XML Index Recommendation with Tight Optimizer Coupling

Iman Elghandour [1*], Ashraf Aboulnaga [1], Daniel C. Zilio [2], Fei Chiang [3†],
Andrey Balmin [4], Kevin Beyer [4], Calisto Zuzarte [2]

[1] *University of Waterloo*
{ielghand,ashraf}@cs.uwaterloo.ca

[2] *IBM Toronto Lab*
{zilio,calisto}@ca.ibm.com

[3] *University of Toronto*
fchiang@cs.toronto.edu

[4] *IBM Almaden Research Center*
{abalmin,kbeyer}@us.ibm.com

***Abstract*— XML database systems are expected to handle increasingly complex queries over increasingly large and highly structured XML databases. An important problem that needs to be solved for these systems is how to choose the best set of indexes for a given workload. In this paper, we present an XML Index Advisor that solves this XML index recommendation problem and has the key characteristic of being tightly coupled with the query optimizer. We rely on the optimizer to enumerate index candidates and to estimate the benefit gained from potential index configurations. We expand the set of candidate indexes obtained from the query optimizer to include more general indexes that can be useful for queries other than those in the training workload. To recommend an index configuration, we introduce two new search algorithms. The first algorithm finds the best set of indexes for the specific training workload, and the second algorithm finds a general set of indexes that can benefit the training workload as well as other similar workloads. We have implemented our XML Index Advisor in a prototype version of IBM® DB2® 9, which supports both relational and XML data, and we experimentally demonstrate the effectiveness of our advisor using this implementation.**

I. Introduction

There are currently several native XML database systems [1], [2], and XML support has also been added to most commercial relational database systems [3], [4], [5]. All these systems employ various types of structural and value XML indexes to improve performance, potentially by orders of magnitude.

Users of XML database systems now face the problem of deciding on the set of indexes to create for a given XML database and query workload. This is of particular importance for XML database systems that allow for *partial indexing* of XML documents. A partial index is an index on parts of an XML document that match *index patterns* specified by the user. These patterns can be represented, for example, by XPath path expressions, in which case only the XML elements that are reachable by these path expressions are included in the index [4], [6]. 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 [7]. Partial indexes are supported in database systems such as DB2 9 [6], and Oracle 11g [4]. However, users now face the problem of choosing the set of XML patterns to index. In this paper, we present an XML Index Advisor that addresses this problem by automatically recommending the best set of XML index patterns for a given database and query workload while taking into account the cost of updating the index on data modification.

Recommending indexes as part of the physical database design process has previously been studied extensively in the context of relational databases, and most commercial database systems now include "Index Advisors" that automatically recommend indexes [8], [9]. The high-level outline of the index recommendation process for XML databases is similar to that for relational databases. However, recommending indexes for XML databases presents some unique challenges that make the problem more difficult than the relational case, and that lead to the details of the solutions being significantly different.

The challenges for XML index recommendation stem from the richness of XML query languages and the potential complexity of the structure of XML data. XPath supports wildcards and descendant navigation, and XML elements can be recursive. Thus, for any query, there can be several potentially useful indexes and index patterns. For example, the XPath query `/Security[Yield>4.5]` can benefit from a value index on the index patterns `/Security/Yield`, `/Security/*` or `//Yield`[1]. The rich structure of XML also leads to an exponential increase in the number of candidate index configurations that need to be searched to find

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

†This work was done while the author was at the IBM Toronto Lab.

[1] Throughout this paper, we use examples from the TPoX benchmark [10].

the optimal configuration, which places additional importance on the search algorithm used and makes it important to try to minimize the number of optimizer calls to evaluate the benefit of index configurations.

One of the key features of our Index Advisor is that it is tightly coupled with the query optimizer of the XML database system. We rely on the query optimizer to enumerate the candidate index patterns for a query and to evaluate the benefit to a query of having a particular index configuration. This tight coupling with the query optimizer helps us leverage its index selection and cost estimation capabilities and provides a solid and easy way for ensuring that the indexes that we recommend are actually *used* by the optimizer in the query execution plans that it generates. Moreover, we can easily support the different query languages supported by the optimizer. For example, our XML Index Advisor implementation in DB2 supports both XQuery and SQL/XML simply by virtue of the fact that the DB2 query optimizer supports both of these languages. Developing an Index Advisor independent of the query optimizer entails emulating – outside of the optimizer – the parsing, access path selection, and cost estimation steps performed by the optimizer. This emulation involves a significant amount of work and creates the possibility of having inconsistencies between the Index Advisor and query optimizer, which can lead the advisor to recommend indexes that are never used by the optimizer.

The rest of the paper is organized as follows. We present related work in Section II. Section III presents our framework for index recommendation. Next, we present our contributions, which can be summarized as follows:

- An algorithm for enumerating candidate XML indexes for a query that leverages the index matching capabilities of the query optimizer. These indexes are the basis of our space of index configurations (Section IV).
- A generalization algorithm that expands the set of candidate indexes by deriving new candidates from existing ones, such that the derived candidates can benefit multiple queries in the current workload and also similar queries in future workloads (Section V).
- Two novel algorithms for searching the space of possible index configurations to find the best one that fits within the available disk budget. The first algorithm is based on greedy search augmented with heuristics that maximize the number of queries in the workload that use the selected indexes. The second algorithm has the objective of selecting as many general indexes as possible to fit in the disk space budget (Section VI).
- A technique to reduce the number of calls to the optimizer for evaluating the benefit of candidate configurations (Section VI-C).
- An implementation of the XML Index Advisor in a prototype version of DB2 and an experimental study using the TPoX benchmark [10] (Section VII).

