XML Index Recommendation with Tight Optimizer Coupling

Technical Report CS-2007-22
July 11, 2007

Iman Elghandour
University of Waterloo

Ashraf Aboulnaga
University of Waterloo

Daniel C. Zilio
IBM Toronto Lab

Fei Chiang *
University of Toronto

Andrey Balmin
IBM Almaden Research Center

Kevin Beyer
IBM Almaden Research Center

Calisto Zuzarte
IBM Toronto Lab

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 candidate indexes 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 recommended 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.

1 Introduction

There are currently several native XML database systems [14, 17], and XML support has also been added to most commercial relational database systems [6, 31, 35]. 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 [5]. 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. This useful feature is supported in database systems such as DB2 9 [5] and the upcoming Oracle 11g [32]. However, users now face the problem of choosing the set of XML patterns to include in an 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, also 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 [9, 38]. 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.

An Index Advisor needs to address three questions: (1) how to determine the candidate indexes that would be useful for a query or a workload consisting of a set of queries, (2) how to estimate the benefit for a given query of a particular *index configuration* (i.e., a set of indexes), and (3)

*This work was done while the author was at the IBM Toronto Lab

how to search the space of possible index configurations for the optimal configuration that provides the maximum benefit to the workload, taking into account the increased cost of update statements due to indexes, and satisfying disk, schema, and other system constraints. In this paper, we present novel techniques to address each of these questions, and we integrate these techniques into an XML Index Advisor. We have implemented our XML Index Advisor in a prototype version of IBM® DB2® 9 for Linux, Unix, and Windows (henceforth referred to simply as DB2), which supports both relational and XML databases, and we have used this implementation to verify the efficiency of our Index Advisor and the high quality of the index configurations that it recommends.

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 data 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 the optimal one, 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 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. Tight coupling between index recommendation and query optimization has been a feature of relational index recommenders for a long time [18], and one of our contributions in this paper is to extend this coupling to XML.

[1]Throughout this paper, we use examples from the TPoX benchmark [29].

The rest of the paper is organized as follows. We present related work in Section 2. Section 3 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 4).

- 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 5).

- A technique for estimating the benefit of candidate XML indexes or index configurations, relying on the query optimizer. Our technique takes index interaction into account, and attempts to reduce the number of calls to the query optimizer (Section 6).

- 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 7).

- An implementation of the XML Index Advisor in a prototype version of DB2 and an experimental study using the TPoX [29] and XMARK [36] benchmarks (Section 8).

2 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 [10, 25, 30, 31, 33]. In the past few years, there has been a considerable amount of work on index advisors for relational data [1, 9, 38, 40]. Unfortunately, none of these works extend directly to XML databases.

A few attempts were made to recommend indexes for XML data that are shredded and stored in relational databases [8, 19]. In [19], 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 from the database system which can lead to inaccurate estimates. In [8], a new approach to take the interplay of logical and physical design into consideration when shredding XML data into relational databases is proposed. 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 [22, 34]. 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 benefits of candidate index configurations are estimated with any accuracy. In addition, neither of them tackles the issue of generalizing the initial set of candidates, which is equivalent to merging physical design structures in relational databases [1]. We address these shortcomings and we also propose a configuration enumeration algorithm that takes into account the interaction between indexes and yet is efficient in the number of optimizer calls it makes.

In [34], 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. The candidate indexes used in [34] are all paths that occur in the data, with some grouping of structurally equivalent candidates 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 candidates. The candidate generation process does not attempt to generate candidates that are useful for multiple queries. In our work, we rely on the query optimizer to enumerate only the relevant candidate indexes, and we generalize the candidates to generate additional candidates that are useful to similar queries that may appear in future workloads. This results in a much smaller search space of possible configurations, with much more relevant indexes.

Another index recommender for XML is presented in [21, 22]. This index recommender analyzes the workload periodically and creates or drops XML indexes on the fly. As in [34], 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 either a greedy search, which can be inaccurate, or an exhaustive search, which is slow. The configuration enumeration step in [21, 22] also ignores the penalty for updates, inserts, and deletes.

3 Overview and Architecture

3.1 XML Databases and XML Indexes

With the increasing need of storing data in XML format, different approaches have been proposed for XML data storage. One approach is to store the XML data natively in systems that support only XML [14, 15, 20, 24]. Another approach attempts to benefit from the mature technology in relational database systems by shredding XML data and storing it in relational tables [7, 11, 12, 37]. Another technique is to store XML as BLOB columns [13]. On the other side of the spectrum, XML can be used to publish relational data [16]. Recently, an XML column data type has been added to several commercial database systems so XML data can now be stored natively in relational database systems. Such database systems are referred to as hybrid relational/XML database systems [6, 28, 35].

XML query languages use XPath path expressions to retrieve elements in the data. This retrieval can be helped by the presence of an XML index, and there have been many proposals for different types of XML indexes over the past few years. XML indexes can be categorized into *structural indexes* that speed up navigation through the hierarchical structure of the XML data (e.g., [26]), and *value indexes* that help in retrieving XML elements based on some condition on the values they contain (e.g., [30, 31]). 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 yield greater than 4.5).

Several of these XML indexes have the ability to *partially index* the XML data to improve the speed of index maintenance and lookups. In this case, the index includes only the XML elements that are reachable via specific *index patterns* [5, 25]. These index patterns are typically specified as XPath expressions. For example, we can have an index that includes only XML elements that are reachable by the pattern `/Security/*`. This index 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`.

An example of a system that allows partial indexing of XML data is DB2. In DB2 [5, 6], XML data are stored natively in columns with XML data type. In the create table statement, one or more columns can be defined to be of type XML. For every row in the data, a well formed XML doc-

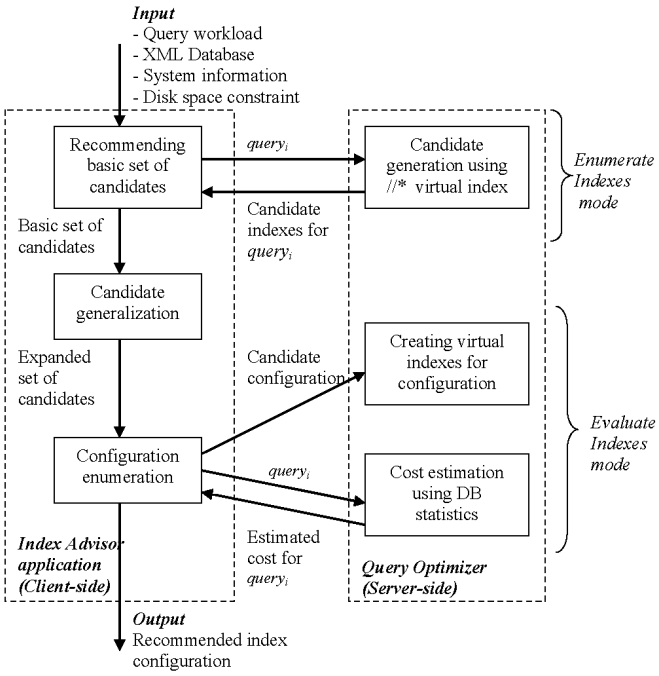

Figure 1. XML Index Advisor architecture.

ument is stored for each XML column. XML indexes are created for one XML column and would only include elements from all documents of that column that are reachable by a pattern that is given in the create index statement. For example, the data definition language statement for creating an index on an XML column `SDOC` (of type XML) of table `Security`, with index pattern `/Security/*`, is as follows:

Example: DDL for creating XML index

```
CREATE INDEX securityVals ON Security(SDOC)
GENERATE KEY USING XMLPATTERN /Security/*
AS SQL DOUBLE
```

3.2 XML Index Advisor Architecture

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 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 [9, 38]. 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.

