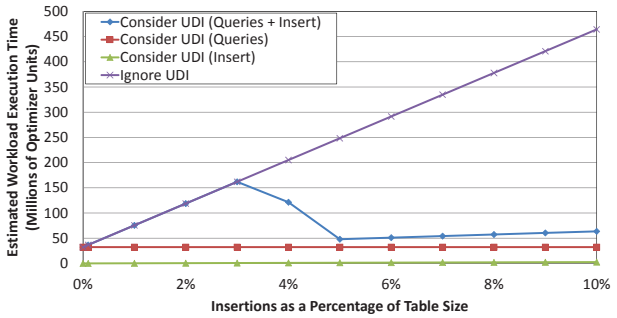


Fig. 24 Effect of updates on index recommendations. Disk budget = 100MB.

the TPoX workload that we used in the previous experiment, we were able to increase the number of candidate indexes by 30 % through candidate generalization.

To show the effect of recommending general indexes on the speedup of various workloads, we perform an experiment where the training workload used by the Index Advisor for recommending indexes is different from the test workload used to evaluate the recommended configuration. We used a workload of 20 queries, the 11 TPoX queries followed by 9 synthetic queries to increase workload diversity. The synthetic queries where generated by using three of the TPoX queries as templates. These query templates represent different return value complexities and touch the three XML tables in the TPoX database. In the synthetic queries, we used random XPath path expressions that occur in the data to replace the XPath expressions in the original queries. We train (i.e., recommend configurations) based on n queries, and we test based on the entire workload, and we vary n from 1 to the number of queries (20). Figure 23 shows the estimated speedup on the test workload as we vary the training workload size with a disk space budget of 20 GB (effectively an unbounded disk space budget). A training workload with size n is the same as the training workload with size $n-1$ after adding one additional query to it. The figures show the speedup for top down search and greedy search with heuristics. The figures show that as the advisor sees more and more of the test workload, it can recommend a configuration of indexes that can be useful to unseen queries. The figures also show that top down search is quite effective at using the available disk space to generalize from the queries seen in the training workload to the unseen queries in the test workload, whereas greedy search with heuristics is unable to perform such generalization.

Figures 23 shows the results of the experiment when we added TPoX queries to the training workload in one order. To confirm that the conclusions are not affected by the order of the queries, we repeated the above experiment for different query orders and found that the conclusions hold for all query orders that we tried.

6.2.3 Evaluating Candidate Configurations

The quality of the configurations recommended by the XML Index Advisor depends on the accuracy of estimating the benefit of candidate index configurations in the Evaluate XML Indexes optimizer mode and the penalty of UDI statements.

The key statistic used by Evaluate XML Indexes mode is the size of a virtual index. We have found that for the TPoX workload, the median relative estimation error for this statistic is 6 and 12 % for the 10 and 1 GB data, respectively . Notably, we are able to estimate the size of large indexes, which have the most impact on performance, with a very small error. For example, the largest candidate indexes for TPoX are indexes on `/FIXML/Order/OrdQty/@*` and `/FIXML/Order//@*`, and we are able to estimate their size with 3.7 and 5.5 % error, respectively.

Figure 24 illustrates the effect of estimating the penalty of updating candidate index configurations in response to UDI statements. We add to the TPoX workload a varying number of UDI statements that insert documents into one of the tables (the `ORDER` table), and we use the Index Advisor to recommend a configuration with a 20 GB disk space budget. The figure shows the estimated execution time as we vary the number of UDI statements. The figure shows the case where the Index Advisor ignores UDI statements while recommending an index configuration, and for the case where it takes UDI statements into account. The figure also shows the cost of the queries and insert statements individually for the latter case. As the number of UDI statements increases, workload execution time increases in all cases, but the advisor that takes into account UDI statements is able to reduce the increase in execution time by dropping indexes when the penalty for updating them exceeds their benefit (which happens when insertions are

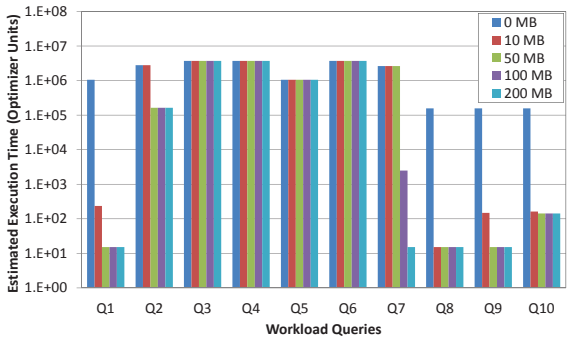


Fig. 25 Estimated query execution times for the recommended XMLTable views.

around 3 % of the table size). Thus, the Index Advisor can effectively estimate the benefit of indexes even in the presence of updates.