II. Related Work

Several XML indexing schemes have been proposed, and many of these schemes allow partial indexing of XML documents and so would benefit from an XML Index Advisor to help in selecting index patterns [4], [11], [12], [13], [14]. In the past few years, there has been a significant amount of work on index advisors for relational databases [8], [9], [15], [16]. Unfortunately, none of these works extends directly to XML databases.

A few attempts were made to recommend indexes for XML data that is shredded and stored in relational databases [17], [18]. In [17], the proposed approach focuses on a specific type of structural index that can be used over relational databases. The proposed solution cannot be generalized to other types of database systems and the proposed cost model is independent of the database system which can lead to inaccurate estimates. In [18], a new approach is proposed that considers the interplay of logical and physical design when shredding XML data into relational databases. The physical design targets relational database systems and so cannot be adopted in database systems that store XML data natively.

Two recent works have made preliminary attempts to tackle the index recommendation problem for XML databases [19], [20]. 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 optimizer will use the recommended indexes and no guarantee that the benefits of candidate index configurations are estimated with any accuracy. In addition, neither of these works tackles the issue of generalizing the initial set of candidates, which is equivalent to merging physical design structures in relational databases [15]. We address these shortcomings and we also propose a configuration enumeration algorithm that takes into account the interaction between indexes yet is efficient in the number of optimizer calls it makes.

In [20], 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. The candidate indexes used in [20] are the paths that occur in the data with some grouping of structurally equivalent candidate indexes based on schema information if this information is available. This method is inefficient because it leads to an uncontrolled explosion of the space to search for the best set of indexes. Furthermore, the candidate generation process does not attempt to generate candidates that are useful for multiple queries.

Another index recommender for XML is presented in [19], [21]. This index recommender analyzes the workload periodically and creates or drops XML indexes on the fly. As in [20], the cost model used is independent of the query optimizer and hence likely to be inaccurate. Candidate enumeration is not described. For configuration enumeration, [21] proposes using

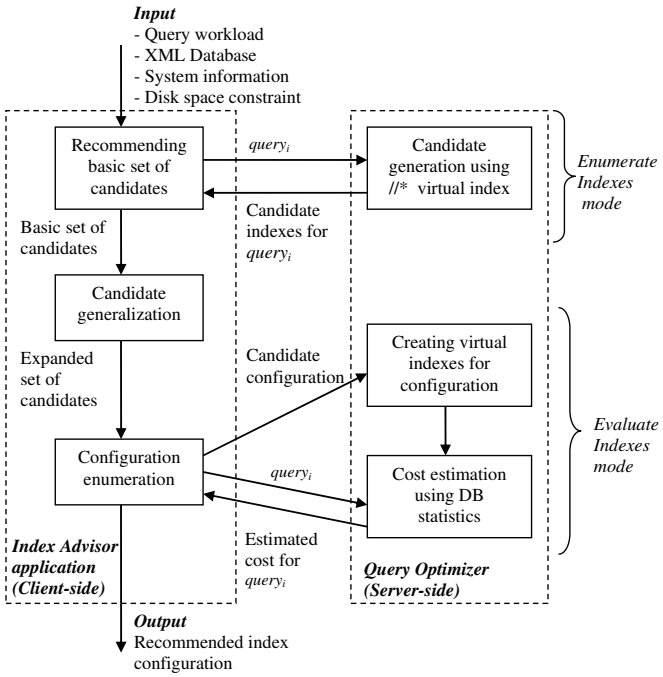

Fig. 1. XML Index Advisor architecture.

either a greedy search, which can be inaccurate, or an exhaustive search, which is slow. The configuration enumeration step in [19], [21] also ignores the penalty for data modification.

III. Overview and Architecture

XML query languages use XPath path expressions to retrieve elements in the data. XML indexes can be categorized into *structural indexes* that speed up navigation through the hierarchical structure of the XML data (e.g., [22]), and *value indexes* that help in retrieving XML elements based on some condition on the values they contain (e.g., [4], [13]). In both cases, the index includes pointers to XML elements that are reachable via specific *index patterns* [6], typically expressed as XPath path expressions. Value indexes also include the values of indexed elements.

In this paper, we focus on indexes that are represented by index patterns expressed as *linear XPath path expressions* that do not include predicates. For example, we would consider an XML index with index pattern `/Security/Yield`, which includes all values of `Yield` and so can be used to answer a query like `/Security[Yield >= 4.5]`. Indexes with linear XPath index patterns are an important class of indexes, analogous to single-column indexes in the relational case. It is important to point out that while the *index patterns* enumerated by the optimizer contains no predicates (Section IV), the XPath expressions in our *query workload* can contain predicates at arbitrary locations.