In the rest of the paper, we use as a running example, a workload consisting of the following two queries from the TPoX benchmark [29].

Q1: Return a security having a 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>
```

4 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 [2]. 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) [4, 27, 39].

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 allows our Index Advisor to recommend usable indexes by this system.

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 the document and hence can be matched with any XPath pattern 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. Essentially, we have enabled the optimizer to answer the question: "If all possible indexes were available, which ones would be considered for this query?"

To conform with the XPath standard, DB2 uses different indexes for different data types, and different indexes for elements and attributes [5, 6]. Thus, in our implementation of the XML Index Advisor in DB2, we create several `//*` indexes. For each data type, we create an index with the pattern `//*` (for elements) and an index with the pattern `//@*` (for attributes). All of these indexes are used by the optimizer in Enumerate Indexes mode to recommend candidate index patterns.

The candidate index patterns enumerated by the optimizer will already 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 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.

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

Table 1. Basic set of candidates.

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.

5 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.

In Section 5.1, we focus on index patterns that are expressed as *linear XPath path expressions* that do not contain predicates. For example, we would handle an XML index with index pattern `/Security/Yield`, which can be used to answer a query like `/Security[Yield >= 4.5]`. Database systems such as DB2 and Oracle currently support only these linear XPath path expressions for their index patterns [30, 32] and so their query optimizer would only enumerate such index patterns in the Enumerate Indexes

mode. 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, the XPath expressions in the *query workload* can contain predicates at arbitrary locations. Next, in Section 5.2, we extend our approach to handle branching XPath index patterns that include predicates (analogous to multi-column relational indexes). In Section 5.3, we present an approach for generalizing index patterns in the basic candidate set individually, even if they cannot be generalized with other path expressions in the basic candidate set.

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. The steps we follow to generalize the XPath expressions in a candidate set are presented in Algorithm 1. 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 generalizeXPCandidates($XPset$)

```
1:  start ← 0
2:  end ← XPset.size
3:  repeat
4:     genXPathNo ← 0
5:     for i = start to end − 1 do
6:        for j = i + 1 to end do
7:           if p_i and p_j have same data type, namespace,
             and defined on same table and column then
8:              genXPath ← generalizeStep(null, p_i, p_j)
9:              if genXPath ∉ XPset then
10:                add genXPath to XPset
11:                genXPathNo ← genXPathNo + 1
12:             end if
13:          end if
14:       end for
15:    end for
16:    start ← end + 1
17:    end ← end+ genXPathNo.size
18: until genXPathNo = 0
```

5.1 Generalizing Pairs of Linear Candidates

For the case when all candidate indexes are linear XPath path expressions, 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: $generalizeStep$ 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 $generalizeStep(null, p_i, p_j)$ (Algorithm 2), 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 XPaths). 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).

Algorithm 2 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)}
```

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 2, 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

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 `/*` 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 `/*` 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 `/*` 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 `/*` or `//*` with a `//` axis in the next step.

Table 2. Rules used by *advanceStep*.

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 the 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 2, is a final rewrite step that we do before returning an XPath. Rule 0 replaces every occurrence of one or more contiguous `/*` steps appearing in the middle of an expression with a descendant axis in the step following it. For example, we rewrite both `/a/*/b` and `/a/*/*/b` to `/a//b`.

For example, to generalize candidates C1 and C2 from Table 1, 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`, `/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 2, 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 expressions is produced. Based on these results, we can extend the basic candidates in Table 1 to include candidate C4 in Table 3. Candidate C3 cannot be generalized with either C1 or C2 because it is of a different data type.

C4	`/Security//*`	string
C5	`/Security/*`	numerical

Table 3. Generalized candidates.

5.2 Generalizing Pairs of Multi-value (Branching) Candidates

5.2.1 Representing Multi-value XML Index Patterns

XPath patterns can be used to specify more than one value to be indexed. For example, we can recommend an index `/Security[Yield]/SecInfo/*[Sector]` for query Q2. This pattern indicates indexing the values of elements reachable by the XPath `/Security/Yield` as well as those reachable by the XPath `/Security/SecInfo/*/Sector`. This more general class of index patterns is analogous to multi-column indexes in relational databases. Due to the rich structure of this new class of indexes, we need to change the internal representation of an index pattern from a list to a tree to accommodate the multiple values. This class of index patterns contains navigational steps and specifies the indexed elements in predicates. When indexing more than one element with the same ancestors, we use conjuncts to represent this. For example, to build an index on the patterns `/Security/Yield` and `/Security/PE` together, we represent this by the pattern `/Security[Yield and PE]`. For uniformity, we also change the representation of linear patterns, hence to index elements reachable by the XPath `/Security/Yield` we use the pattern `/Security[Yield]` to represent them. We make the assumption that there is at least one indexable element (one predicate) in a given XML pattern.

The query optimizer of a database system that supports multi-valued XML indexes would generate such indexes in

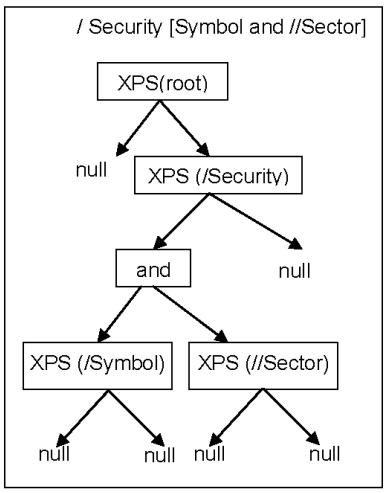

Figure 2. XPath XPS tree example.

its *Enumerate Indexes* mode. Our focus in this section is on generalizing such indexes to create more candidates.

