Automatic Physical Design for XML Databases

by

Iman Elghandour

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Database systems employ physical structures such as indexes and materialized views to improve query performance, potentially by orders of magnitude. It is therefore important for a database administrator to choose the appropriate configuration of these physical structures (i.e., the appropriate physical design) for a given database. Deciding on the physical design of a database is not an easy task, and a considerable amount of research exists on automatic physical design tools for relational databases. Recently, XML database systems are increasingly being used for managing highly structured XML data, and support for XML data is being added to commercial relational database systems. This raises the important question of how to choose the appropriate physical design (i.e., the appropriate set of physical structures) for an XML database. Relational automatic physical design tools are not adequate, so new research is needed in this area.

In this thesis, we address the problem of automatic physical design for XML databases, which is the process of automatically selecting the best set of physical structures for a given database and a given query workload representing the client application's usage patterns of this data. We focus on recommending two types of physical structures: XML indexes and relational materialized views of XML data. For each of these structures, we study the recommendation process and present a design advisor that automatically recommends a configuration of physical structures given an XML database and a workload of XML queries. The recommendation process is divided into four main phases: (1) enumerating candidate physical structures, (2) generalizing candidate structures in order to generate more candidates that are useful to queries that are not seen in the given workload but similar to the workload queries, (3) estimating the benefit of various candidate structures, and (4) selecting the best set of candidate structures for the given database and workload. We present a design advisor for recommending XML indexes, one for recommending materialized views, and an integrated design advisor that recommends both indexes and materialized views. A key characteristic of our advisors is that they are tightly coupled with the query optimizer of the database system,

and rely on the optimizer for enumerating and evaluating physical designs whenever possible. This characteristic makes our techniques suitable for any database system that complies with a set of minimum requirements listed within the thesis. We have implemented the index, materialized view, and integrated advisors in a prototype version of IBM DB2 V9, which supports both relational and XML data, and we experimentally demonstrate the effectiveness of their recommendations using this implementation.

Acknowledgements

Thanks to Allah the most merciful for all the gifts He has bestowed me with throughout my life.

I am grateful to my supervisor Prof. Ashraf Aboulnaga for his continuous support, encouragement, and guidance. Nominating me to work on the XML Design Advisor project was the seed for this thesis and has led to many positive experiences throughout the following years. Prof. Aboulnaga, you were always patient and understanding. You always had time to give me thorough feedback on various papers and presentations. You made tasks that seemed overwhelming and daunting manageable. I have learned a lot from you. Prof. Aboulnaga; I am fortunate to have worked with you.

I would like to thank my committee members Prof. Frank Tompa and Prof. Tamer Özsu for their valuable feedback on my proposal, which helped in shaping this thesis and for their final comments on this work. I would also like to thank my external examiner, Prof. Neoklis Polyzotis from the University of California Santa Cruz, for his interest in this thesis and his valuable comments, and my internal external examiner, Prof. Lin Tan, for her time and comments.

Most of this research has been done as part of the XML Design Advisor project, which is an IBM CAS (Center for Advanced Studies) project. I was financially supported by an IBM CAS fellowship for the first three years of my PhD. In the following two years, I was supported by an IBM PhD fellowship.

I would like to thank Danny Zilio, Calisto Zuzarte, and Fei Chiang from IBM Toronto Lab for their valuable guidance and support while working with them on the XML Design Advisor project. Working with them has contributed a lot to my experience. I learned from Danny about how research ideas and prototypes can be realized into code that meets industrial standards. Calisto was always a strong supporter of me and my work, and his advice and guidance was always valuable. I would also like to thank Kevin Beyer and Andrey Balmin from the IBM Almaden Research Center for their contribution to the first part of this research.

I would like to thank my parents, Afaf Nada and Ibrahim Elghandour, my beloved grandmother, Hana, my dearest brothers, Mohamed and Ahmed, and my sister-in-law Hager Elgendy for your prayers, patience, support, and encouragement. I always felt that things were easier for me because of your prayers.

During my PhD program I have met many nice people. Special thanks to Gloria Ichim, Iman Thabet, Hadir Abdel Rahman, Safaa Bedawi, and Zainub Ibrahim who positively supported me during my down times. Thanks to my friends Amal AbdelKarim, Yara Osman, Samah Shady, Mona Adel, and Yasmine Abouelseoud who where always there for me. Your emails and encouragement really made a difference.

Finally, I would like to thank Nicole Keshav for helping me with my technical writing during the first two years in my PhD program. Nicole's comments were a great help in improving my writing.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Recently, an increasing amount of data is exchanged, processed and stored in XML format. In addition, XML is now commonly used in many applications to represent and exchange semi-structured data. This has led to an increased focus on XML data management. There are three main approaches for storing and managing XML data: (1) Native XML databases, (2) Shredding XML data into relational databases, and (3) XML column type. In this thesis, we focus on XML data that is stored in an XML column in a table in a relational database. This approach is now supported by most commercial database systems [15, 73, 76].

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

The various XML physical structures can potentially improve the performance of XML database systems by orders of magnitude, but users of these systems now face the problem of deciding on the best set of physical structures to create for a given XML database and query workload. Automatic physical database design has been studied extensively in the context of relational databases, and most commercial database systems now include *Design Advisors/Tuners* that automatically recommend various physical structures [3, 22, 23, 87, 92]. However, automatic physical design for XML databases has not been studied as extensively in the database literature.

In this thesis, we study automating the physical design of XML databases. We answer the question "what physical structures are useful for an XML database and an XML query workload?" and build a system to recommend these physical structures given an XML database and query workload. We study the automatic recommendation of two physical structures: XML indexes and XML materialized views. A well established architecture for physical design recommendation has been developed in the context of relational physical design advisors. A design advisor needs to address four questions: (1) how to determine the candidate structures that would be useful for a query or a workload consisting of a set of queries, (2) how to expand the candidates with more general ones, (3) how to estimate the benefit of a physical design configuration (i.e., a set of physical structures), and (4) how to search all the possible configurations for the best one. The recommendation process is divided into several phases where each phase addresses one of these questions. Figure 1.1 shows the general architecture of relational database design advisors, which we follow in our proposed XML design advisors. The query optimizer is extended with operation modes that allow it to accomplish two tasks: (1) recommend physical structures that can be useful for a query, and (2) construct a query execution plan and estimate its cost while assuming the existence of some physical structures.

XML databases have unique characteristics and so their physical structures are also different from the ones that are defined and built for relational systems. The

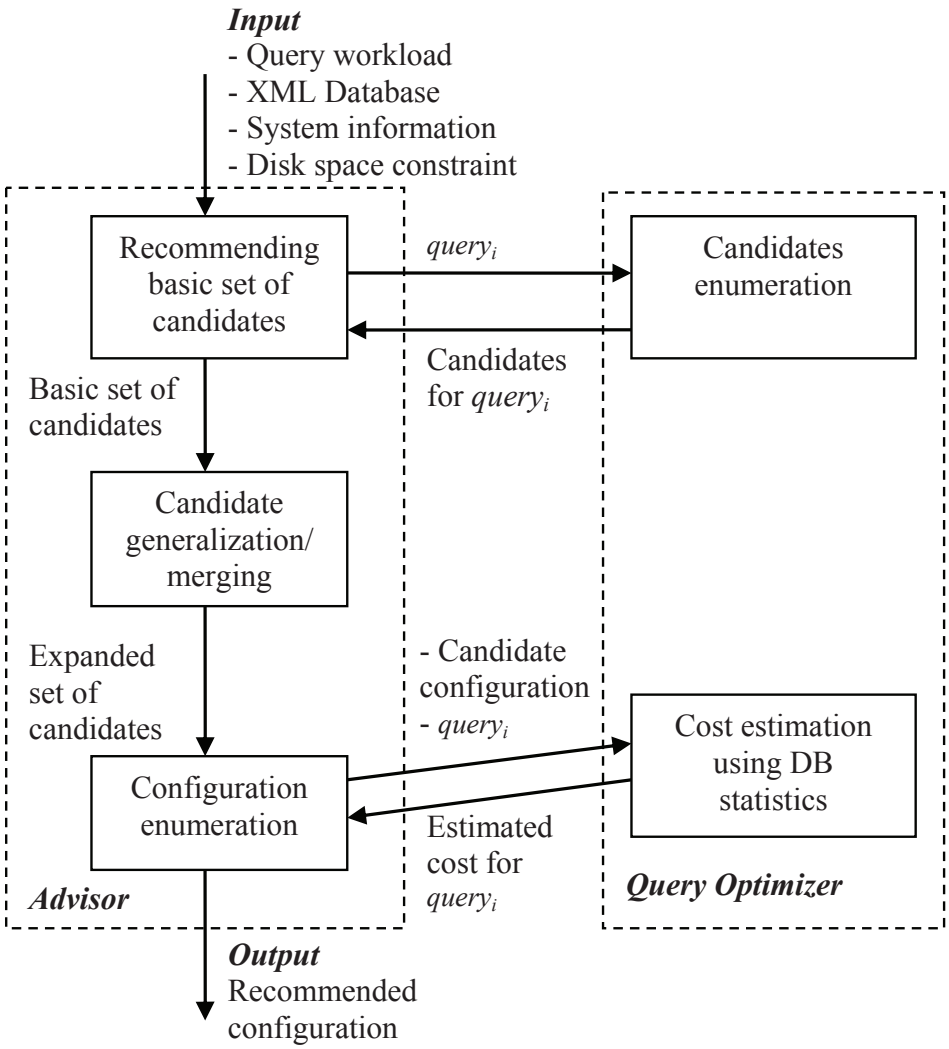


Figure 1.1: General architecture of a physical design advisor.

unique challenges introduced by XML databases make automatic physical design for XML databases more difficult than that for relational databases and lead to the details of the physical design procedure being significantly different. Also, the physical structures for XML databases are not yet well established and so there is an opportunity for research on automatic physical design to impact the definition of the physical structures being recommended. For example, a wide variety of XML indexes have been explored in the literature [24, 41, 51, 53, 64, 71, 73, 74]. On the other hand, XML materialized views of various types are still being investigated in research [1, 7, 44, 73]. In this thesis, we explore using the result of the XMLTable

functions [49, 73] as relational-structured materialized views to speed up answering XQuery queries, and we develop an advisor that recommends XMLTable views for a given workload of XQuery queries.