The architecture of the XML Index Advisor is presented 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 this particular query. 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 workloads. Finally, we search the space of possible index configurations to find the optimal configuration, which is the one that maximizes the performance benefit to the workload while satisfying the disk space constraint provided by the user.

Much of the functionality of the advisor is implemented in a client-side application. However, to be able to use the query optimizer for index recommendation, we need to extend it with two new *query optimizer modes*. In the first mode, which we call the *Enumerate Indexes* mode, the optimizer takes a query and enumerates the indexes that can help this query, hence enabling us to start with a basic set of candidate indexes known to be useful. In the second mode, which we call the *Evaluate Indexes* mode, 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.

In the new modes, the optimizer needs to work with hypothetical indexes that do not exist, but are still needed to accomplish its task. 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. The virtual indexes cannot be used for query execution, and so they are only created in the special query optimizer modes, where the goal is not to generate query execution plans. Virtual indexes are used in relational index advisors to enable the optimizer to estimate the cost of candidate index configurations [8], [9]. 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.

To estimate the benefit of an index configuration, we first create all the indexes in this configuration as virtual indexes. We then invoke the optimizer in the Evaluate Indexes mode for each statement (query or update/delete/insert) in the workload to estimate its cost while these virtual indexes are in place. Subtracting the new cost of a workload statement (s_{new}) from its original cost (s_{old}), we get the benefit to this statement due to the configuration. The benefit of each unique statement in the workload is multiplied by its frequency of occurrence in the workload, $freq_s$. The total benefit to all statements is the benefit of the index configuration.

The XML Index Advisor architecture allows us to rely completely on the query optimizer for cost estimation, leveraging its tuned, well-developed cost model. It is beyond the scope of this paper to discuss XML cost models, an active area of research in its own right. Moreover, a detailed description of the cost model of the DB2 optimizer, which we use in our prototype, can be found in [23]. To take advantage of the

cost estimation capabilities of the query optimizer, we need to provide it with accurate statistics on our virtual indexes for use by its cost model. To do that, we use the statistics collection command of the database system (`RUNSTATS` in DB2) to collect data statistics for the XML data. We then derive the required index statistics (index size, number of levels, etc.) for the virtual indexes from these data statistics.

Our XML Index Advisor takes into account the cost of updating indexes in response to update, delete, and insert statements. If the cost of updating indexes is included in the optimizer cost estimates of these statements, no special processing is required for them. In some database systems, such as DB2, the optimizer cost estimates do not include the cost of updating indexes. In this case, to account for the index maintenance cost, we subtract from the calculated benefit the maintenance cost mc of all indexes in the configuration. This cost is only calculated for update, delete and insert statements, and is equal to zero for query statements. We describe our cost model for mc, as well as our derivation of index statistics from data statistics, in [24]. Thus, for indexes $x_1, \ldots, x_n$ and workload W, we have:

$$Benefit(x_1, x_2, \ldots, x_n; W) = \sum_{s \in W} (freq_s * (s_{old} - s_{new}) - \sum_{i=1}^{n} mc(x_i, s))$$

In the rest of the paper, we use as a running example, a workload consisting of the following two queries from the TPoX benchmark [10].

Q1: Return a security having the specified Symbol

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

Q2: List securities in a particular sector given a yield range

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

IV. Basic Candidate Set

XQuery and SQL/XML are fairly complex languages. In these languages, XML patterns can appear in different parts of the statement, but indexes cannot be used for some of them [7]. In addition, the process of deciding on which indexes can benefit which patterns in the 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) [25], [26], [27].

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. Adding our proposed index enumeration mode to the query optimizer of any database system would allow our Index Advisor to recommend usable indexes by this system.

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

TABLE I
BASIC AND GENERALIZED CANDIDATES.

To leverage the index matching capability of the query optimizer for enumerating candidate XML indexes, we modify the optimizer to create a special Enumerate 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 an XML 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, we collect all the index patterns in the query that were matched with the `//*` virtual index, and we terminate the optimization process. Essentially, we have enabled the optimizer to answer the question: "If all possible indexes were available, which rewritten query patterns would benefit from them?"

The candidate index patterns enumerated by the optimizer will have already taken predicates into account and will 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 our example queries, Q1 and Q2. C1 and C2 are only exposed by query rewrites of Q1 and Q2, respectively. All three candidates take predicates into account to determine the target nodes of the index patterns.

The XML Index Advisor optimizes each workload query in Enumerate Indexes mode. The resulting candidate index patterns of all queries are considered as a basic candidate set that is expanded in the generalization step, which we describe next.

V. Generalizing the Candidates

The optimizer helps us identify linear 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. To address this shortcoming of relying on the optimizer for candidate enumeration, we expand the set of candidates generated by the optimizer by applying a set of *generalization rules*. These rules allow us to generate more general candidate indexes that can be useful for multiple queries from the specific index patterns enumerated by the optimizer for individual queries.