Our revised version of the generalization algorithms works on XPath path expressions represented as expression trees with the same structure as the XPS trees defined in [4]. An XPS tree (*XP*ath *S*tep tree) is composed of labeled nodes. Each node is labeled with its navigation axis and its node test, where the navigation axis is the special axis `root` or one of: `child`, `descendant`, or `attribute`. The test can be either a name test or a wildcard test. Each node can have two children, the left child represents any predicate on the node, while the right child represents the next step in the expression. To navigate the tree, we advance the navigational pointer to the right children of nodes. Also, to check a predicate of a node, we check its left child. In the current implementation, we do not allow predicates to be nested. Figure 2 shows an example of an XPS tree representing a path expression. In the example, we are indexing the values of `/Security/Symbol` and `/Security//Sector` together in the same index. We consider XPath index patterns that can contain both navigational steps and predicate branching. The navigation can contain label wildcards, `"*"`, child axis navigation, `"/"`, and descendant navigation, `"//"`. The predicates can involve conjunctions to indicate the indexing of more than one element with the same ancestors. We obtain this tree representation of the path expressions by parsing the path expressions in the basic candidate set. We perform a postprocessing step after generalization to obtain the path expression of the generalized tree in a linear format. We describe next the versions of Algorithm 2 and the rules in Table 2.

5.2.2 Generalizing Multi-value Patterns

To generalize a pair of expressions we rely on the function $generalizeTreeStep$, which generalizes the roots of the pair of expression trees passed to it, and then navigates to the next nodes in the trees and recursively calls itself to generalize them. $generalizeTreeStep$ takes two expression trees p_i and p_j and a list of general expression trees built to this point, and returns a list of all general expression trees constructed after adding the generalization of the current steps of p_i and p_j. The initial call to generalize a pair of expressions is $generalizeTreeStep(null, p_i, p_j)$. Recursive calls are made in $generalizeTreeStep$ to generalize all the steps of the expression trees. The rules for generalizing a pair of nodes and advancing the path expression pointers are described in Table 4 and continued in Table 5. In Rule 1, we terminate the navigation of the two expressions when at least of them reaches its end. Rule 2 handles the case of two steps with no predicates. In this case we try to find an occurrence of the root node of the first (second) expression in the second (first) expression and generalize them together. If both searches fail, we only generalize the current steps together. In these two cases, a copy of the new general node is appended to all the existing general tree expressions that are under construction, and the new trees are returned to the calling function. In Rules 3 and 4, we generalize two steps where only one of them has a predicate. In these two cases, we try to advance the pointer of the step with no predicate to a step with a predicate similar to the one in the other expression. If this fails, we try to find a matching step with a similar name to generalize with. If the previous two attempts fail, the current two steps are generalized together while ignoring the predicate. Rule 5 deals with the case of two steps having predicates. In this case an attempt to generalize each one of them with a similar predicate is made. If these attempts fail, the two nodes are generalized together and a new predicate with a conjunction of the original two predicates is created.

In order to accomplish the task of $generalizeTreeStep$, some helper functions are used. $hasPred$ is a boolean function that returns true if the current step of the expression has a predicate child. For example, $hasPred($`/Security[Yield]`$)$ returns true. $appendStep$ navigates to the last step of an expression and appends a next step child to it. For example, $appendStep($`/Security`$,$ `/SecInfo`$)$ returns `/Security/SecInfo`. Similar to $appendStep$ is $appendPred$, which navigates to the last step of an expression and then appends a predicate child to it. For example, $appendPred($`/Security`$,$ `/Yield`$)$ returns `/Security[Yield]`. $generalizeNode$, which is described in Algorithm 3, takes two nodes and generalizes them to $newNode$, and returns $newNode$. To perform this generalization, we check if the two nodes have the

1	p_i=**null or** p_j=**null** return {*genXPathTree*}

2	**not(*hasPred*(p_i)) and not(*hasPred*(p_j))**
	$p_{in} \Leftarrow$ first occurrence of root node of p_j in p_i
	$p_{jn} \Leftarrow$ first occurrence of root node of p_i in p_j
	if (p_{in} != *null*) **then**
	genTree1 $\Leftarrow$
	generalizeStepNoPred(*genXPathTree*, p_{in}, p_j, "*")
	$p_i \Leftarrow p_{in}$
	end if
	if (p_{jn} != *null*) **then**
	genTree2 $\Leftarrow$
	generalizeStepNoPred(*genXPathTree*, p_i, p_{jn}, "*")
	$p_j \Leftarrow p_{jn}$
	end if
	if (p_{in} = *null* **and** p_{jn} = *null*) **then**
	genTree3 $\Leftarrow$
	generalizeStepNoPred(*genXPathTree*, p_i, p_j, *null*)
	end if
	genTrees $\Leftarrow$ *genTree1* $\cup$ *genTree2* $\cup$ *genTree3*
	return *generalizeTreeStep*(*genTrees*, p_i.*next*, p_j.*next*)

3	***hasPred***(p_i) **and not(*hasPred*(p_j))**
	$p_{jn} \Leftarrow$ first occurrence of root node of p_i in p_j
	$p_{jpn} \Leftarrow$ first occurrence in p_j where p_{jpn}.*pred* = p_i.*pred*
	if (p_{jpn} != *null*) **then**
	genTree1 $\Leftarrow$
	generalizeStepPred(*genXPathTree*, p_i, p_{jpn}, "*")
	$p_j \Leftarrow p_{jpn}$
	end if
	if (p_{jn} != *null* **and** p_{jpn} = *null*) **then**
	genTree2 $\Leftarrow$
	generalizeStepNoPred(*genXPathTree*, p_i, p_{jn}, "*")
	$p_j \Leftarrow p_{jn}$
	end if
	if (p_{jn} = *null* **and** p_{jpn} = *null*) **then**
	genTree3 $\Leftarrow$
	generalizeStepNoPred(*genXPathTree*, p_i, p_j, *null*)
	end if
	genTrees $\Leftarrow$ *genTree1* $\cup$ *genTree2* $\cup$ *genTree3*
	return *generalizeTreeStep*(*genTrees*, p_i.*next*, p_j.*next*)

4	**not(*hasPred*(p_i)) and *hasPred*(p_j)** similar to Rule 3 after switching p_i with p_j

Table 4. Rules used by *generalizeTreeStep*.

5	***hasPred***(p_i) **and *hasPred*(p_j)**
	$p_{jpn} \Leftarrow$ first occurrence in p_j where p_{jpn}.*pred* = p_i.*pred*
	$p_{ipn} \Leftarrow$ first occurrence in p_i where p_{ipn}.*pred* = p_j.*pred*
	if (p_{jpn} != *null*) **then**
	genTree1 $\Leftarrow$
	generalizeStepPred(*genXPathTree*, p_i, p_{jpn}, "*")
	$p_j \Leftarrow p_{jpn}$
	end if
	if (p_{ipn} != *null*) **then**
	genTree2 $\Leftarrow$
	generalizeStepPred(*genXPathTree*, p_{ipn}, p_j, "*")
	$p_i \Leftarrow p_{ipn}$
	end if
	if(p_{ipn} = *null* **and** p_{jpn} = *null*) **then**
	genTree3 $\Leftarrow$
	generalizeStepPred(*genXPathTree*, p_i, p_j, *null*)
	end if
	genTrees $\Leftarrow$ *genTree1* $\cup$ *genTree2* $\cup$ *genTree3*
	return *generalizeTreeStep*(*genTrees*, p_i.*next*, p_j.*next*)

Table 5. Rules used by *generalizeTreeStep* (continued).