In this thesis, we focus on developing techniques and algorithms to automate the recommendation of XML indexes and XMLTable materialized views for a given XML database and XML query workload. We present two end-to-end advisors: an XML Index Advisor and an XMLTable View Advisor. We then incorporate these two advisors into one integrated advisor that recommends both XML indexes and XMLTable materialized views for an XML database.

To test the validity of our proposed techniques, we have implemented our index, view and integrated advisors on top of IBM DB2 9 [30, 31, 32, 34]. To support our index advisor, we have implemented optimizer extensions in a prototype version of IBM DB2 9, which supports both relational and XML data. These extensions have been incorporated into the mainline code of DB2 and are available in versions of DB2 starting with DB2 9.7. Our XML Index Advisor application, which uses the server-side extensions that we have implemented, is available for download from the IBM alphaWorks web site [33].

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents a summary of research areas that are related to this work. The following chapters present our contributions.

In Chapter 3, we describe our proposed architecture for building an XML Index Advisor. We start by describing all the required phases of the recommendation process and the new components added to the database query optimizer. Next, we present our algorithms for recommending two common types of XML indexes: linear and multipath XML indexes. Finally, we describe our implementation of an XML Index Advisor for IBM DB2 that recommends linear XML indexes and present the results of an experimental evaluation.

In Chapter 4, we focus on one of type of materialized views that have been proposed in the literature for XML databases, namely XMLTable views. We describe the recommendation process for XMLTable views given an XML database and XQuery workload. Finally, we present our implementation and evaluation results for our XMLTable View Advisor.

In Chapter 5, we combine our index and view advisors proposed in Chapters 3 and 4 to build an integrated advisor. We compare the results from the XML Index Advisor (Chapter 3), the XMLTable View Advisor (Chapter 4), and the Integrated Index-View Advisor we present in this chapter to show the efficiency of our integration techniques.

We conclude by summarizing our contributions in Chapter 6. We also present some directions for future work. Finally, we include the following three appendices. Appendix A describes the details of the process we followed to verify that the estimation of the penalty incurred by the update statements because of the presence of XML indexes in the database is close to the actual value. Appendix B lists the workloads that we have used in our experiments. Appendix C describes in detail a run made by the advisor and the various decisions made by its different modules.

Chapter 2

Background and Related Work

2.1 XML Databases

Several approaches have been proposed in the database research literature for storing and querying XML data. We note the following approaches:

- **Native XML databases.** In this approach, an XML document is the logical unit of storage and a logical model is defined for querying these documents. One of the early attempts to build a database management system for semi-structured data using the native storage approach is Lore [64]. Natix [37] and eXist [35, 65] are other examples of database systems that are designed for natively storing and processing XML data.

- **Shredding XML data into relational databases.** This approach attempts to benefit from the mature technology in relational database systems by shredding XML data and storing it in relational tables [17, 18, 39, 26, 27, 84, 91]. These shredding techniques are well studied in the literature. They range from schema oblivious storage where XML elements are stored in relational tables regardless of their schema [39, 26, 27, 89], to schema aware approaches where the schema of the XML data is used to determine a good relational storage strategy [18, 81].

- **XML column type.** Recently, major commercial database vendors have added support for XML storage and querying to their relational database systems by expanding their column type definition to include an *XML column type* [15, 73, 76]. This extension involves not only supporting a new storage type but also extending the query processor and optimizer to handle XML queries, in particular XQuery and SQL/XML queries [14, 58]. Database systems that support an XML column type also support building XML physical structures that speed up query processing, such as XML indexes [71].

In this thesis, we focus on recommending physical designs for XML data that is stored natively. We have developed our design advisors on top of a relational database system extended with an XML column type, but our proposed algorithms can also be used for XML data that is stored and managed in a native XML database: an XML query processor and the support of XML physical structures are the two main requirements for our framework.

2.2 XML Query Languages

There are several programming and query languages that can process XML. In this section, we briefly describe XPath, XQuery and SQL/XML. XPath [25] is a native XML programming language that is used to address (i.e., refer to) parts of an XML document by modeling the document as a logical tree of nodes and then operating on them. XPath enables users to identify different parts of an XML document, but does not support more complex operations. XQuery [16] and SQL/XML [67] are query languages that can handle more complex operations on XML such as joins, aggregation, or user defined functions. Both these languages use XPath path expressions to extract data in XML documents.

XQuery is a native XML query language that is supported by most major database vendors. It is an extension of the XPath query language to allow integrating data from different data sources and creating new XML structures as the result of queries. An XQuery query has four main blocks (or clauses): FOR, LET,

WHERE, ORDER BY, and RETURN. XQuery queries are referred to as *FLWOR expressions* or *FLWOR blocks*, pronounced "flower."

SQL/XML is an extension of the SQL query language with new functions that handles XML data. It is mainly useful when queries span relational and XML data sources. SQL/XML extends SQL with functions that allow retrieving XML data, creating new XML structures, and performing other essential XML data processing operations.

2.3 XML Indexes

The retrieval of elements from XML data 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 [24, 41, 51, 53, 64, 71, 73, 74]. XPath path expressions are usually used to specify the elements in the XML data that are being included in the indexes. XML indexes can be categorized into *structural indexes* and *value indexes*. We discuss in more details the different types of XML indexes and their implementation in commercial database systems in Chapter 3.

2.4 XML Materialized Views

Creating views of relational and XML data can take place on either the logical or physical level or both. On the logical design level, data can be XML and be published as relational views [44, 63], or data can be relational and be published as XML views [19, 28, 36, 63, 80]. Queries are written according to the published schema, so if, for example, the published schema is XML and the data is stored in relational format, we need to (1) translate the XML queries to SQL queries according to the stored schema, and (2) transform the XML data to relational to be stored in the relational store, and vice versa for query answers.

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

1. Views of XML data fragments that are defined by XQuery queries [7, 72]. The queries written against the views are also in XQuery. Result containment is checked to decide if a view can be used to answer a query.

2. Views of XML data fragments that are defined by XPath path expressions [12, 61, 88]. Queries can be either XPath or full XQuery. In the latter case, indexes containing fragments of the data constitute the XML views that are used to answer the XQuery queries.

3. Views of XML data elements and their values that are defined by XPath path expressions and stored in relational tables. Queries can be written using XQuery or SQL/XML. When XQuery queries are used, they need to be translated into SQL queries to be executed on these materialized relational views. The expected benefit of using these views stems from replacing XQuery queries on the XML data by SQL queries on the relational materialized views which saves the navigation of the XML data. This approach has some similarities to shredding the XML data into relational tables [17, 18, 39, 62, 81, 84]. We adopt this approach in this thesis and elaborate on it in Chapter 4.

2.4.1 XMLTable Views of XML Data

Using relational materialized views for XML data and queries allows us to benefit from the rich and mature infrastructure for these views built into many database systems. Using these views provides a simple and effective way to improve the performance of XML query workloads by leveraging existing database system infrastructure. However, building relational views of XML data requires a mechanism that maps between XML elements and their corresponding column names in the relational views. For example, in ROX [44], the XML Wrapper of IBM DB2 [50] is used to do this mapping. The XML Wrapper allows CREATE NICKNAME statements that include nicknames for XPath expressions in the XML document.

A new approach for creating relational views for XML data is to use the XMLTable function [1, 40, 73, 90]. XMLTable is an SQL table function that cre-

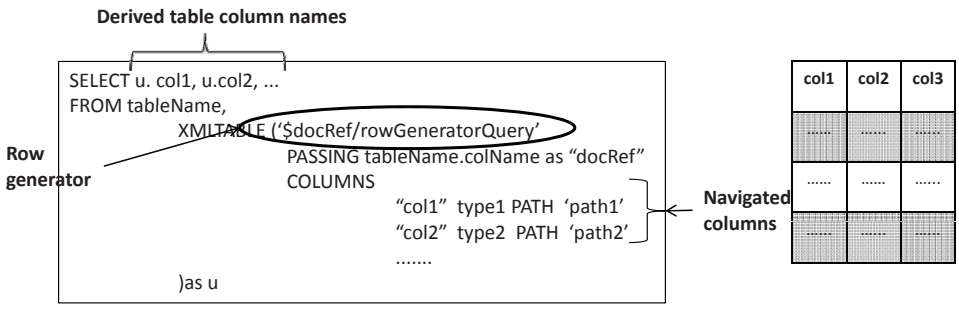


Figure 2.1: XMLTable view example.

ates a virtual derived relational table based on XML data. The virtual table can then be queried using SQL or materialized as a relational view [40]. An example of using the XMLTable function to create indexes is described in [60]. The XMLTable function is executed on a table with an XML-typed column. Figure 2.1 illustrates an example SQL query with an XMLTable function. The description of the syntax of the XMLTable function is as follows:

- A row generator XQuery string, which is an XQuery expression. The XMLTable function iterates through the results of the XQuery expression in the row generator and generates a tuple in the derived table for every one of these results. An XQuery expression can be an XPath expression and hence each element reachable by this path expression is used to create a row in the derived table.