For example, in queries Q1 and Q2 the following two XPath path expressions are identified by the query optimizer as candidates for indexing: `/Security/Symbol` and `/Security/SecInfo/*/Sector`. Based on these two path expressions, we expand the set of candidates 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`. This more general candidate index is a new alternative that can be recommended by our Index Advisor 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. But this new general index 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 XML index patterns can be found. The rules consider two XPath expressions concurrently and try to find common path nodes (representing common subexpressions) between these two paths. This commonality is captured in a newly formed, generalized XPath expression. We add this newly formed XPath expression to our set of candidates. Before attempting to generalize two patterns together, we check their compatibility under any other constraints, such as data type and namespace. During the generalization of a pair of expressions, we divide each path into two parts: the last step, which represents the nodes we are indexing, and the steps leading to this last step.

Algorithm 1 generalizeStep($genXPath$, p_i, p_j)

```
1: if (isLast(p_i) and !isLast(p_j)) or (!isLast(p_i) and isLast(p_j)) then
2:     return {advanceStep(genXPath, p_i, p_j)}
3: end if
4: create newNode
5: if p_i.nameTest = p_j.nameTest then
6:     newNode.nameTest = p_i.nameTest
7: else
8:     newNode.nameTest = "*"
9: end if
10: newNode.axis = genAxis(p_i.axis , p_j.axis )
11: append newNode to genXPath
12: return {advanceStep(genXPath, p_i, p_j)}
```

We represent path expression patterns as linked lists in which each node represents a path step. Our pair generalization process is divided into two functions: *generalizeStep* (Algorithm 1) and *advanceStep*. Each of these functions returns one or more linked lists representing generalized patterns. We refer to the generalized pattern currently being built as $genXPath$. To generalize a pair of path expressions, we make an initial call $generalizeStep(null, p_i, p_j)$, where p_i and p_j are pointers to the head nodes of the linked lists representing the path expressions (the initial steps of the two XPath expressions). The algorithm generalizes the nodes pointed to by p_i and p_j to $newNode$ and appends this new node to the $genXPath$ path expression built up to this point. To perform this generalization, we check if p_i and p_j have the same name test. If so, the newly generated node retains the same name test as these nodes. If not, we replace the name test with a wildcard label, $*$. The navigation axis of $newNode$ is determined by calling a function $genAxis(p_i.axis, p_j.axis)$, which returns *descendant axis* (`//`) if at least one of the inputs is a descendant axis, and returns *child axis* (`/`) otherwise. We also use a function $isLast(p)$ to test whether p points to the last step of a path expression (the target of the navigation).

The other function, $advanceStep$, plays the role of traversing the expression lists by advancing the pointers p_i and p_j according to the rules summarized in Table II, which are designed to generate candidates that are as general as possible. In the first rule, we terminate the navigation of the two expressions once we finish generalizing their last steps. A last step node can only be generalized with another last step node, so Rules 2 and 3 test for the case that one expression has reached its last step while the other has not and advance the pointer of the latter to reach its last step. Rule 4 handles the case when we are generalizing two middle steps. In this case, we return the results of three generalizations: (1) advance the pointers of both expressions one step and generalize them, (2) and (3) try to find an occurrence of the first node of first (second) expression in the second (first) expression and generalize them together. In cases (2) and (3), no generalization is performed if the search fails. These two cases handle the reoccurrence of nodes in an expression, for example generalizing `/a/b/d` and `/a/d/b/d` will return `/a//d` and `/a//b/d`. Rule 0 in Table II, is a final rewrite step that we do before returning an XPath. This rule replaces every occurrence of one or more contiguous `/*` steps appearing in the middle of an expression with a descendant axis in the following step. For example, we rewrite both `/a/*/b` and `/a/*/*/b` to `/a//b`.

For example, to generalize candidates C1 and C2 from Table I, we initially make a call *generalizeStep*(*null*, `/Security/Symbol`, `/Security/SecInfo/*/Sector`). *generalizeStep* looks at the nodes `/Security` in both paths and recognizes that they have the same name tests, therefore it creates a node with a `/Security` name test and appends it to the *genXPath* being produced. It then calls *advanceStep*(`/Security`, `/Security/Symbol`, `/Security/SecInfo/*/Sector`) to complete processing these expressions. In this call, Rule 4 of *advanceStep* fires, and we have three possible generated XPath expressions. The first is the result of advancing the pointer of each of them to the next step: *generalizeStep*(`/Security`, `/Symbol`, `/SecInfo/*/Sector`). This call will result in another call *advanceStep*(`/Security`, `/Symbol`,

1	***isLast***(p_i) **and** ***isLast***(p_j) return {*genXPath*}
2	***isLast***(p_i) **and !*isLast***(p_j) $p_{jL} \Leftarrow$ last step in p_j expression. *genXPath* $\Leftarrow$ Append /$\star$ onto *genXPath* return *generalizeStep*(*genXPath*, p_i.next, p_{jL})
3	**!*isLast***(p_i) **and** ***isLast***(p_j) $p_{iL} \Leftarrow$ last step in p_i expression. *genXPath* $\Leftarrow$ Append /$\star$ onto *genXPath* return *generalizeStep*(*genXPath*, p_{iL}, p_j.next)
4	**Otherwise** $p_{in} \Leftarrow$ first occurrence of root node of p_j in p_i.next $p_{jn} \Leftarrow$ first occurrence of root node of p_i in p_j.next *genXPath* $\Leftarrow$ Append /$\star$ onto *genXPath* return {*generalizeStep*(*genXPath*, p_i.next, p_j.next), *generalizeStep*(*genXPath*, p_{in}, p_j.next), *generalizeStep*(*genXPath*, p_i.next, p_{jn})}
0	**Rewrite Rule** Replace any middle step node having /$\star$ or //$\star$ with a // axis in the next step.