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 use $generalizeStepNoPred$, outlined in Algorithm 4, to generalize two nodes and add their generalization to all the general expressions being constructed as a next step. A similar function is $generalizeStepPred$ (Algorithm 5), which generalizes two nodes *and their predicates* and then appends the new generalized step and predicate to all the general expressions being constructed .

Algorithm 3 generalizeNode(p_i, p_j)

```
1: create newNode
2: if p_i.nameTest = p_j.nameTest then
3:     newNode.nameTest = p_i.nameTest
4: else
5:     newNode.nameTest = "*"
6: end if
7: newNode.axis = genAxis(p_i.axis , p_j.axis )
8: append newNode to genXPath
9: return newNode
```

We illustrate candidate generalization for multi-valued indexes using the following two queries on the TPoX data, Q3 and Q4:

Q3: List securities with a particular industry type given a yield range

```
for $sec in SECURITY('SDOC')/Security
```

```
where $sec/Yield < 3
  and $sec/SecInfo/*/Industry= "Personal"
return $sec
```

Q4: List security names 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>
```

For these two queries, the enumerated indexes are: `/Security[Yield]/SecInfo/*[Industry]` and `/Security[Yield]/SecInfo/*[Sector]`. Initially we make a call *generalizeTreeStep*(*null*, `/Security[Yield]/SecInfo/*[Industry]`, `/Security[Yield]/SecInfo/*[Sector]`). Since the roots of both expressions have predicates, Rule 5 is fired. We attempt to find `Security` in the first expression, and also `Security` in the second expression. Since the result is the same as the root node, we continue with only one of the result paths. *generalizeStepPred*(*null*, `/Security[Yield]`, `/Security[Yield]`) is called and a new expression `/Security[Yield]` is returned. Next, the pointers of the two expressions are advanced and *generalizeTreeStep*(`/Security[Yield]`, `/SecInfo/*[Industry]`, `/SecInfo/*[Sector]`) is called again to continue processing the expressions. The current steps have no predicates, so Rule 2 is fired. The two nodes are the same so only one path is used and a new node is appended to the current general expression. Then *generalizeTreeStep*(`/Security[Yield]/SecInfo`, `/*[Industry]`, `/*[Sector]`) is called. Rule 5 is fired as the two expressions have predicates. But the two predicates `Industry` and `Sector` are not equal. Hence, a new predicate node with a conjunction of `Industry` and `Sector` is created. Finally a new expression of `/Security[Yield]/SecInfo/*[Industry and Sector]` is returned and the generalization is terminated by Rule 1 when we encounter null steps.

5.3 Generalizing Individual Candidates

Some path expressions in the basic candidate set might not be generalized with any other path expression. An example of this is candidate C3 in Table 1. To get more general candidates even from these individual candidate paths that have no common sub-expressions with other candidates, we use a heuristic technique that predicts the existence of other expressions similar to a candidate. The heuristic replaces the last non-`*` navigation step in the candidate path with a `*` navigation step. For example, we can generalize path C3 to `/Security/*`, C5 in Table 3. This

Algorithm 4 generalizeStepNoPred($genXPathTree$, p_i, p_j, $preNode$)

1: $genXPathTreeNew \Leftarrow \{\}$
2: *newNode* $\Leftarrow$ *generalizeNode*(p_i , p_j)
3: **for all** t such that $t \in genXPathTree$ **do**
4: **if** $preNode \neq null$ **then**
5: $t_{new} \Leftarrow$ *appendStep*(t, $preNode$)
6: $t_{new} \Leftarrow$ *appendStep*(t_{new}, $newNode$)
7: **else**
8: $t_{new} \Leftarrow$ *appendStep*(t, $newNode$)
9: **end if**
10: $genXPathTreeNew \Leftarrow genXPathTreeNew \cup t_{new}$
11: **end for**
12: **return** $genXPathTreeNew$

Algorithm 5 generalizeStepPred($genXPathTree$, p_i, p_j, $preNode$)

1: $genXPathTreeNew \Leftarrow \{\}$
2: $newNode \Leftarrow$ *generalizeNode*(p_i , p_j)
3: $newPred \Leftarrow$ *generalizeNode*($p_i.pred$, $p_j.pred$)
4: **for all** t such that $t \in genXPathTree$ **do**
5: **if** $preNode \neq null$ **then**
6: $t_{new} \Leftarrow$ *appendStep*(t, $preNode$)
7: $t_{new} \Leftarrow$ *appendStep*(t_{new}, $newNode$)
8: **else**
9: $t_{new} \Leftarrow$ *appendStep*(t, $newNode$)
10: **end if**
11: **if** $newPred \neq null$ **then**
12: $t_{new} \Leftarrow$ *appendPred*(t_{new}, $newPred$)
13: **else**
14: $t_{new} \Leftarrow$ *appendConjunctPred*($p_i.pred$, $p_j.pred$, t_{new})
15: **end if**
16: $genXPathTreeNew \Leftarrow genXPathTreeNew \cup t_{new}$
17: **end for**
18: **return** $genXPathTreeNew$

approach could be extended to consult the data to determine the usefulness of such a generalization and recommend other generalizations. In the latter case, a * replacement would only be performed when there are other paths in the data with the same common leading path.

6 Estimating the Benefit of XML Indexes

Relational index advisors leverage the query optimizer to estimate the benefit to a query workload of having a particular index configuration [9, 38]. To do the same in our XML Index Advisor, we added a new query optimizer mode that we call the Evaluate Indexes mode. This mode relies on creating virtual indexes and estimating the cost of workload queries with these virtual indexes in place. The optimizer can 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 Indexes mode, the optimizer returns the set of indexes that were used, plus their 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.

The XML Index Advisor architecture allows us to rely completely on the query optimizer for cost estimation by using its Evaluate Indexes mode, 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 [3]. Next, we describe the general approach used in the Evaluate Indexes mode, and the details of the Evaluate Index mode that we have implemented in the DB2 query optimizer, and then we describe how we use the information returned by the optimizer in our Index Advisor application.

6.1 Evaluate Mode Implementation

While finding the query execution plan in the presence of one or more virtual indexes, the optimizer needs statistics about these virtual indexes to get better cost estimates. Some of these statistics are *data statistics*, such as the distinct XPaths in the data that are being indexed and their frequencies, while others are *index statistics* such as the number of disk pages occupied by the index. Our approach is to collect all the necessary data statistics if needed using the query optimizer's normal (i.e., non-virtual) statistics collection module (RUNSTATS in DB2). We then use these data statistics to estimate the index statistics for the virtual indexes. We use the same approach for estimating an XML index cardinality that is described in [3].

DB2 implements XML indexing using a B-tree index, and the query optimizer requires two statistics for an XML index: its cardinality and its size on disk [3]. The cardinality, or total number of entries, of an index is the total number of XML nodes in all the XML documents that are stored in the column of the table that the index is defined on that match the index pattern. As described in [3], to estimate the XML column cardinality, a count of all linear rooted pathes occurring in the documents of that column is collected. But because the number of occurring rooted paths can be huge, this count is only kept for the most frequently occurring paths. To estimate the cardinality of an XML pattern, we check all the most frequent occurring paths, stored in the catalog, for the ones that can be matched with this pattern, and calculate their average. The calculated average is used as an estimate of the number of nodes that are reachable by this XML pattern.