- Column navigators are XPath navigation expressions that define the columns of the derived table to be created by the XMLTable function. Each column navigator entry is bound to the result of the XPath navigation expression and is used to populate a column in the derived table.

- An optional XMLNAMESPACES clause, which contains the namespace declarations that are referenced by the row generator and the column navigators (not shown in Figure 2.1).

- An optional passing clause, which specifies input values.

Consider the data shown in Figure 2.2, and assume that this data is stored in table `security` that has an XML column `sdoc`. Now, let us execute the SQL query

```
<?xml version="1.0" encoding="US-ASCII"?>
<Security id="10000“>
        <Name>Track Communications, Inc.</Name>
        <SecurityInformation>
                <StockInformation>
                        <Sector>Healthcare</Sector>
                </StockInformation>
        </SecurityInformation>
        <Yield>6.95</Yield>
</Security>

<?xml version="1.0" encoding="US-ASCII"?>
<Security id="10001“>
        <Name>Micro International Limited</Name>
        <SecurityInformation>
                <StockInformation>
                        <Sector>Conglomerates</Sector>
                </StockInformation>
        </SecurityInformation>
        <Yield>4.78</Yield>
</Security>
```

Figure 2.2: Example XML data.

in Figure 2.3 (which uses the XMLTable function) on this data. This query can be used to define a relational materialized view on the XML data. First, we execute the query in the row generator `/Security`. The resulting XML fragments are similar to the original data shown in Figure 2.2. Next, we iterate through these resulting XML fragments and generate a row for each one of them. For every resulting XML fragment, a further navigation is needed to process the path expressions in the column navigators. Finally, the extracted values from processing the column navigators are stored in the columns of the corresponding row. Table 2.1 shows the resulting relational table.

Using the XMLTable function to create materialized relational views of the XML data allows us to benefit from both the mature relational view matching [43] and also XPath view matching [12, 61, 88]. The XMLTable is defined in the FROM clause of a SELECT statement which allows two levels of matching of queries with views. The query optimizer matches queries that contain XMLTable functions with

```
SELECT u. Yield, u.Sector, u.Name
FROM security,
        XMLTABLE ('$sec/Security'
           PASSING security.sdoc as "sec"
           COLUMNS
                "Yield"  double  PATH  'Yield'
                "Sector"  varchar (20)  PATH  'SecInfo/*/Sector'
                "Name"   varchar(100)  PATH  'Name'
        ) as u
```

Figure 2.3: An XMLTable example.

Yield	Sector	Name
6.95	Healthcare	Track Communications, Inc.
4.78	Conglomerates	Micro International Limited

Table 2.1: Result of query with XMLTable.

XMLTable views. Next, XMLTable definitions of the query and view can use XPath matching to find the needed compensation and so to rewrite the query to use the contents of the view.

A discussion of the possible techniques and issues related to matching and rewriting SQL/XML queries with XMLTable functions to use XMLTable views is presented in [40]. That work focuses on describing the matching and rewriting rules needed by a query optimizer to use XMLTable views. The input queries to the optimizer are SQL/XML queries that have XMLTable functions. The materialized views stored in the database are XMLTable views which are similar to the ones described above. The row generators and column navigators are assumed to be XPath path expressions.

2.4.2 XQuery Views of XML Data

Query containment decides if a view contains the answer of a query. Containment in nested XML queries is studied in [29]. The authors show that the containment of conjunctive XML Queries (c-XQueries), which are analogous to select-project-join

queries in SQL, can be checked in polynomial time if the fanout (number of sibling sub-blocks) of the query is restricted to be 1[1]. The authors also show that (1) query containment for c-XQueries with arbitrary fanout but fixed nesting depth is coNP-complete, and (2) query containment for c-XQueries with arbitrary fanout is coNP-hard. Due to the complexity of checking query containment, and hence query rewriting, all the work on rewriting XQuery queries using XQuery views addressed views that are a restricted subset of XQuery. The main challenge of XML query containment is the nesting that can occur in the XML structures in the query results. Rewriting XQuery queries using views that are a subset of XQuery and allow nesting in the result structure is studied in [7, 72]. The work presented in [72] focuses on rewriting XQuery queries using views that are a subset of XQuery and feature nested FLWOR blocks. A different approach is studied in [7], namely rewriting XQuery queries in the presence of a structural summary of the document (for example, a Dataguide [41]) as a constraint. In that work, both queries and views are described by extended tree patterns [6], which are semantically close to XQuery. Using tree patterns to represent the queries and the views helps query matching and query containment of XQuery queries, and hence allows expanding the set of views that can be used to rewrite queries.

2.5 Automatic Physical Design for Relational Databases

Most major commercial database systems include physical design advisors that help database administrators automatically select a physical design for an input database and query workload. These advisors are coupled with the query optimizer to estimate the benefit of using candidate physical structures [3, 22, 38, 92]. In this section, we summarize the progress made over the last decade in the area of physical design for relational databases.

[1]We restrict ourselves to this class of queries in Chapter 4

2.5.1 What-if Analysis: Virtual Physical Structures

A common approach that is used by almost all commercial design advisors of relational database systems is using the optimizer cost model to estimate the cost of a query in the presence of a candidate physical structure. In [38], a new `Explain` statement, which triggers executing queries in a special optimizer mode, is introduced. In this optimizer mode, the optimizer estimates the cost of queries when hypothetical (virtual) physical structures are assumed to be present in the database. Moreover, building statistics for these hypothetical indexes is introduced in [38]. A detailed discussion of collecting statistics for such hypothetical indexes and of cost estimation formulas that use these indexes can be found in [87]. The new optimizer mode allows the physical design tool to use the optimizer's cost model instead of building a new cost model to estimate the cost of queries in the presence of different candidate physical structures. This process is called *what-if analysis* and a detailed description of a tool that performs this process appears in [22]. The tool uses the what-if analysis to help a database administrator choose between physical structures.

2.5.2 Automatic Recommendation of Relational Indexes

Two end-to-end relational index advisors are described in [23] and [87]. The architecture that we described in Chapter 1 (Figure 1.1) is followed, and hence the solutions presented in these papers address the following areas: (1) enumerating candidate indexes, (2) enumerating index configurations (sets of indexes) and selecting one of them according to a performance criteria and constraints, and (3) estimating the benefit of indexes and index configurations. Next, we describe some of the ideas developed for candidate index enumeration and index selection.

Candidate Enumeration

The number of indexes enumerated by the index advisor tool is considered as a measure of its efficiency. The efficiency of the index advisor is reduced when there

is a large number of candidate indexes to choose from. Yet, an advisor should not ignore candidates that can benefit workload queries. Columns that appear in `WHERE`, `GROUP BY`, and `ORDER BY` clauses are indexable columns that can be considered as candidates when searching for the best index configuration [23, 87].

Indexable columns provide candidate single column indexes. However, multi-column indexes are also important. An iterative algorithm that starts with single column indexes and then consider other combinations of multiple indexes that are admissible is proposed in [23]. According to the algorithm, all the columns that are included in a multi-column index must be important indexable columns themselves.

Enumerating Index Configurations and Searching for the Best Confirguartion

The number of indexable columns can grow very large for any workload. Hence, this large set of possible indexes is pruned into a smaller set before enumerating index configurations and searching for the best one that fits the performance criteria.

In [23], the best index configuration is selected for every query in the workload. The union of the resulting indexes constitutes the candidate indexes for the entire workload and a configuration search algorithm is then used to select a configuration of indexes of size k that has the lowest workload cost. An exhaustive search of all the index configurations is infeasible because the number of candidates can be large and hence an approximate search algorithm is used. To select the best index configuration for the entire workload, first, an exhaustive search is performed to select the first m indexes. Next, a greedy search is performed to select the remaining $k - m$ indexes.

In [87], a greedy search is performed to select indexes where the total size of the selected indexes can fit within a disk space constraint. Moreover, to reduce the number of optimizer calls, the optimizer is extended to include the enumeration algorithm and hence only one call to the optimizer is made by the index advisor tool to select the best index configuration.

2.5.3 Automatic Recommendation of Relational Materialized Views

Materialized views have richer structure than indexes, which makes automatically selecting a configuration of materialized views for SQL databases a nontrivial task compared to selecting indexes. Moreover, materialized views can have relational indexes on them. The approaches presented in [4] and in [92] select indexes and materialized views for a database given a workload of queries.

A large number of syntactically valid materialized views can exist for a workload of queries. However, considering all of these materialized views in the configuration enumeration process will limit the scalability and practicality of the materialized view advisor. Algorithms for enumerating candidate materialized views are introduced in [4, 92]. In [4], the candidate materialized views enumeration process is divided into the following steps:

1. **Finding interesting table subsets.** A new metric is defined to capture the usefulness of a table subset (i.e., columns of a table in the database) in materialized views. Only table subsets that have a weight according to this metric exceeding a certain threshold are considered in the following steps.