6.3 Effectiveness of the XMLTable View Advisor

In this section, we present an evaluation of the XMLTable View Advisor. We also show the effectiveness of the two approaches that we use to expand the views search space: (1) merging XMLTable views and (2) creating relational indexes on XMLTable views. In this section and the next, we use the TPoX data with scale factor 1 GB.

Figure 25 shows the estimated execution time of each query for a configuration with no views and the recommended view configurations with different disk space budgets. The maximum estimated speedup that can be achieved, when we create all the recommended views for the TPoX workload (total size 115 MB), is only 1.6 because some queries in the workload (Q3–Q6) did not benefit from the views. Queries Q1, Q2, Q7, Q8, Q9, and Q10, which range from simple navigation to join queries, benefit from the recommended XMLTable views. Even for a configuration size as small as 10MB, the average speedup for these queries that benefit from views is 134. This shows that XMLTable views can be useful for many query types, and that our XMLTable View Advisor is quite effective at recommending these views.

6.3.1 Recommending Merged XMLTable Views

To evaluate the performance of generalized views, we compare two configurations: (1) *Basic*, which contains all the views enumerated for the queries in the workload, and (2) *Generalized*, which contains a new set of generalized views generated by merging views in the first configuration using the generalization rules presented in Section 4.3.2. In this experiment, we only use the queries that can be helped by at least one view from one of the two configurations (i.e., queries Q3-Q6 in Figure 25 are omitted). We measured the actual execution time of all queries in the workload after materializing each configuration, and we report results based on these measurements in Table 3[3].

Configuration	Size (MB)	Speedup	Benefit/size ratio
Basic	58.2	354.6	5.3
Generalized	48.8	198.0	6.3

Table 3 Effect of merging views on performance.

Table 3 shows that 16% of the total size of the configuration is saved by merging views. The measured speedup with the generalized configuration is lower than the speedup with the basic configuration. However, the benefit/size ratio achieved is higher for the generalized configuration. From this we conclude that using generalized views reduces the execution cost of queries, but the benefit is lower than using basic views that only contain data referenced by the queries. However, if we also consider the reduction in disk size needed to create the generalized configuration, the merged views are a more efficient alternative.

6.3.2 Recommending XMLTable Views with Relational Indexes

In this section, we investigate the benefit of building relational indexes on XMLTable views. For this experiment, we let the advisor choose a configuration consisting only of XMLTable views (with no relational indexes on them) in one case. In the other case, we let the advisor choose a configuration from a set of candidates consisting of XMLTable views and XMLTable views with relational indexes on them. The disk space budget was 2 GB in both cases (effectively unbounded). Figure 26 shows actual execution time in both these cases, and when there are no views. We omit the queries that do not benefit from XMLTable views from the figure. In all the shown queries, using relational indexes over the XMLTable views reduces the execution time of the queries in the workload. The speedup achieved due to using relational indexes (compared to using views with no indexes) ranges from 1.5 to 32.5 per query. This demonstrates the effectiveness of our approach for recommending relational indexes on the XMLTable views.

[3] More results based on actual execution time can be found in [8, 10, 11].

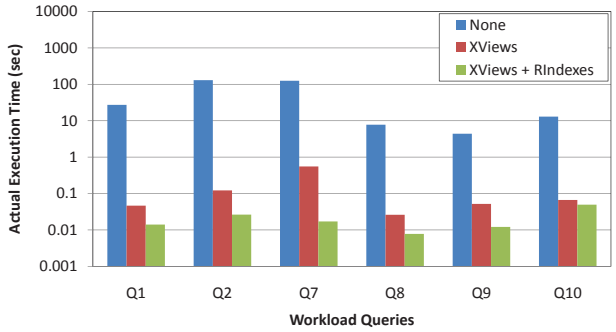


Fig. 26 Actual query execution times for the recommended XMLTable views and XMLTable views with relational indexes on them.

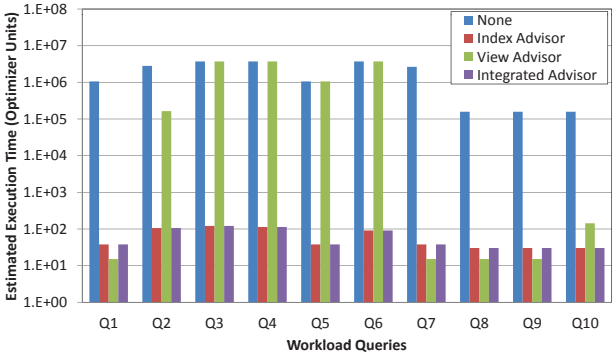


Fig. 27 Estimated query execution times for advisor recommendations. Disk budget = 400MB.