TABLE II

RULES USED BY *advanceStep*.

/SecInfo/*/Sector) because we are trying to generalize a last step with a middle step. Rule 2 is now fired and the pointer of the second expression is advanced until its last step and a call *generalizeStep*(/Security/*, /Symbol, /Sector) is issued. Finally, *advanceStep*(/Security/*/*, /Symbol, /Sector) is called from line 12 of Algorithm 1, Rule 1 is fired, a rewrite step is performed, and /Security//* is returned. The second and third alternatives generated by Rule 4 are to search for /Symbol in /SecInfo/*/Sector and for /SecInfo in /Symbol, but as both searches fail, no generalized path expression is produced. Based on these results, we can extend the basic candidates in Table I to include candidate C4. Candidate C3 cannot be generalized with either C1 or C2 because it is of a different data type.

VI. SEARCHING FOR THE OPTIMAL CONFIGURATION

After the candidate enumeration and generalization steps, we have in hand an expanded set of candidate indexes. We need to search the space of possible index configurations consisting of indexes from this candidate set to find the index configuration with the maximum benefit, subject to a constraint specified by the user on the disk space available for the configuration.

This combinatorial search problem can be modeled as a 0/1 knapsack problem [9], 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, and a *benefit* computed as described in Section III.

The problem is further complicated by the fact that indexes interact with each other. The benefit of an index for a query can change depending on whether or not other indexes exist. 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 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 the optimizer plans. We have also implemented a *top down* search that chooses indexes that are as general as possible given 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. We describe these search algorithms next and then we describe a technique to reduce the number of calls to the optimizer that we make in these search algorithms.

A. Greedy Search with Heuristics

The greedy approximation of the 0/1 knapsack problem has proven to be effective for relational index advisors [9], but it is not effective for our XML Index Advisor. The benefit of an index is highly dependent on the existence of other indexes in the configuration. Moreover, the greedy search can select general indexes that can be used for path expressions already covered by other indexes in the configuration. Unfortunately, the optimizer can use only one of these indexes in its 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. Filling up the space reclaimed by eliminating these indexes will not necessarily produce the best index configuration. A better approach is to detect redundant indexes as soon as possible and reclaim the space that they use while we are searching for the best configuration.

To address the index interaction problem, we evaluate the benefit of the entire configuration to decide on adding a new candidate to it or not. Evaluating the configuration is optimized using the technique described in Section VI-C.

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

The greedy search algorithm with heuristics maintains a bitmap of XPath patterns in the workload queries that have indexes on them. Then, before adding any general index to our configuration, we use this bitmap to make sure that this index will not be a replication of others already chosen. When a general index, $x_{general}$, is added to the recommended index configuration, it must be "better" than the indexes it generalizes, $x_1, x_2, \ldots, x_n$. We define $IB(X)$, the *improved benefit* of the set of indexes X, as the benefit of the current configuration when X is added to it. A general index is added to the configuration only if the following two heuristic conditions are satisfied:

$$IB(x_{general}) \geq IB(x_1, x_2, \ldots, x_n)$$

$$Size(x_{general}) \leq (1+\beta)\sum_{i=1}^{n} Size(x_i)$$

Most of the time, general indexes are larger than the specific indexes that they generalize because the specific indexes contain more nodes from the data. The second heuristic restricts the expansion in size that we allow when we choose a general index, and the first heuristic ensures that the general index is at least as good as the specific indexes. Hence, we are biased towards choosing the smallest configuration that is the best for the current workload. The value β is a threshold that specifies how much increase in size we are willing to allow. We have found $\beta = 10\%$ to work well in our experiments.

B. Top Down Search

The greedy search with heuristics recommends the best configuration that specifically fits the given workload. Because of that, it can be viewed as *over-training* for 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 will not change at all. For example, 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 that 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 elements from the data that are reachable by different paths 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 construct a *Directed Acyclic Graph* (DAG) of the candidate indexes while generalizing them. Each node in the DAG represents an XML pattern, and has as its parents the possible generalizations of this pattern, based on our candidate generalization algorithm. For example, when generalizing the two candidates `/Security/Symbol` and `/Security/SecInfo/*/Sector` to get `/Security//*`, a node will be created in the DAG for `/Security//*` and this node will be a parent of the two candidates. At the end of this construction phase, we will have a DAG rooted at 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 the maximum benefit that can be achieved. However, we note that 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 new metrics ΔB and ΔC. Assume that candidates $x_1, x_2, \ldots, x_n$ 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 $x_1, x_2, \ldots, x_n$. We define ΔB and ΔC as follows:

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

Since our goal is to obtain the maximum total benefit for the workload with 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 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 is general.

We implemented two versions of the top down algorithm. In the first version, we ignore index interaction when calculating ΔB. The benefit of a configuration is calculated as the sum of the benefits of its indexes. We call this version *top down lite*. In the second version, we evaluate the benefit of every configuration using the technique described next. We refer to this version of the search algorithm as *top down full*.