2. **Finding a configuration of materialized views for every query in the workload.** New rules are introduced to find the set of materialized views that are useful for a query based on the table subsets found in Step 1. Next, the best configuration of materialized views is selected for every query in the workload.

3. **Merging views found in the previous step.** If storage is constrained, then some of the views recommended for all the queries in the workload will not be recommended in the final configuration, leading to sub-optimal performance. A step of merging views is performed to try to avoid such sub-optimal recommendations.

In [92], the following steps are followed to enumerate candidate materialized views:

1. For every query in the workload, the combination of SELECT-FROM-WHERE-GROUP BY columns in the query is used to generate a starting set of candidate materialized views. Next, rules are applied to generalize the generated views. For example, predicates like `A=5` are generalized to `GROUP BY A`.

2. Multi-query optimization described in [57], is then used to find common subexpressions among multiple views.

2.5.4 Recommending Both Indexes and Materialized Views

Both indexes and materialized views can improve query execution. Yet, they compete for the same resources (for example, storage). There are two approaches for enumerating indexes and materialized views for a workload:

1. Selecting indexes followed by materialized views or vice versa. This approach has the drawback of missing opportunities because it ignores the interaction between indexes and materialized views and that their benefit to a workload of queries can increase due to the presence of other physical structures. In addition, a decision should be made about the amount of resources (typically storage) to be dedicated to each type of physical structure.

2. Using the design advisor's combinatorial search algorithm to select indexes, materialized views, and indexes on materialized views at the same time.

Work described in both [4] and [92] follows the second approach. Weights are introduced in [92] to favor one type of physical structure over the other.

2.6 XML Index Advisors

A few attempts were made to recommend indexes for XML data that is shredded and stored in relational databases [21, 42]. In [42], the proposed approach focuses

on a specific type of structural index that can be used over relational databases. The proposed solution cannot be generalized to other types of database systems and the proposed cost model is independent of the database system, which can lead to inaccurate estimates. In [21], 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 [46, 75]. 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 problem of generalizing the initial set of candidates, which is equivalent to merging physical design structures in relational databases [3]. 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 [75], 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 [75] are all path expressions that occur in the data, with some grouping of structurally equivalent candidate indexes based on schema information if this information is available. This method is inefficient because it leads to an uncontrolled explosion of the space to search for the best set of 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 [45, 46]. This index recommender analyzes the workload periodically and creates or drops XML indexes on the fly. As in [75], the cost model used is independent of the query optimizer and hence likely to be inaccurate. Candidate enumeration is not described. For configuration enumeration, [45] proposes using either a greedy search, which can be inaccurate, or an exhaustive search, which is slow. The configuration enumeration step in [45, 46] also ignores the penalty for updates, inserts, and deletes.

2.7 XML View Advisors

In Chapter 4 we propose an advisor that recommends relational materialized views (XMLTable views) for XML queries. In this section, we first discuss existing approaches that decide on how to store the data based on its characteristics. Next, we present previous cost based approaches that are used to recommend materialized views for XML databases. Finally, we briefly present background about XQuery to SQL translation which is required by the query optimizer to be able to rewrite XQuery queries using XMLTable views.

2.7.1 XML and Relational Storage

Relational and XML data reside side by side in current database systems [14]. Query execution cost depends on the storage mode of the data, and so there are situations where it is efficient to use a relational representation of the data and others where it is more efficient to use an XML representation. A discussion of the factors affecting the choice of using a relational or XML representation to store data is presented in [48, 56, 69]. The situations where using XML data representations is beneficial are as follows: (1) the schema is continuously evolving, (2) the data

is inherently hierarchical and hence the XML representation saves (a) normalizing the data when storing it and (b) reconstructing the data by joining tables at query time, (3) the schema has sparse attributes, and (4) the data has a complex structure and small components only have meaning in their context. On the contrary, using relational representation is more beneficial in these situations: (1) high performance is required and XML data parsing is frequently needed to answer queries, (2) data is required to be stored as relational, for example when we need to apply OLAP processing to the data in a data warehouse, and (3) the data is naturally tabular or can easily be normalized. Since the appropriate representation depends on the data and the usage scenario, the work in [48, 56, 69] attempts to find a logical design for a database given the characteristics of the data to be stored in it.

Application access patterns of the data can also help in choosing how to store this data. These alternative access patterns can be exploited to add materialized views to the database to enhance query execution performance [43]. To incorporate both relational and XML data models in the same database system, several hybrid XML-relational architectures are presented in [44]. In Chapter 4, we study building relational materialized views as an alternative access pattern for XML data.

2.7.2 Cost Based Recommendation of Materialized Views for XML Data

Another area where relational and XML data coexist is publishing relational data as XML, an area that has been extensively studied in the last few years. In these systems, data is stored in relational stores and published as an XML schema, which requires translating XQuery queries written for the published XML schema into SQL queries to be executed on the relational data stores, and translating relational data into XML data that satisfies the published XML schema. Most publishing techniques have one fixed way to translate the relational data into XML based on the XML schema. However, some research projects attempt using a cost based analysis for choosing the best translation [17, 28].

In MARS [28], the data is originally stored in relational and XML format. Partial relational and XML views of the data are also created. One virtual XML view is published and the incoming queries are translated according to the source that is chosen to answer them. A cost based analysis to choose the best query translation is proposed.

LegoDB [17] also uses a cost based approach to choose the mapping between XML and relational views of the data. The application using the data is represented by a workload of queries. A subset of the XML schema, called *p-schema*, is used to describe the data. P-schema has the advantage that it can be directly mapped to relational data, and it is also annotated with statistical information. Initially, different candidate p-schemas are enumerated. Then, a greedy heuristic search is used to find the best schema. The cost of a schema is estimated by mapping between the XML data and the relational storage, translating the XML workload according to this mapping, importing the XML statistics into the new relations, and finally, estimating the cost of the workload using a relational query optimizer.

An attempt to partially automate the logical design of a hybrid (Relational-XML) database is presented in [69]. The input to the proposed *Schema Advisor* is an annotated information model that is considered as a conceptual design for the database. Based on this annotated model, the schema advisor analyzes various storage alternatives and chooses the best of them according to a scoring function. Users of the system can also give their input to the tool to guide the advisor process.

Another cost based approach for automating the logical design of XML databases is proposed in the ULoad project [8]. That work uses the XML Access Modules (XAMs) algebraic formalism to represent the data and its storage structures. ULoad uses a fixed set of designs to choose from, but users can expand them with their own persistent data structures using the same graphical language. A structural summary of the data is then used to estimate the cost of answering a workload of queries given a configuration of XAMs.

2.7.3 XQuery Translation to SQL

In this thesis, we propose using relational materialized views of XML data to speed up the execution of XQuery queries on this data. This requires translating XQuery queries on the XML data to SQL queries on the materialized views. In the literature, XQuery translation to SQL has traditionally taken place at the application level, where the XQuery string gets translated into an SQL string before it is sent to the database server [39, 26, 27, 84, 89]. In comparison, XQuery native compilation, described in [59] takes place inside the database server. During XQuery native compilation, an XQuery query is compiled into the server internal data structures which are shared between XQuery and SQL queries.

The main focus in [59] is to rewrite XQuery queries into SQL queries using the SQL/XML extensions provided by the Oracle DBMS. This rewriting is done during query compilation to take advantage of the powerful capabilities of the full-fledged relational query optimizer. The first phase of XQuery compilation is to parse the query into the XQueryX [66] representation. Next, static type checking, which is important for XQuery optimization, is performed. Finally, the XQuery query is rewritten to an SQL/XML query. To rewrite XQuery to SQL/XML, each XQuery expression is converted into an SQL operator or operator tree or a sub-query block. In some cases, when native compilation is not possible (i.e. a mapping between XQuery and SQL/XML is not available), a hybrid approach is taken, and a co-processor is used to handle these parts of the XQuery query. In Chapter 4, we take a similar but simpler approach for XQuery to SQL/XML translation. We limit ourself to a subset of XQuery that can be mapped to SQL/XML with XMLTable functions.

Chapter 3

Recommending XML Indexes

Having presented the necessary background and survey of related work in the previous chapter, we now move to presenting the main technical contributions of the thesis. In this chapter we present the XML Index Advisor, and in Chapters 4 and 5 we present the XMLTable View Advisor and the Integrated Index-View Advisor, respectively.

3.1 Introduction

XML database systems employ various types of structural and value XML indexes to improve performance, potentially by orders of magnitude. Also, *partial indexes* are now supported by some commercial database systems [71, 73]. 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 [14]. 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 [10]. The large number of partial indexes that a user can choose makes the decision of which ones to build more difficult. In the rest of this

chapter, we present an XML Index Advisor that automatically recommends the best set of XML index patterns for a given database and query workload, while taking into account the cost of updating the index on data modification. XML index patterns can be linear patterns or general patterns that allow branching. We refer to general index patterns that allow branching as *multipath index patterns*. Hence, in this chapter, for each phase of the recommendation process, we outline the basic requirements for this phase, and we then describe our approaches for recommending both linear XML index patterns and multipath XML index patterns specific to this phase. In our implementation of the XML Index Advisor, and in our experimental evaluation using this implementation, we only recommend linear index patterns. Implementing and evaluating the recommendation of multipath XML index patterns requires more support from the database system (the index manager and query optimizer) than was available to us in DB2 at the time of this writing.