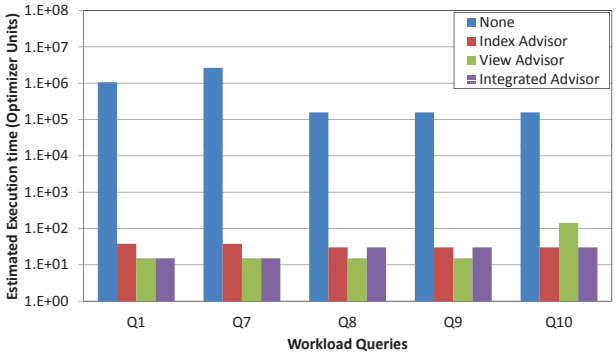


Fig. 28 Estimated query execution times for advisor recommendations. Disk budget = 400MB.

6.4 Effectiveness of the Integrated Index-View Advisor

In this experiment, we compare the recommendations of the XML Index Advisor, the XMLTable View Advisor, and the Integrated Index-View Advisor. Figure 27 is an updated version of Figure 19 (shown in Section 5) after adding one more column to represent the estimated execution times of the queries in the workload when the recommendations of the Integrated Index-View Advisor are materialized in the database. We observe that the Integrated Index-View Advisor always chooses XML indexes for all queries in this experiment even though indexes might have lower benefit than the candidate views for the same query. This could be due to one of the following reasons: (1) indexes have much smaller sizes and hence their benefit/size ratio are higher or (2) an index can be useful to other queries in the workload while the materialized view is only useful to one query, so the benefit of the index to the entire workload is higher than the benefit of the materialized views recommended for each of the queries helped by this index. For example, for query Q1 in the TPoX workload, the Integrated Index-View Advisor recommends an XML index for it although selecting an XMLTable view is expected to result in a lower execution time. This can be explained as follows: Query Q1 benefits from building an XML index I1, which is also useful for queries Q4 and Q5. Query Q1 also benefits from building an XMLTable view V1. The estimated benefit of the XML index I1 when calculated for the entire workload is higher than the estimated benefit of the XMLTable view V1. Hence, the candidate XML index is chosen by the search algorithm.

To eliminate the effect of this type of interaction, we compare the recommendations of the three advisors when the input workload is composed of queries Q1 and Q7–Q10. The results are shown in Figure 28. In this figure, we can see that the advisor sometimes recommends indexes and it sometimes recommends views. The Integrated Index-View Advisor selects the candidate structures that lower the execution time of individual queries when these structures also lower the execution time of the entire workload.

These experiments demonstrate that our Integrated Index-View Advisor effectively recommends suitable physical designs for XML workloads. This integrated advisor puts together all the contributions of this paper into one tool that can be used by DBAs of XML databases.

7 Conclusion

We presented physical design tools that automatically recommend XML indexes and XMLTable materialized views for XML databases. We first described an XML Index Advisor that recommends the best set of indexes for a given XML database and query workload and that is tightly coupled with the query optimizer, using the optimizer for both enumerating and evaluating indexes. We then described an XMLTable View Advisor that recommends relational materialized views (XMLTable views) for XQuery workloads. Finally, we analyzed the different benefits that XML indexes and XMLTable views can provide to various types of XQuery queries,

and we concluded that both of them are useful and that they benefit different queries with different degrees. We presented an Integrated Index-View Advisor that searches for the best physical design for a workload in a pool of candidate physical structures that contains XML indexes and XMLTable views. We have implemented our proposed advisors in a prototype version of IBM DB2. Our experiments with this implementation show that our advisors can effectively recommend physical designs that result in significant speedups for workload queries.

Acknowledgements We would like to thank Kevin Beyer, Andrey Balmin, and Fei Chiang for their contributions to the earlier stage of this work [10]. This work was supported by the IBM Center for Advanced Studies.