C. Efficient Benefit Evaluation

To evaluate the benefit of a configuration consisting of multiple indexes, 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. A simplistic approach for taking index interaction into account is to evaluate 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, we have developed a more efficient approach that reduces the number of calls to the optimizer while taking index interaction into account.

While we are generating the set of candidate indexes (basic and generalized), we keep track for each index, x, of which (XQuery or SQL/XML) workload statements produced basic candidate index patterns that are covered by this index. These are the statements that can benefit from x, and we call this set of statements 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 I, 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, which gives us the two sub-configurations {C1} and {C2, C3}. If we need to estimate the benefit of C2 while taking potential interactions with other indexes into account, we only need to evaluate the benefit of {C2, C3}, not {C1}. When doing this, we only need to call the optimizer for the affected sets of C2 and C3, not for the entire workload.

VII. Experiments

A. Experimental Setup

IBM DB2 9 (pureXML™) supports both relational and XML data [3], [13]. We have modified the DB2 9 query optimizer to create a prototype version that supports the two new optimizer modes that our Index Advisor requires. These new modes are implemented as `EXPLAIN` modes in the optimizer. The client side XML Index Advisor is implemented in Java™ 1.5, and communicates with the prototype server via JDBC. We have conducted our experiments on a Dell PowerEdge 2850 server with two Intel® Xeon® 2.8GHz CPUs (with hyperthreading) and 4GB of memory running SUSE Linux® 10. The database is stored on a 146GB 10K RPM SCSI drive.

We use two XML benchmarks for our experiments: TPoX [10] and XMark [28]. Due to lack of space, we only present the results for TPoX in this paper. The results for XMark can be found in [24]. We generate the benchmark data using a scale factor of 1GB. We evaluate our XML Index Advisor on the 11 XQuery queries that are given in the TPoX benchmark specification. To illustrate the effectiveness of our generalization algorithm, we also use synthetic queries in Section VII-C.

DB2 stores XML data in XML-typed columns of tables, and it can create XML indexes on these columns with specific index patterns that are given as XPath path expressions [13]. The indexes can be used to answer structural or value queries on the data. Hence, the goal of the XML Index Advisor is to recommend index patterns for indexes on the XML-typed columns of a table, based on the workload queries.

Our metric for evaluating the recommendations of the XML Index Advisor is *estimated speedup*: The estimated execution time of the workload with no XML indexes divided by the estimated execution time of the workload with the index configuration recommended by the Index Advisor.

B. Effectiveness of Recommendations

We have implemented five different combinatorial search algorithms in our Index Advisor. Four of these algorithms are described in Section VI: (1) greedy search (without heuristics), (2) greedy search with the heuristics, (3) top down lite, and (4) top down full. We have also implemented a dynamic programming search, which finds the optimal solution to the knapsack problem, but is prohibitively expensive and ignores index interaction (i.e., it finds an optimal solution *modulo index interactions*).

Figure 2 shows the estimated speedup for the different search algorithms with varying disk space budgets. The figure also shows the speedup for a configuration in which we have XML indexes for every indexable XPath expression in the query workloads (the *All Index* configuration). This is the best possible configuration for a workload that consists of queries with no updates. The size of this configuration is 95MB. We use the 11 TPoX queries for both recommending indexes and evaluating the recommendations.

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 *All Index* 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, as can be seen from Figure 2. Greedy search with heuristics and top down lite search are both able to achieve better speedups than greedy search, approaching the performance of dynamic programming. These two search strategies achieve similar speedups in this experiment, but as we see in the next section, the recommended configurations can be different. Top down full search has the best performance because it takes into account index interaction. This makes it perform even better than dynamic programming (which does not take index interaction into account) for some cases.

Figure 3 shows that the superior recommendations of the top down full search come at a cost. The figure shows the run time of the Index Advisor for varying disk space budgets. Top down full search takes up to 7 times more than greedy search with heuristics. However, the run time of top down full 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.

C. Recommending General Indexes

In this section, we demonstrate that our Index Advisor can recommend indexes that are more general than the candidates appearing in 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. To address this question, we generated synthetic workloads consisting of

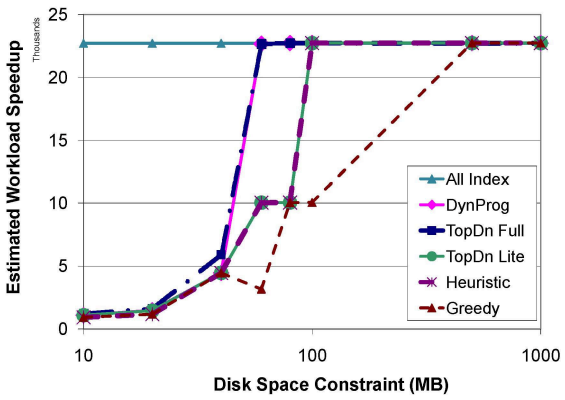

Fig. 2. Estimated speedup.