Approaches developed for recommending indexes for relational databases, discussed in the previous chapter, cannot be used for recommending indexes for XML databases due to some challenges introduced by XML. These challenges stem from the richness of XML query languages and the potential complexity of the structure of XML data. The primary use of XML indexes is to answer XPath [13] queries (path expressions) within XQuery queries and SQL/XML queries. We illustrate the challenges introduced by XML through an XPath example: XPath supports wildcards and descendant navigation, and XML data can be recursive. Thus, for any query, there can be several potentially useful 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 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 query optimizer calls to evaluate the benefit of index configurations.

[1]Throughout this thesis, we use examples from the TPoX benchmark [70].

Because of the added challenges, a new XML Index Advisor is needed. This advisor must address the following four questions: (1) how to determine the candidate indexes that would be useful for a query, (2) how to obtain candidate indexes that would be useful for multiple queries in the workload starting from those that would benefit individual queries, (3) how to estimate the benefit for a given query of a particular *index configuration* (i.e., a set of indexes), taking into account the increased cost of update statements due to indexes in this configuration, and (4) how to search the space of possible index configurations for the optimal configuration that provides the maximum benefit to the workload, while satisfying disk, schema, and other system constraints. We present our design choices in answering these four questions for XML indexes in the remainder of this chapter. These four question have been addressed for relational databases [23, 87] but our proposed solution is different for the following reasons:

1. While relational indexes are built on one or more table columns that can be enumerated by a relational index advisor, an XML index advisor also needs to identify the XPath path expressions to build indexes on. Multiple XPath expressions can exist in the same column and hence multiple indexes could reference the same column of a table.

2. Since XML indexes are different from relational indexes, their generalization process is inherently different.

3. We use what-if analysis to estimate the benefit of XML indexes. This process mainly depends on the virtual indexes that we create in the database and the statistics that we collect for them. We propose a new type of virtual indexes and new formulas for generating statistics on these virtual indexes.

4. New types of interactions between XML indexes can exist and we address them in the search algorithms that we propose.

We have implemented an XML Index Advisor that recommends linear indexes in a prototype version of IBM DB2 9, which supports both relational and XML

data, and we experimentally demonstrate the effectiveness of our techniques using this implementation. Part of this work appeared in [32] and [30], and it was demonstrated in [31].

The rest of this chapter is organized as follows. In Section 3.2, we describe the architecture of the proposed XML Index Advisor. We describe the details of the Index Advisor phases in the following sections. In Section 3.3, we introduce our algorithm for enumerating candidate XML indexes. Section 3.4 presents the generalization algorithm that we use to expand the index candidates. In Section 3.5, we describe the approach we used to estimate the benefit of indexes and index configurations. We then discuss the combinatorial search algorithms that we used to find the optimal index configuration in Section 3.6. Finally, we describe our implementation of an XML Index Advisor for DB2 and show our evaluation results for the proposed techniques.

3.2 Overview and Architecture

3.2.1 XML Indexes

XML query languages (for example, XQuery and SQL/XML) use XPath path expressions to represent elements to be retrieved from the data. The retrieval of elements from the XML data can be helped by the presence of an XML index, and there have been many proposals for XML indexes over the past few years. XML indexes can be categorized into *structural indexes* that speed up navigation through the hierarchical structure of the XML data (e.g., [53]), and *value indexes* that help in retrieving XML elements based on some condition on the values they contain (e.g., [71, 73]). 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).

Covering indexes (for example, DataGuide [41], 1-Index [68], and F&B-Index [2]) can grow as large as the data that they index [52], so they might not improve query execution time. In addition, large indexes are harder to maintain than smaller ones. Unlike covering indexes, several proposals of 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* [14, 73]. 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` because `/Security/*` only indexes elements that are immediate children of `/Security`.

An example of a database system that allows partial indexing of XML data is DB2. In DB2 [14, 15], XML data is 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 XML column of the table, a well formed XML document is stored. 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. Moreover, only elements that can be cast to the data type specified in the `create index` statement are included in this index. For example, the data definition language (DDL) statement for creating an index on an XML column `SDOC` of table `Security`, with index pattern `/Security/*`, is shown in Figure 3.1. Linear XML indexes are implemented in DB2 as B-trees, so they can use the same data structure and concurrency control and recovery algorithms as relational indexes. A detailed description of the XML indexes supported by DB2 appears in [14, 71]. Every entry of the index refers to one XML node stored in a row of the table. For a single row in the table, there can be zero, one, or multiple index entries. Every index entry includes:

1. A RowID to identify the row in the table and hence the document containing the indexed node.

2. A PathID to identify the path to the indexed node.

3. A NodeID to identify the specific node being indexes.

4. The value stored at that XML node.

```
CREATE INDEX securityVals_db2 ON Security(SDOC)
GENERATE KEY USING XMLPATTERN /Security/*
AS SQL DOUBLE
```

Figure 3.1: Example DDL for creating an XML index in DB2.

Partial indexes are also supported in Oracle 11*g* [73]. Oracle 11*g* allows two types of indexes: Function-Based Indexes and XMLIndex. Function-Based Indexes are similar to the XML indexes allowed in DB2, as one index includes the elements and values specified by a linear XPath expression. XMLIndex is a more general index that can include all the elements in the XML documents while allowing *path subsetting*, which is the inclusion and exclusion of path expressions. Elements that are reachable by multiple paths in the data are included in the XMLIndex, so it is an example of a multipath XML index. Path subsetting allows us to create smaller indexes and hence improves the performance of index maintenance. Every entry of the index includes:

1. A RowID, which specifies the row ID of the table.

2. A PathID, which is a unique identifier for the XPath path to the node.

3. An ORDER_KEY to identify the hierarchical position of the node.

4. A LOCATOR, which specifies the location of the corresponding document fragment.

5. A VALUE, which is the text of an attribute node or a simple element node.

Secondary indexes can be built on the XMLIndex to enhance the performance of index navigation. Example DDL statements for creating Function-Based and XMLIndex indexes are shown in Figures 3.2 and 3.3, respectively. The DDL statement

that creates a Function-Based index shown in Figure 3.2 includes all the elements that are reachable by the path expression `/Security/Symbol`. We can use an XMLIndex index to include elements reachable by multiple path expressions. For example, the DDL statement shown in Figure 3.3 creates an XMLIndex index that includes elements that are reachable by the path expressions: `/Security/Symbol` and `/Security/SecInfo//Sector`.

```
CREATE INDEX securityVals_ORACLE11g_fn_based ON Security
        (extractValue(SDOC, '/Security/Symbol'))
```

Figure 3.2: Example DDL for creating a Function-Based index in Oracle.

```
CREATE INDEX securityVals_ORACLE11g_XI ON Security(SDOC)
INDEXTYPE IS XDB.XMLINDEX
PARAMETERS ('PATHS (INCLUDE (/Security/*
                             /Security/SecInfo//Sector))')
```

Figure 3.3: Example DDL for creating an XMLIndex that uses path subsetting in Oracle.

In the rest of this chapter, we present algorithms for two types of XML indexes: linear XML indexes and multipath XML indexes. Linear XML indexes are similar to the XML indexes implemented in DB2 and the Function-Based indexes implemented in Oracle. Multipath indexes are similar to XMLIndex indexes in Oracle 11*g*. We represent linear path expression patterns as linked lists in which each node represents a path navigational step. Figure 3.4 shows the representation of the index patterns `/Security/Symbol`.

A linear index represents all the XML nodes reachable by the index pattern. Each entry in the index represents one of these nodes, and contains the value at this node, the record or document ID in which the node appears, and the node ID within this record or document. A query operator can use the index as a structural index or a value index. To use the index as a structural index, the query operator would scan all the entries in the index, thereby obtaining all XML nodes reachable

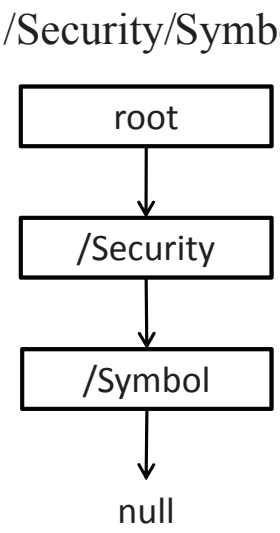


Figure 3.4: An example of the structure used to represent linear indexes.

by the XPath path expression used as the index pattern. To use the index as a value index, the query operator would provide a lookup value and get back all the nodes that contain this value.

Multipath indexes can be represented as a tree where the leaf nodes are the indexed nodes. For example, if the following two path expressions are included in the multipath index: `/Security/Yield` and `/Security/SecInfo/*/Sector`, then we represent them with an index tree that has the linear representation `/Security⟨Yield⟩/SecInfo/*⟨Sector⟩`. The nodes enclosed in angle brackets in the XPath expression are the nodes that are being indexed in this tree.

Another example of a multipath index is an index that includes these two path expressions: `/Security` and `/Security/Yield`. A representation for this index is `/⟨Security⟨Yield⟩⟩`. In addition, when indexing more than one element with the same ancestors, we use ∪ 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 ∪ PE⟩`.