To estimate the size of an index, we again use the data statistics to estimate the different components needed: the size of the index key and the number of keys. While the number of keys for an index entry is fixed and based on the index implementation, the size of the index key is calculated as the average size of the index keys for the most frequently occurring paths. Multiplying the size of the index key by the number of keys gives us an estimate of the total size of the index. With the cardinality and index size statistics of a virtual index in place, this index can be used for cost estimation like any real index.

6.2 Estimating the Benefit of an Index Configuration

In the XML Index Advisor application, we make use of the information returned by the optimizer after evaluating a query in the Evaluate Indexes mode with a specific virtual index configuration in place. The benefit of using an index is estimated as the reduction in query execution cost when the index is created. The benefit of index x to query q is calculated as the difference between the initial cost of query $C_{old(q)}$ and its cost after creating the index $C_{new}(q)$. Thus, benefit of index x to query q is $Benefit(x;q) = C_{old}(q) - C_{new}(q)$. We use the Evaluate 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, we generalize the above calculation to: $Benefit(x;W) = \sum_{q \in W}(C(q)_{old} - C(q)_{new})$. Furthermore, to calculate the benefit of a configuration consisting of multiple indexes, we create all the indexes in the configuration as virtual indexes and then optimize all queries in the workload in Evaluate Indexes mode to estimate their new costs. Thus, we have

$$Benefit(x_1, x_2, \ldots, x_n; W) = \sum_{q \in W}(C_{old}(q) - C_{new}(q)).$$

6.3 Update, Delete, and Insert Costing

Our workloads may contain update, delete, and insert (UDI) statements in addition to queries. Any index 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. The benefit of having an index for update or delete statements is estimated just like the benefit of indexes for queries. 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. Hence, we have special techniques in our application to estimate the maintenance cost of indexes under UDI statements.

To estimate the maintenance cost for an index x_i because of a UDI statement, we use the data statistics to estimate the number of XML documents that have changes because of this update statement *s*, *docChanged(s)*, and the total number of elements included in this index *numElement(x_i)*. Given the total number of XML documents in the database *numDocs*, we can estimate the number of elements that the statement will affect in the index as $nodesUpdated(x_i, s) = (numElement(x_i)/numDocs) \times docChanged(s)$. We are assuming that the number of indexed XML elements from different documents is the same. We are also assuming that all the index entries corresponding to these XML elements will need to be updated, and we use the *docChanged(s)* value to estimate the maintenance cost for this index because of statement s. Based on the system, two calibration constants are used (1) *CPUCostPerNode*: number of CPU operations performed per an index node and (2) *IOCostPerNode*: number of I/O operations performed per an index node. Thus the total maintenance cost of an index x_i because of a statement s is calculated as:

$$mc(x_i, s) = nodesUpdated(x_i, s) \times CPUCostPerNode + nodesUpdated(x_i, s) \times numBTreeLevels \times IOCostPerNode$$

Putting it together, to account for the index maintenance cost, we subtract from the calculated benefit the maintenance cost (mc) of all indexes in the configuration. Thus, for indexes $x_1, x_2, \ldots, x_n$ and workload W:

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

6.4 Efficient Index Configuration 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 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.

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 indexes 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, which gives us the two sub-configurations {C1} and {C2, C3}. To evaluate the {C1} sub-configuration, we only need to optimize Q1 while C1 is created as virtual index. Similarly, to evaluate the {C2, C3} sub-configuration, 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}.

7 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, sub-

ject 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 [38], 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 6.

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 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. We describe these search algorithms next and then we describe a technique to reduce the number of calls to the optimizer.

Figure 3 illustrates the relation between the queries in the workload, extracted XML pattern, and the generalized candidates. 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 4. 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. We build the next levels in the graph until we reach the most general candidates as shown in the figure. For each new candidate, we associate with it a list of XML patterns that were the cause of generating it as well as their descendant patterns. Hence, for any candidate index pattern in the graph, it will have associated to it a list of all candidates in the subtree rooted at this pattern, which we call the *coverage list*. Along with the coverage list, we keep a list of affected queries (described in Section 6.4). The affected list of a generalized pattern is the concatenation of the affected lists of its children.

7.1 Greedy Search with Heuristics

The greedy approximation of the NP-complete 0/1 knapsack problem works 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, starting with the highest ratio, and we continue until the available disk space budget is exhausted. As this is an approximate solution, we try to improve it by skipping candidates that do not fit into the available disk space budget and continuing to add other candidates that can fit into the budget, trying to accommodate as many indexes as possible.

The greedy approximation has proven to be effective for relational index advisors [38], but it was 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. The problem with this solution is that we free up extra disk space that we never use again for adding more indexes, even though this space could be very useful.

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. The configuration evaluation is optimized using the technique described in Section 6.4.

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 search, and heuristics are added to attempt to enforce the new objective in a best effort manner.

This new search algorithm 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, $x_1, x_2, \ldots, x_n$. Algorithm 6 outlines the procedure we follow to search all candidates. 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 (lines 14 and 19 in Algorithm 6):

$$\begin{aligned} IB(x_{general}) &\geq IB(x_1, x_2, \ldots, x_n) \\ Size(x_{general}) &\leq (1+\beta)\sum_{i=1}^{n} Size(x_i) \end{aligned}$$

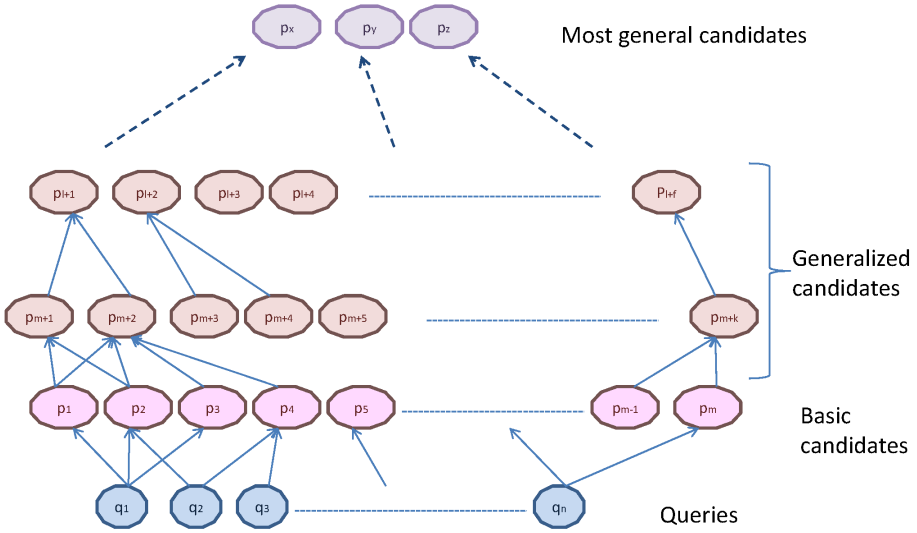

Figure 3. Relation between workload queries and candidate XML patterns

Algorithm 6 heuristicSearch(*candidates*)