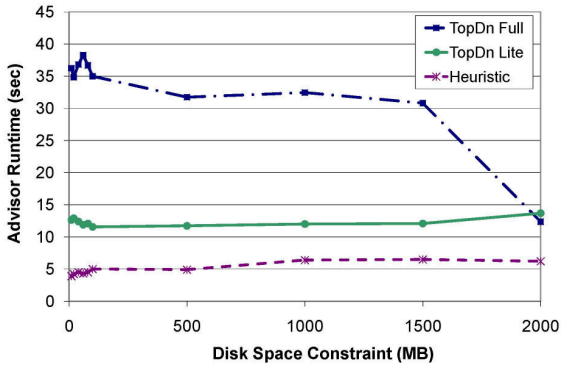

Fig. 3. Advisor run time.

random XPath path expressions that occur in the data. Table III shows the number of basic candidate indexes generated by the query optimizer in Enumerate Indexes mode for these workloads as the number of workload queries increases, and also the total number of candidate indexes after candidate generalization. The numbers show that, even for these random workloads with little or no commonality, we are able to expand the number of candidate indexes by up to 50% by adding general candidate indexes.

The next question we address is how many of the general candidate indexes we generate can be recommended by our top down algorithm, and how useful these recommended indexes are. Recall that the goal of top down search is to recommend a set of indexes that is useful for the workload and as general as possible given the disk space budget. The generality of these indexes is typically not expected to add any benefit to the workload queries, but it will make the configuration more usable if the workload has new unseen queries added to it in the future.

Table IV shows the number of general and specific indexes recommended for different disk space budgets by greedy search with heuristics, top down lite search, and top down full search. Greedy search with heuristics is not designed with the explicit goal of recommending general indexes and so it is very conservative about recommending them. Top down search, on the other hand, recommends more general indexes the more disk space it has.

To show the effect of recommending general indexes on

Queries	Basic Cands.	Total Cands.
10	12	16
20	23	34
30	33	49
40	42	60
50	52	81

TABLE III
NUMBER OF CANDIDATE INDEXES.

Disk Budget	Top Down Lite	Top Down Full	Heuristics
100MB	G: 1, S: 14	G: 1, S: 14	G: 0, S:15
500MB	G: 3, S: 9	G: 2, S: 11	G: 0, S: 15
1000MB	G: 4, S: 7	G: 3, S: 8	G: 0, S: 15
2000MB	G: 8, S: 0	G: 8, S: 0	G: 1, S: 13

TABLE IV
NUMBER OF GENERAL (G) AND SPECIFIC (S) INDEXES RECOMMENDED.

speedup for different 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. 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 20. Figure 4 shows the estimated speedup on the test workload as we vary the training workload size with a disk space budget of 2GB. The figure shows the speedup for top down lite search, greedy search with heuristics, and an *All Index* configuration that is based on the entire test workload. In this case, the speedup of top down full search is similar to that of top down lite, so we eliminate it from the figure for clarity. The figure shows that as the advisor sees more and more of the test workload, it can recommend a configuration approaching the *All Index* configuration using either search strategy. However, it is clear 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.

Figure 5 shows the *actual* speedup corresponding to Figure 4. When computing actual speedup, we had to eliminate from the workload two queries that we timed out after 10 hours when there were no indexes, but that finished in less than 30 seconds using the index configuration recommended for them by our Index Advisor. These queries gain the maximum benefit from our Index Advisor, but they cannot be plotted on the figure since their speedup is infinite! We can see that the actual speedup corroborates the conclusions drawn from our estimated speedup experiments.

In addition to the experiments presented here, we have experimentally demonstrated the accuracy of our cost estimation using virtual indexes, which is needed for accurate benefit estimation. We have also shown that the Index Advisor accurately takes into account the cost of index maintenance when making its recommendations. For this more comprehensive set of experiments, please refer to [24].

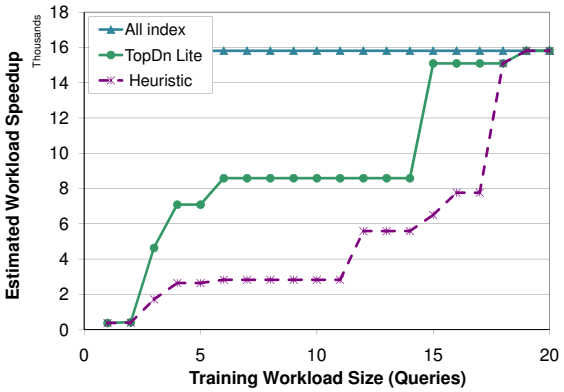

Fig. 4. Generalization to unseen queries.

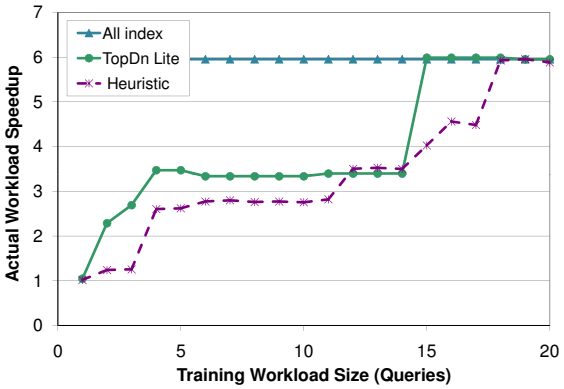

Fig. 5. Generalization to unseen queries - Actual speedup.