References

1. Agrawal, S., Chaudhuri, S., Kollár, L., Marathe, A.P., Narasayya, V.R., Syamala, M.: Database tuning advisor for Microsoft SQL Server 2005. In: VLDB (2004)
2. Arion, A., Benzaken, V., Manolescu, I., Papakonstantinou, Y.: Structured materialized views for XML queries. In: VLDB (2007)
3. Balmin, A., Beyer, K.S., Özcan, F., Nicola, M.: On the path to efficient XML queries. In: VLDB (2006)
4. Balmin, A., Özcan, F., Beyer, K., Cochrane, R.J., Pirahesh, H.: A framework for using materialized XPath views in XML query processing. In: VLDB (2004)
5. Beyer, K., et al.: DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. IBM Systems Journal **45**(2), 271–298 (2006)
6. Bohannon, P., Freire, J., Haritsa, J.R., Ramanath, M.: LegoDB: Customizing relational storage for XML documents. In: VLDB (2002)
7. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for Microsoft SQL Server. In: VLDB (1997)
8. Elghandour, I.: Automatic physical design for xml databases. Ph.D. thesis, University of Waterloo (2010)
9. Elghandour, I., Aboulnaga, A., Zilio, D.C., Chiang, F., Balmin, A., Beyer, K., Zuzarte, C.: An XML index advisor for DB2 (demonstration). In: SIGMOD (2008)
10. Elghandour, I., Aboulnaga, A., Zilio, D.C., Chiang, F., Balmin, A., Beyer, K., Zuzarte, C.: XML index recommendation with tight optimizer coupling. In: ICDE (2008)
11. Elghandour, I., Aboulnaga, A., Zilio, D.C., Zuzarte, C.: Recommending XMLTable views for XQuery workloads. In: XSym (2009)
12. Godfrey, P., Gryz, J., Hoppe, A., Ma, W., Zuzarte, C.: Query rewrites with views for XML in DB2. In: ICDE (2009)
13. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In: VLDB (1997)
14. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal **10**(4), 270–294 (2001)
15. Halverson, A., Josifovski, V., Lohman, G.M., Pirahesh, H., Mörschel, M.: ROX: Relational over XML. In: VLDB (2004)
16. Hammerschmidt, B.C., Kempa, M., Linnemann, V.: A selective key-oriented XML index for the index selection problem in XDBMS. In: DEXA (2004)
17. Hammerschmidt, B.C., Kempa, M., Linnemann, V.: Autonomous index optimization in XML databases. In: SMDB (2005)
18. IBM Corp.: IBM DB2 Database for Linux, UNIX, and Windows Information Center (2006). Available at: `http://publib.boulder.ibm.com/infocenter/db2luw/v9/`
19. Josifovski, V., Massmann, S., Naumann, F.: Super-Fast XML wrapper generation in DB2: A demonstration. In: ICDE (2003)
20. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: SIGMOD (2002)
21. Lapis, G.: XML and relational storage – Are they mutually exclusive? In: Proc. Conf. on XML, the Web and beyond (XTech) (2005)
22. Liu, Z.H., Chang, H.J., Sthanikam, B.: Efficient support of XQuery Update Facility in XML enabled RDBMS. In: ICDE (2012)
23. Liu, Z.H., Krishnaprasad, M., Arora, V.: Native XQuery processing in Oracle XMLDB. In: SIGMOD (2005)
24. Liu, Z.H., Krishnaprasad, M., Chang, H.J., Arora, V.: XMLTable index an efficient way of indexing and querying XML property data. In: ICDE (2007)
25. Melton, J., Muralidhar, S.: XML syntax for XQuery 1.0 (XQueryX). W3C Recommendation (2007). Available at: `http://www.w3.org/TR/xqueryx`
26. Moro, M.M., Lim, L., Chang, Y.C.: Schema advisor for hybrid relational-XML DBMS. In: SIGMOD (2007)
27. Nicola, M., Kogan, I., Schiefer, B.: An XML transaction processing benchmark. In: SIGMOD (2007). Benchmark Available at: `https://sourceforge.net/projects/tpox/`
28. Nicola, M., Van der Linden, B.: Native XML support in DB2 Universal Database. In: VLDB (2005)
29. Onose, N., Deutsch, A., Papakonstantinou, Y., Curtmola, E.: Rewriting nested XML queries using nested views. In: SIGMOD (2006)
30. Oracle Corp.: Oracle Database 11g Release 1 XML DB Developer's Guide (2007). Available at: `http://www.oracle.com/pls/db111/homepage`
31. Runapongsa, K., Patel, J.M., Bordawekar, R., Padmanabhan, S.: XIST: An XML index selection tool. In: XSym (2004)
32. Schmidt, K., Härder, T.: On the use of query-driven XML auto-indexing. In: SMDB (2010)
33. Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered XML using a relational database system. In: SIGMOD (2002)
34. Valentin, G., Zuliani, M., Zilio, D.C., Lohman, G., Skelley, A.: DB2 advisor: An optimizer smart enough to recommend its own indexes. In: ICDE (2000)
35. Xu, W., Özsoyoglu, Z.M.: Rewriting XPath queries using materialized views. In: VLDB (2005)
36. Zilio, D.C., Rao, J., Lightstone, S., Lohman, G.M., Storm, A., Garcia-Arellano, C., Fadden, S.: DB2 design advisor: Integrated automatic physical database design. In: VLDB (2004)