```
1:  sort candidates according to their benefit/size ratio
2:  recommended ← φ
3:  currSize ← 0
4:  currCoverage ← φ
5:  while currSize < diskConstraint do
6:     best ← pick the next best cand in candidates
7:     if currCoverage = φ then
8:        recommended ← recommended ∪ cand
9:        currCoverage ← currCoverage ∪ cand.coverage
10:    else if currCoverage ∩ best.coverage = φ then
11:       add best to recommended
12:    else if currCoverage = best.coverage then
13:       evaluate recommended and best
14:       select the bestConf according to benefit and size
15:       recommended ← bestConf
16:    else if currCoverage ∩ best.coverage ≠ φ then
17:       construct tentative with recommended after removing indexes with a coverage subsumed by best
18:       evaluate recommended and tentative
19:       select the bestConf according to benefit and size
20:       recommended ← bestConf
21:    end if
22: end while
23: return recommended
```

Most of the time, general indexes are larger than specific indexes because they 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.

7.2 Top Down Search

The greedy search with heuristics recommends the best configuration that fits the specific 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 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.

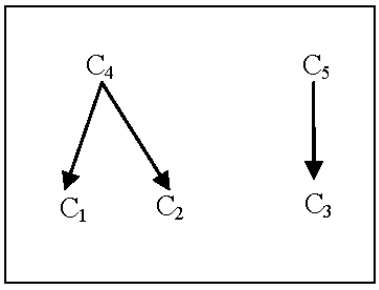

Figure 4. Directed acyclic graph of the candidates.

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/SecurityInformation/*/Industry` 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. Figure 4 illustrates the DAG constructed for the expanded set of candidates for our running example. We start with the 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. 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 (Algorithm 7). That is, we replace general indexes whose additional benefit per unit cost over their children is the 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, 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 in Section 6.4. We refer to this version of the search algorithm as *top down full*.

Algorithm 7 topDownSearch($topCandidates$)

1: $candidates \leftarrow topCandidates$
2: $currSize \leftarrow candidates.size$
3: **while** $currSize > diskConstraint$ **do**
4: **for all** $cand \in candidates$ **do**
5: calculate $\Delta B/\Delta C$ of $cand$
6: **end for**
7: $candidates \leftarrow$configuration after replacing the candidate with minimum ($\Delta B/\Delta C$) with its children
8: $currSize \leftarrow candidates.size$
9: **end while**

7.3 Dynamic Programming

Both algorithms described above find approximations to the optimal solution. To find an optimal solution, we use a dynamic programming algorithm that searches the exponential space of possible configurations but does not consider all possible index interactions.

We implemented a dynamic programming algorithm given in [23], which has a time and space complexity of $O(min\{2^n, n\sum_{1 \le i \le n} p_i, nm\})$. To decide whether to include an index in the solution, this algorithm tries to add it to all configurations that were computed so far, keeping a dynamic programming table that caches the optimal subconfigurations that were evaluated so far. To account for index interaction, we evaluate the benefit to the workload of having all the indexes in a configuration. Since we consult the optimizer to evaluate the workload execution cost for each configuration, this adds a high overhead to the solution. This solution suffers from pruning some configura-

tions in the space because of their low overall benefit. But because indexes can have higher benefit to a query when combined with other indexes, the low benefit configurations might turn to be the best ones later on.

8 Experiments

8.1 Experimental Setup

IBM DB2 9 (pureXML) supports both relational and XML data [6, 30]. 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 used two XML benchmarks for our experiments: the recent TPoX [29] benchmark and XMark [36]. We generate the data for both benchmarks using a scale factor of 1GB. For both benchmarks, we evaluate our XML Index Advisor on the standard queries that are part of the benchmark specification: 11 XQuery queries for TPoX and 16 XQuery queries for XMark. To illustrate the effectiveness of our generalization algorithm, we also use synthetic queries on the TPoX data in Section 8.3.

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 [30]. 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 XML-typed columns, 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.

In the following sections, we illustrate that our XML Index Advisor makes good index recommendations that effectively use the available disk space budget and that it is efficient in terms of run-time. We also show that by using the top down search algorithm, the advisor can recommend general configurations that are useful beyond the training workload. Furthermore, we demonstrate the accuracy of the statistics we create for cost estimation in the Evaluate Indexes mode, and of our estimation of the cost of updating indexes.

8.2 Effectiveness of Recommendations

We have implemented five different combinatorial search strategies in our Index Advisor. The five strategies are described in Section 7: (1) greedy search (without heuristics), (2) greedy search with the heuristics, (3) top down lite, (4) top down full, and (5) dynamic programming.