Since multipath indexes have a richer structure than linear indexes, we change the internal representation of an index pattern from a list to a tree to accommodate the multiple indexed values. For this purpose, we use expression trees

with the same structure as the XPS trees defined in [12]. 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 (e.g., `/Secruity`) or a wildcard test (i.e., `/*`). Each node can have two children, the left child represents nodes to be included in the index (nodes enclosed in angle brackets), 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 if a node has an immediate descendant that is indexed, we check its left child. The XPS trees navigation would path through a left node and then right nodes to handle multiple elements being indexed along the same path, for example /⟨`Security`⟨`Yield`⟩⟩. Figures 3.5, 3.6, and 3.7 show the XPS tree representations of indexes `/Security`⟨`Yield`⟩`/SecInfo/*`⟨`Sector`⟩, /⟨`Security`⟨`Yield`⟩⟩, and `/Security`⟨`Yield` ∪ `PE`⟩, respectively. In our multipath index representation, we consider XPath index patterns that can contain both navigational steps and indexed elements. The navigation can contain label wildcards, `"*"`, child axis navigation, `"/"`, and descendant navigation, `"//"`. A union of elements enclosed in angle brackets indicates that these elements that share the same ancestors are being indexed. A query operator does a lookup in a multipath index by providing values for one or more of the indexed nodes in the index pattern. The index returns the node IDs and document (or record IDs) for the nodes that contain these values.

3.2.2 XML Index Advisor Architecture

The architecture of the XML Index Advisor is presented in Figure 3.8. 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 but related workloads. Finally, we search the space of possible index

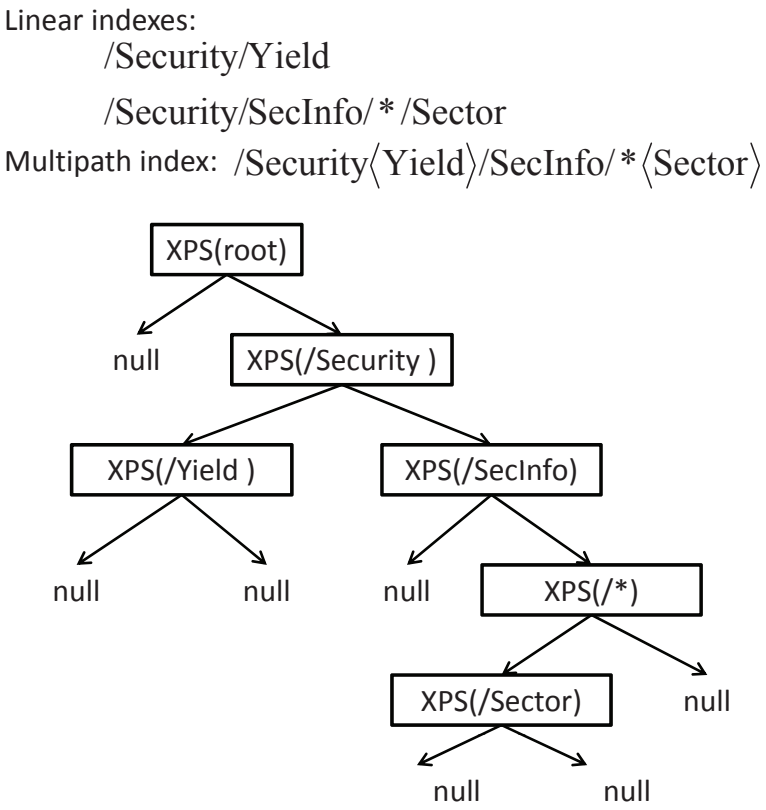


Figure 3.5: XPath XPS tree for the index pattern `/Security⟨Yield⟩/SecInfo /*⟨Sector⟩`.

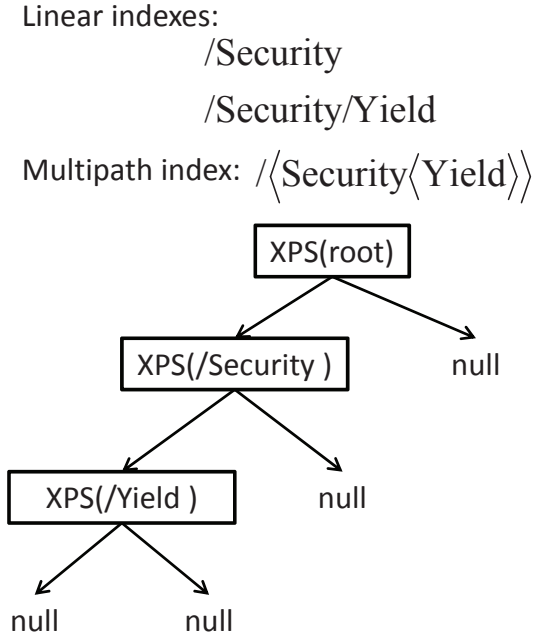


Figure 3.6: XPath XPS tree for the index pattern `/⟨Security⟨Yield⟩⟩`.

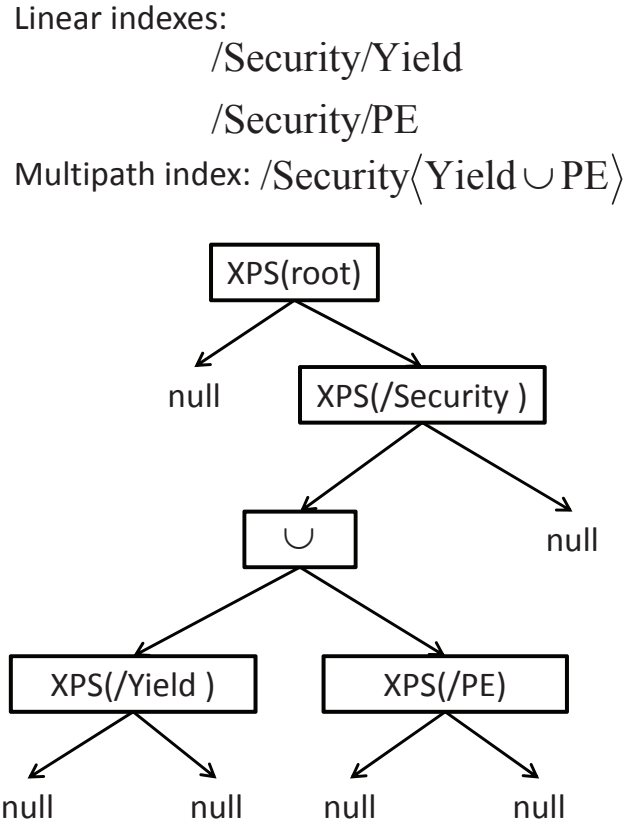


Figure 3.7: XPath XPS tree for the index pattern `/Security⟨Yield ∪ PE⟩`.

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 XML 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 XML 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,

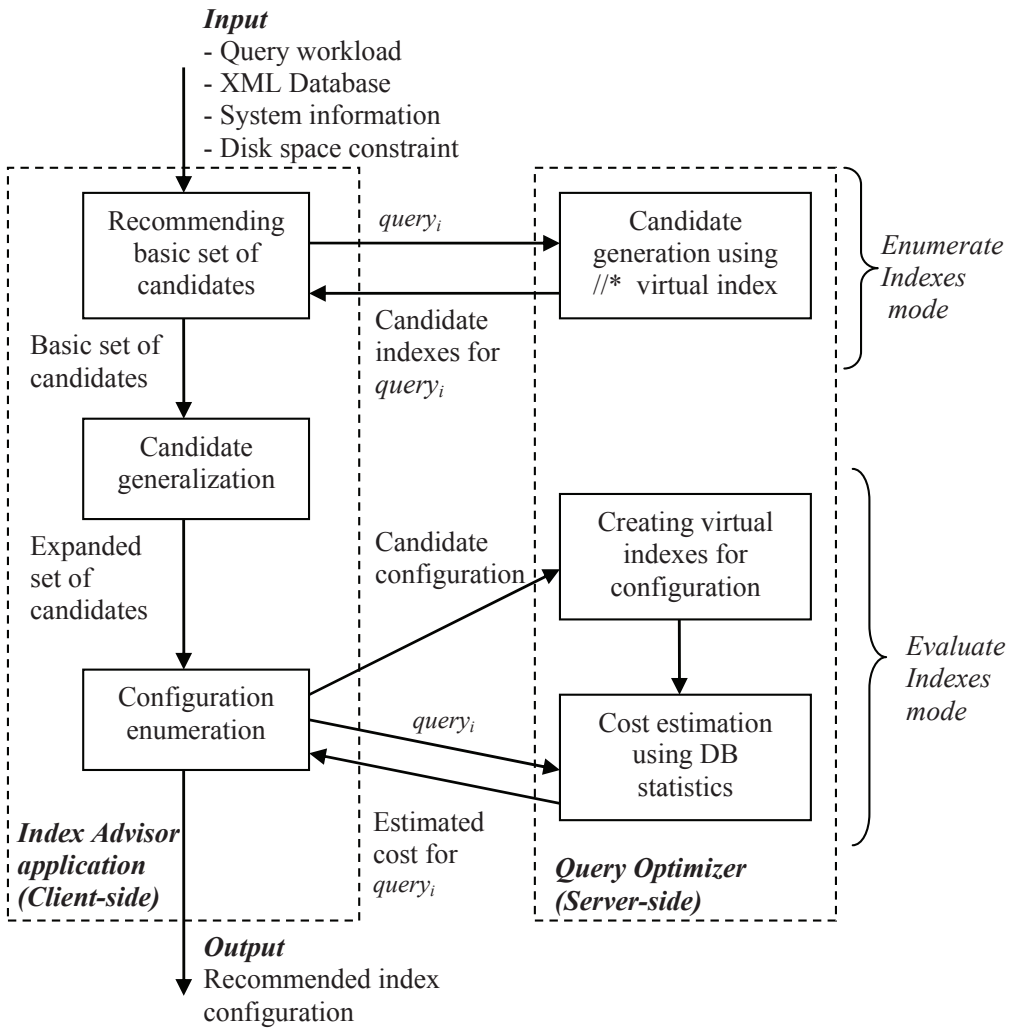


Figure 3.8: XML Index Advisor architecture.

and they eliminate the need to replicate any functionality that is already available in the optimizer. Moreover, the XML Index Advisor client application is now useful for any database system that supports XML indexes, and whose optimizer is extended with our proposed modes.