VIII. Conclusions

In this paper, we presented an XML Index Advisor that recommends the best set of indexes for a given XML database and query workload. The advisor is tightly coupled with the query optimizer, using it for both enumerating and evaluating indexes. It employs search algorithms that can recommend indexes that are useful beyond the training workload. During its search, the advisor makes a minimal number of optimizer calls, making it very efficient. We have implemented our XML Index Advisor in a prototype version of DB2, and shown that it can recommend indexes that result in significant speedups for workload queries.

References

[1] "eXist: An Open Source Native XML Database," available at: `http://exist.sourceforge.net/`.

[2] T. Fiebig and et. al, "Anatomy of a native XML base management system," *VLDB Journal*, vol. 11, no. 4, 2002.

[3] K. Beyer *et al.*, "System RX: One part relational, one part XML," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2005.

[4] Oracle Corp., *Oracle Database 11g Release 1 XML DB Developer's Guide*, 2007, available at: `http://www.oracle.com/pls/db111/homepage`.

[5] M. Rys, "XML and relational database management systems: Inside Microsoft SQL Server 2005," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2005.

[6] K. Beyer and et. al, "DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL," *IBM Systems Journal*, vol. 45, no. 2, 2006.

[7] A. Balmin, K. S. Beyer, F. Özcan, and M. Nicola, "On the path to efficient XML queries," in *Proc. Int. Conf. on Very Large Data Bases*, 2006.

[8] S. Chaudhuri and V. R. Narasayya, "An efficient cost-driven index selection tool for Microsoft SQL Server," in *Proc. Int. Conf. on Very Large Data Bases*, 1997.

[9] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley, "DB2 advisor: An optimizer smart enough to recommend its own indexes," in *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2000.

[10] M. Nicola, I. Kogan, and B. Schiefer, "An XML transaction processing benchmark," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2007, benchmark Available at: `https://sourceforge.net/projects/tpox/`.

[11] C.-W. Chung, J.-K. Min, and K. Shim, "APEX: An adaptive path index for XML data," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.

[12] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering indexes for branching path queries," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.

[13] M. Nicola and B. Van der Linden, "Native XML support in DB2 Universal Database," in *Proc. Int. Conf. on Very Large Data Bases*, 2005.

[14] C. Qun, A. Lim, and K. W. Ong, "D(k)-index: An adaptive structural summary for graph-structured data," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[15] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala, "Database tuning advisor for Microsoft SQL Server 2005," in *Proc. Int. Conf. on Very Large Data Bases*, 2004.

[16] D. C. Zilio *et al.*, "DB2 design advisor: Integrated automatic physical database design," in *Proc. Int. Conf. on Very Large Data Bases*, 2004.

[17] Z. Guo, Z. Xu, S. Zhou, A. Zhou, and M. Li, "Index selection for efficient XML path expression processing," in *Conceptual Modeling for Novel Application Domains, ER 2003 Workshop Proceedings*, 2003.

[18] S. Chaudhuri, Z. Chen, K. Shim, and Y. Wu, "Storing XML (with XSD) in SQL databases: Interplay of logical and physical designs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 12, 2005.

[19] B. C. Hammerschmidt, M. Kempa, and V. Linnemann, "Autonomous index optimization in XML databases," in *Proc. Int. Workshop on Self-Managing Database Systems (SMDB)*, 2005.

[20] K. Runapongsa, J. M. Patel, R. Bordawekar, and S. Padmanabhan, "XIST: An XML index selection tool," in *Proc. Int. Symp. on Database and XML Technologies (XSym)*, 2004.

[21] B. C. Hammerschmidt, M. Kempa, and V. Linnemann, "A selective key-oriented XML index for the index selection problem in XDBMS," in *Proc. Int. Conf. on Database and Expert Systems Applications*, 2004.

[22] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity for indexing paths in graph-structured data," in *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2002.

[23] A. Balmin *et al.*, "Cost-based optimization in DB2 XML." *IBM Systems Journal*, vol. 45, no. 2, 2006.

[24] I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. Beyer, and C. Zuzarte, "XML index recommendation with tight optimizer coupling," University of Waterloo, Tech. Rep. CS-2007-22, 2007, available at: `http://www.cs.uwaterloo.ca/research/tr/2007/CS-2007-22.pdf`.

[25] A. Balmin, F. Özcan, K. Beyer, R. J. Cochrane, and H. Pirahesh, "A framework for using materialized XPath views in XML query processing," in *Proc. Int. Conf. on Very Large Data Bases*, 2004.

[26] B. Mandhani and D. Suciu, "Query caching and view selection for XML databases," in *Proc. Int. Conf. on Very Large Data Bases*, 2005.

[27] W. Xu and Z. M. Özsoyoglu, "Rewriting XPath queries using materialized views," in *Proc. Int. Conf. on Very Large Data Bases*, 2005.

[28] A. R. Schmidt *et al.*, "The XML benchmark project," CWI, Tech. Rep. INS-R0103, 2001.

Trademarks