Figures 5 and 6 show the estimated speedup for the different search strategies with varying disk space budgets for the TPoX and XMark benchmarks, respectively. The figures also show 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 for TPoX and 149MB for XMark. In these figures, there is no generalization to unseen queries in the workloads. We use the benchmark queries (11 for TPoX and 16 for XMark) for recommending the indexes and also for 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 5. 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 7 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 on the TPoX workload. 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. The figure does not show the run time of greedy search or dynamic programming. Greedy search is faster than the greedy search with heuristics, and dynamic programming is more than 5 times slower than top down full search.

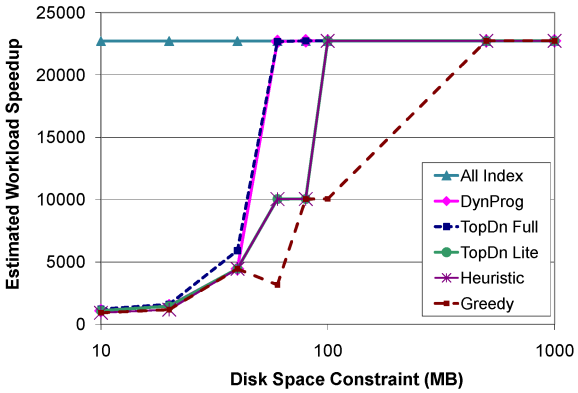

Figure 5. Estimated speedup (TPoX).

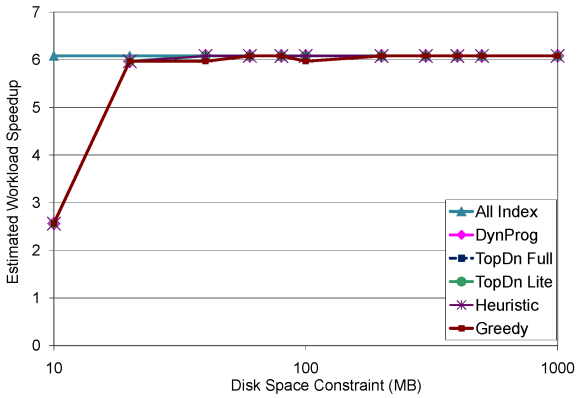

Figure 6. Estimated speedup (XMark).

8.3 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 random XPath path expressions that occur in the data. Table 6 shows for TPoX 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 locality, we are able to expand the number of candidate indexes by more than 25% 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

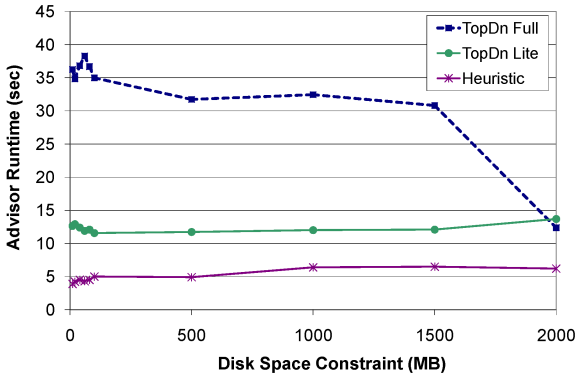

Figure 7. Advisor runtime (TPoX).

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

Table 6. Number of candidates (TPoX).

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.

Tables 7 and 8 show 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 for the 11 TPoX and the 16 XMark benchmark queries, respectively. 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 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

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

Table 7. Number of general (G) and specific (S) indexes recommended (TPoX).

Disk Budget	Heuristics	Top Down Lite	Top Down Full
100MB	G: 1, S:10	G: 0, S: 13	G: 0, S: 13
500MB	G: 1, S: 11	G: 3, S: 5	G: 5, S: 5
1000MB	G: 2, S: 4	G: 3, S: 5	G: 5, S: 5

Table 8. Number of general (G) and specific (S) indexes recommended (XMark).

workload used to evaluate the recommended configuration. For TPoX, we used a workload of 20 queries, the 11 TPoX queries followed by 9 synthetic queries generated as described above to increase workload diversity. For XMark, we used the 16 benchmark 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 in our experiments). Figures 8 and 9 show the estimated speedup on the test workload as we vary the training workload size for TPoX and XMark respectively, with a disk space budget of 2GB. The figures show 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 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 from the TPoX data 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 10 shows the *actual* speedup corresponding to Figure 8. 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! The figure shows actual speedup for the remaining 9 queries of the TPoX benchmark, and we can see that the actual speedup corroborates the conclusions drawn from our estimated speedup experiments.

8.4 Evaluating Candidate Configurations

The quality of the configurations recommended by the XML Index Advisor depends on how accurate we are in estimating the benefit of candidate configurations in the Evaluate Indexes optimizer mode, and in estimating in our client-side application the penalty of updating the index

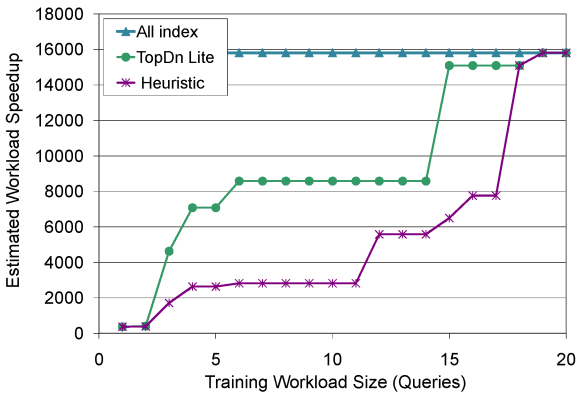

Figure 8. Generalization to unseen queries (TPoX).

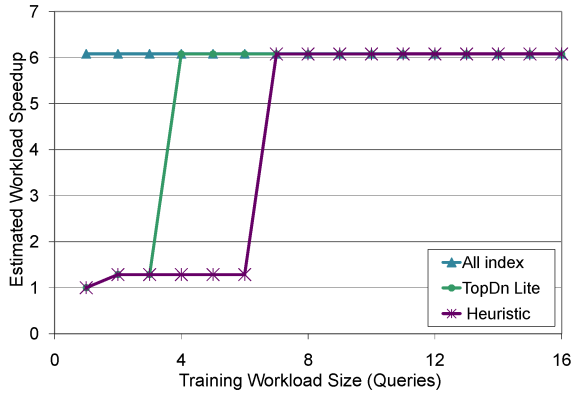

Figure 9. Generalization to unseen queries (XMark).

when updating the database with update, insert, or delete (UDI) statements.

The key statistic used by Evaluate Indexes mode is the size of a virtual index. We have found that for the TPoX and XMark workloads, the median relative estimation error for this statistic is 12% and 11%, respectively. Notably, we are able to estimate size of large indexes – which have the most impact on performance – with a very small error. For example, the largest candidate indexes for TPoX were indexes on `/FIXML/Order/OrdQty/@*` and `/FIXML/Order//@*`, and we were able to estimate their size with 3.7% and 5.5% error, respectively.

In our client-side application, we estimate the penalty of updating candidate configurations if the workload contains UDI statements. Figure 11 illustrates the effect of this estimation. 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 500MB disk space budget. The figure shows execution times (in millions of optimizer time units, or *timerons*) as we vary the number of UDI state-

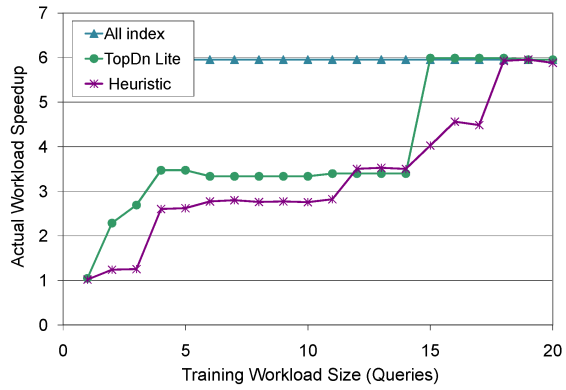

Figure 10. Generalization to unseen queries - Actual speedup (TPoX).

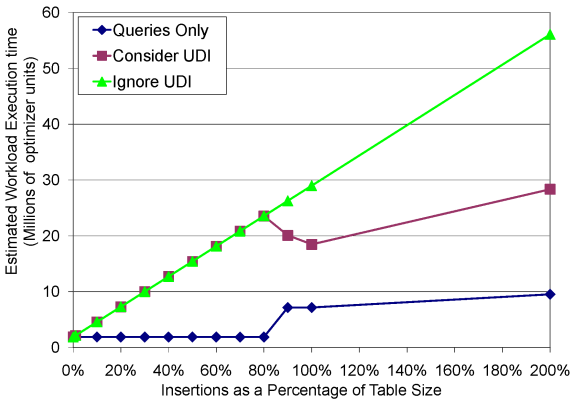

Figure 11. Effect of updates (TPoX).

ments. The figure shows the case where the design advisor ignores UDI statements while recommending a configuration, and for the case where the design advisor takes UDI statements into account. 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 around 80% of the table size). The figure also shows that the queries in the workload suffer when indexes are dropped, but dropping the indexes saves overall time.

9 Conclusions

In this paper, we have presented an XML Index Advisor that recommends the best set of XML indexes for a given database and query workload. Three key features of our Index Advisor are (1) it is tightly coupled with the query optimizer, (2) the search algorithms it employs can recommend indexes that are useful not only for the given workload, but also for other similar workloads that may be seen in the future, and (3) we always minimize the number of optimizer calls.