In the new modes, the optimizer needs to work with hypothetical indexes that do not exist but are still needed to identify candidate indexes or evaluate their cost. To enable this, we modify the query optimizer to allow it to create *virtual indexes* that can then be used during query optimization. These virtual indexes are added to the database catalog and to all the internal data structures of the optimizer, but they are not physically created on disk and no data is inserted into them. 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 [23, 87]. In our XML Index Advisor, we use virtual indexes for cost estimation, but a novel feature of our work is that we also use them for enumerating candidate indexes for workload queries.

Next, we describe the details of the XML Index Advisor recommendation phases for linear XML indexes. We also present our proposal for recommending general XML indexes as an extension to our proposal for linear XML indexes. However, we stress the fact that we have not implemented recommending multipath indexes in our current prototype. In the rest of the chapter, we use as a running example, a workload consisting of the following two queries from the TPoX benchmark [70].

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>
```

3.3 Basic Candidate Set

XQuery and SQL/XML are complex languages. In these languages, XML patterns can appear in various parts of the statement, but indexes are not useful for some of the XML patterns that appear in the queries (e.g., patterns that appear in the return clause [10]). 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) [12, 61, 88].

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 indexes that are usable by this system.

The XML Index Advisor optimizes each workload query in Enumerate XML Indexes mode. The resulting candidate index patterns of all queries are considered as a basic candidate set that is expanded in the generalization step. Next, we present our proposed techniques for enumerating candidate index patterns given a workload of queries for linear XML indexes (Section 3.3.1) and multipath XML indexes (Section 3.3.2).

3.3.1 Linear XML Indexes

To leverage the index matching capability of the query optimizer for enumerating candidate XML indexes, we modify the optimizer to create a special Enumerate XML Indexes query optimizer mode. In this mode, we create a *virtual universal index* over the XML data, which is a virtual index whose index pattern is `//*`. This `//*` *virtual index*, (virtually) indexes all elements in the document and hence can be

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

Table 3.1: Basic set of linear index candidates for queries *Q1* and *Q2*.

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, the optimizer returns to the user all the index patterns in the query that were matched with the `//*` virtual index and quits the query optimization without generating alternate query execution plans. Essentially, we have enabled the optimizer to answer the question: "If all possible indexes were available, which ones would be considered for this query?"

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 3.1 are the patterns enumerated by the DB2 optimizer for our example queries, *Q1* and *Q2*. All three candidates take predicates into account to determine the target nodes of the index patterns.

Our Index Advisor can work with any database system that can construct an XML index with index pattern `//*` and can match this `//*` with all the indexable patterns in the queries. Adding the Enumerate XML Indexes query optimizer mode to the database system requires modifying the query optimizer and index manager to support a *virtual* `//*` index instead of the actual `//*` index.

3.3.2 Multipath XML Indexes

A multipath index (for example, the XMLIndex index described in [73]) can be considered as a set of linear XPath expressions that are used to construct an XPath tree as described in Section 3.2.1. We can rely on the same Enumerate XML Indexes optimizer mode described for linear indexes to recommend multipath indexes. All the linear XML patterns enumerated for every query are included in one multipath index to construct a new expression tree. For example, using the

CG1	/Security⟨Yield⟩/SecInfo/*⟨Industry⟩
CG2	/Security⟨Yield⟩/SecInfo/*⟨Sector⟩

Table 3.2: Basic set of multipath index candidates for queries *Q2* and *Q3*.

basic set of linear indexes recommended by the optimizer in Table 3.1, we can now construct a tree that represents the multipath index we are recommending as `/Security⟨Symbol ∪ Yield⟩/SecInfo/*⟨Sector⟩`. More than one tree can be constructed from the candidate basic linear indexes based on other factors such as the table column that includes them. We illustrate candidate enumeration and generalization, which is described in the next section, for multipath indexes using two queries on the TPoX data: *Q2* presented in Section 3.2.2 and *Q3* which is as follows:

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
```

For *Q3*, we enumerate the linear patterns `/Security/Yield` and `/Security/SecInfo/*/Industry` as candidate patterns using the Enumerate XML Indexes mode developed for linear indexes (Section 3.3.1). These two linear patterns are then used to construct the multipath index tree pattern `/Security⟨Yield⟩/SecInfo/*⟨Industry⟩` (Figure 3.9). Similarly, the multipath index `/Security⟨Yield⟩/SecInfo/*⟨Sector⟩` (Figure 3.5) is enumerated for *Q2*. Table 3.2 summarizes these results.

3.4 Candidate Generalization

The optimizer helps us identify index patterns specific to each query. However, it is unable to identify index patterns that can benefit multiple queries in the current workload and also future queries with similar patterns. We assume that the

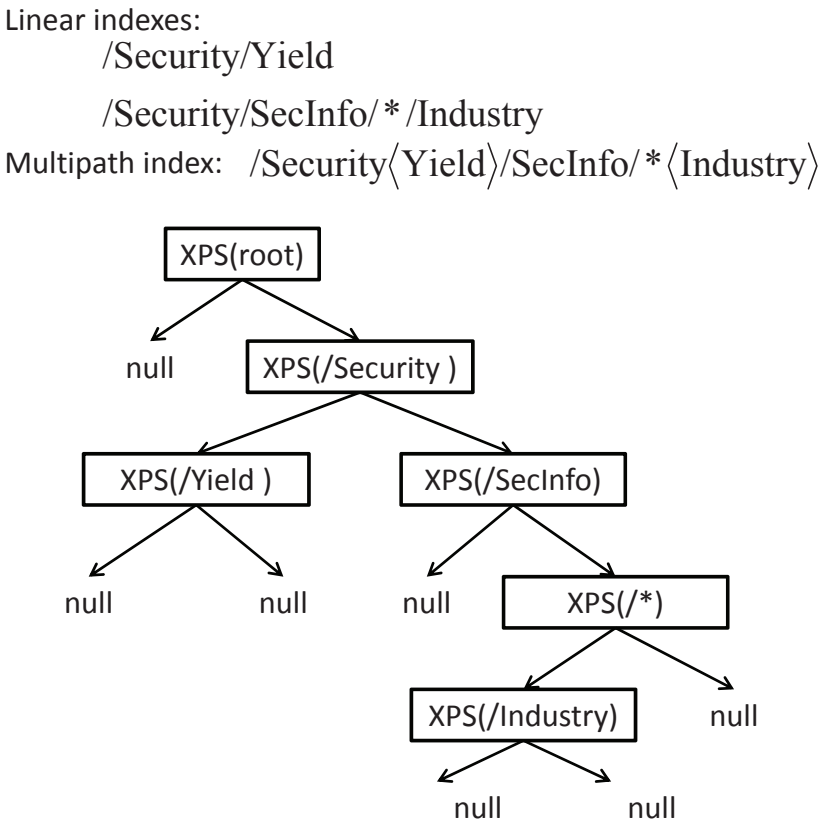


Figure 3.9: XPath XPS tree for the index pattern `/Security⟨Yield⟩/SecInfo /*⟨Industry⟩`.

queries that we have not seen in the input workload and would like to answer in the future have XPath expressions that are slight variants of the XPath expressions that appeared in the queries of the input workload. The XML patterns that can be part of queries in future workloads and that we would like to be able to answer are patterns that have at least one XML expression step shared with at least two XML patterns that appear in queries in the input workload. For example, if we have seen the XPath expression `/Customer/homePhone` in the input workload, we want to recommend an index that can also answer a query that has the expression `/Customer/cellPhone` even though this expression is not seen in the input workload. 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. In addition, we apply these rules on pairs of enumerated candidate XML index patterns to give us an intuition about what other similar expressions that exist in the data. Therefore, every XML index pattern generated by the generalization rules shares a common sub-path expression with at least a pair of XML index patterns from the enumerated candidates.

For example, the extensions we propose to the query optimizer helped us identify the following two XPath path expressions for queries *Q1* and *Q2* for indexing: `/Security/Symbol` and `/Security/SecInfo/*/Sector`. Based on these two path expressions, we expand the set of candidates to include the more general pattern `/Security//*`. This new path expression covers the two original path expressions as well as other path expressions that could potentially exist in the data, such as `/Security//Industry`. This more general candidate index is a new alternative that can be recommended by our Index Advisor instead of the two original candidate indexes. This new candidate index will generally have a size that is greater than or equal to the total size of the two original candidate indexes, since it potentially covers more elements in the data than they do. But this new general index has the advantage that it can answer more queries than the two original indexes and so it can potentially be useful for queries beyond the training workload.