Our index advisor uses the novel notion of a virtual universal index to leverage the query optimizer for recommending candidate indexes. The advisor expands the set of candidate indexes recommended by the optimizer to include additional, more general candidates that could benefit multiple queries in the workload or other queries not seen in the workload. From this set of candidates, our XML Index Advisor chooses an optimal index configuration, taking into account the disk space consumed by the indexes and the additional cost they impose on update, delete, and insert statements. The advisor leverages the costing capabilities of the optimizer to estimate the benefit of candidate index configurations.

Our index advisor can employ a variety of combinatorial search algorithms to find the optimal configuration depending on the goal of the user, whether it is finding a configuration that is best only for the given workload, or finding a configuration that is as general as possible and so can help a wide variety of workloads. If the run time of the advisor is not a concern, then top down full search is an algorithm that can simultaneously accomplish both goals as best as possible, depending on the training workload and the available disk space budget. For a more efficient search that reduces the advisor run time, greedy search with heuristics is the algorithm of choice for finding configurations that are useful only for the specific workload, and top down lite search is the algorithm of choice for finding general configurations that make the best possible use of the available disk space.

We have implemented our XML Index Advisor in a prototype version of DB2, and our experiments with this implementation show that our Index Advisor can effectively recommend indexes that result in significant speedups for workload queries.

References

[1] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *VLDB*, 2004.

[2] A. Balmin, K. S. Beyer, F. Özcan, and M. Nicola. On the path to efficient XML queries. In *VLDB*, 2006.

[3] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. E. Simmen, M. Wang, and C. Zhang. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2), 2006.

[4] A. Balmin, F. Özcan, K. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[5] K. Beyer, R. Cochrane, M. Hvizdos, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, R. Lyle, M. Nicola, F. zcan, H. Pirahesh, N. Seemann, A. Singh, T. Truong, R. C. V. der Linden, B. Vickery, C. Zhang, and G. Zhang.

DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal*, 45(2), 2006.

[6] K. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B. Van der Linden, B. Vickery, and C. Zhang. System RX: One part relational, one part XML. In *SIGMOD*, 2005.

[7] P. Bohannon, J. Freire, J. R. Haritsa, and M. Ramanath. LegoDB: Customizing relational storage for XML documents. In *VLDB*, 2002.

[8] 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*, 17(12):1595–1609, 2005.

[9] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *VLDB*, 1997.

[10] C.-W. Chung, J.-K. Min, and K. Shim. APEX: An adaptive path index for XML data. In *SIGMOD*, 2002.

[11] D. K. Daniela Florescu. Storing and querying XML data using an RDBMS. In *VLDB*, 2003.

[12] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *SIGMOD*, pages 623–634, 2003.

[13] L. Ennser, C. Delporte, M. Oba, and K. M. Sunil. *Integrating XML with DB2 XML Extender and DB2 Text Extender*. IBM Redbooks, 2000. Available at: `http://www.redbooks.ibm.com/redbooks/pdfs/sg246130.pdf`.

[14] eXist: An Open Source Native XML Database. Available at: `http://exist.sourceforge.net/`.

[15] G. Feinberg. Native XML data storage and retrieval. *Linux Journal*, 2005.

[16] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. Silkroute: A framework for publishing relational data in xml. *ACM Transactions on Database Systems*, 27(4):438–493, 2002.

[17] T. Fiebig and et. al. Anatomy of a native XML base management system. *VLDB Journal*, 11(4), 2002.

[18] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1), 1988.

[19] 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.

[20] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. D. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed mode xml query processing. In *VLDB*, 2003.

[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 (DEXA)*, 2004.

[22] 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.

[23] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms in C++*. Computer Science Press, 1998.

[24] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *The VLDB Journal*, 11(4), 2002.

[25] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[26] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.

[27] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[28] R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A.-T. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy. Towards an enterprise XML architecture. In *SIGMOD*, 2005.

[29] M. Nicola, I. Kogan, and B. Schiefer. An XML transaction processing benchmark. In *SIGMOD*, 2007. Benchmark Available at: `https://sourceforge.net/projects/tpox/`.

[30] M. Nicola and B. Van der Linden. Native XML support in DB2 universal database. In *VLDB*, 2005.

[31] Oracle Corp. *Oracle Database 10g Release 2 XML DB Technical Overview*, 2005. Available at: `http://www.oracle.com/technology/tech/xml/xmldb/`.

[32] A. Prassinos and A. Tarachandani. Using XMLIndex and Binary XML for Motorola BIS. Presentation at Oracle OpenWorld, 2006. Available at: `http://www28.cplan.com/cbo_export/PS_S281187_281187_139-1_FIN_v2.pdf`.

[33] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[34] 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.

[35] M. Rys. XML and relational database management systems: Inside Microsoft SQL Server 2005. In *SIGMOD*, 2005.

[36] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical Report INS-R0103, CWI, 2001.

[37] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, 2002.

[38] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[39] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[40] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *VLDB*, 2004.