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

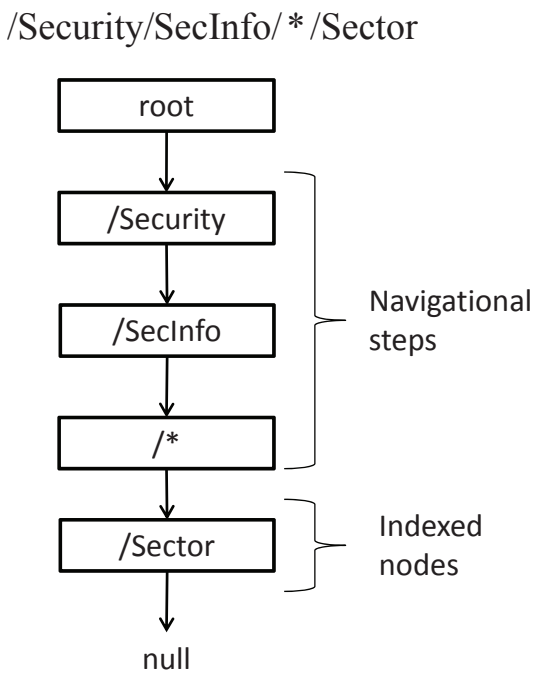


Figure 3.10: Linked list for index pattern .

In Section 3.4.1, we focus on index patterns that are expressed as *linear XPath path expressions* that contain a single element to be indexed throughout the XML documents. 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]`. In Section 3.4.2, we extend our approach to handle multipath indexes.

3.4.1 Linear XML Indexes

We represent path expression patterns as linked lists in which each node represents a path step. 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 navigational steps leading to this last step. For the index pattern `/Security/SecInfo/*/Sector`, the nodes `Security`, `SecInfo`, and `*` are navigational steps, while `Sector` represents the nodes being indexed by this pattern. Figure 3.10 shows the linked list that we use to represent this index pattern.

To generalize a pair of XML patterns, we start at the head nodes of the linked lists representing the path expressions and perform a synchronized traversal of the two lists. We examine each navigational step in the two patterns and check if they match. If a match is found, we add a matching step in the generated pattern. If an immediate match is not found we skip steps looking for a match and this is reflected in the generated pattern by adding $*$ steps. We continue this procedure until we reach the indexed nodes. Our process for generalizing pairs of index patterns is divided into two functions: *generalizeStep* and *advanceStep*. Each of these functions returns one or more linked lists representing generalized patterns. To generalize a pair of path expressions, we make an initial call $generalizeStep(null, p_i, p_j)$ (Algorithm 1), 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). In the algorithm, we refer to the generalized pattern currently being built as *genXPath*. The algorithm generalizes the nodes pointed to by p_i and p_j to *newNode*, and appends this new node to the *genXPath* path expression built up to this point. To perform this generalization, we check if p_i and p_j have the same name test. If so, the newly generated node retains the same name test as these nodes. If not, we replace the name test with a wildcard label, $*$. The navigation axis of *newNode* is determined by calling a function $genAxis(p_i.axis, p_j.axis)$, which returns *descendant axis* (`//`) if at least one of the inputs is a descendant axis, and returns *child axis* (`/`) otherwise. We also use a function $isLast(p)$ to test whether p points to the last step of a path expression (the target of the navigation). The function *generalizeStep* generalizes two steps if both are navigational steps or both are last steps, and hence we need another function to navigate through the expressions.

The other function, *advanceStep*, plays the role of traversing the expression lists by advancing the pointers p_i and p_j as outlined in Algorithm 2. The function *advanceStep* is designed to generate candidates that are as general as possible. We terminate the navigation of the two expressions once we finish generalizing their last steps (Line 1 in Algorithm 2). A last step node can only be generalized with another last step node, so the conditions listed in Lines 3 and 7 in Algorithm 2

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

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

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. A wildcard `*` step is inserted into the generalized expression to account for advancing the navigation pointer at one of the input expressions. The last case (Line 11) handles generalizing two navigational middle steps. In this case, we return the results of three generalizations: (1) advance the pointers of both expressions one step and generalize them, (2) and (3) try to find an occurrence of the first node of first (second) expression in the second (first) expression and generalize them together. In cases (2) and (3), no generalization is performed if the search fails. These two cases handle the reoccurrence of nodes in an expression, for example generalizing `/a/b/d` and `/a/d/b/d` will return `/a//d` and `/a//b/d`. We perform a final rewrite step before returning an XPath by replacing 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: `/Security/Symbol` and C2: `/Security/SecInfo/*/Sector` from Table 3.1, we follow these steps:

Algorithm 2 advanceStep(*genXPath*, p_i, p_j)

1: **if** *isLast*(p_i) **and** *isLast*(p_j) **then**
2: **return** {*genXPath*}
3: **else if** *isLast*(p_i) **and** !*isLast*(p_j) **then**
4: $p_{jL} \Leftarrow$ last step in p_j expression
5: *genXPath* $\Leftarrow$ append a node with *nameTest* `"*"` to *genXPath*
6: **return** *generalizeStep*(*genXPath*, p_i, p_{jL})
7: **else if** !*isLast*(p_i) **and** *isLast*(p_j) **then**
8: $p_{iL} \Leftarrow$ last step in p_i expression
9: *genXPath* $\Leftarrow$ append a node with *nameTest* `"*"` to *genXPath*
10: **return** *generalizeStep*(*genXPath*, p_{iL}, p_j)
11: **else**
12: $p_{in} \Leftarrow$ first match of *nameTest* of first node of p_j.*next* in p_i.*next* expression
13: $p_{jn} \Leftarrow$ first match of *nameTest* of first node of p_i.*next* in p_j.*next* expression
14: $genXPath_{new} \Leftarrow$ append a node with *nameTest* `"*"` to *genXPath*
15: **return** {*generalizeStep*(*genXPath*, p_i.*next*, p_j.*next*),
generalizeStep($genXPath_{new}$, p_{in}, p_j.*next*),
generalizeStep($genXPath_{new}$, p_i.*next*, p_{jn})}
16: **end if**

1. We 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.

2. To complete processing the two expressions, we call *advanceStep*(`/Security`, `/Security/Symbol`, `/Security/SecInfo/*/Sector`). In this call, the condition listed in Line 11 of *advanceStep* fires, and we have three possible generated XPath expressions.

   (a) 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.

   (b) We search for `/Symbol` in `/SecInfo/*/Sector`, but the search fails and no generalized path expressions is produced.

   (c) We search for `/SecInfo` in `/Symbol`, but the search fails and no generalized path expressions is produced.

3. After checking the previous alternatives, we only proceed with the call *advanceStep*(`/Security`, `/Symbol`, `/SecInfo/*/Sector`). In *advanceStep*, the condition listed in Line 3 is now triggered and the pointer of the second expression is advanced until its last step and a call *generalizeStep*(`/Security/*`, `/Symbol`, `/Sector`) is issued.

4. Finally, *advanceStep*(`/Security/*/*`, `/Symbol`, `/Sector`) is called from line 12 of Algorithm 1, the condition listed in Line 1 is triggered, a rewrite step is performed, and `/Security//*` is returned.

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

Table 3.3: Generalized candidates obtained from linear indexes in Table 3.1.

Based on these results, we can extend the basic candidates in Table 3.1 to include candidate C4 in Table 3.3. Candidate C3 cannot be generalized with either C1 or C2 because it is of a different data type.

Since the training workload might not allow us to have enough opportunities to expand the set of candidate indexes, we propose heuristic approaches 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. An example of a path expressions in the basic candidate set that might not be generalized with any other path expression is candidate C3 in Table 3.1. To get more general candidates even from these individual candidate paths that have no common sub-expressions with other candidates, we explore using 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 3.3. This approach could be extended to consult the XML data in the database to determine the usefulness of such a generalization and recommend other generalizations. With this extension, a `*` replacement would only be performed when there are other paths in the data with the same common leading path. In the current implementation, described in Section 3.7, we do not consult the data as described above.

3.4.2 Multipath XML Indexes

We revisit the generalization algorithm to make it capable of generalizing expression trees representing the index patterns for multipath XML indexes. To generalize a pair of expression trees we rely on the function *generalizeTreeStep* (Algorithm 3) and its helper functions outlined in Algorithms 4–9. The function *generalizeTreeStep* generalizes the first steps 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. *generalizeTreeStep* applies multiple conditions on p_i and p_j and calls the respective helper functions. The conditions checked by *generalizeTreeStep* are as follows:

1. When at least one of the expressions reaches its end (Line 1, Algorithm 3), we terminate the navigation of the two expressions and return the expression trees generated to this point.

2. When generalizing two steps with no index children (Line 3, Algorithm 3), we call the helper function *generalizeTree–NoIndexedChildren* outlined in Algorithm 4. Recall that a navigational step that does not have an index child is represented in the expression tree as a node with no left child. In this case we try to find an occurrence of the first 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 expression trees that are under construction, and the new trees are returned to the calling function.

3. When we generalize two steps where only one of them has an index child (Lines 5 and 7, Algorithm 3), we call the helper function *generalizeTree–OneIndexedChild* outlined in Algorithm 5. In this case, we try to advance the pointer of the step with no index child to a step with an index child 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 index child.

4. When the current two steps have index children (Line 9, Algorithm 3), we call the helper function *generalizeTree–IndexedChildren* outlined in Algorithm 6. In this case, an attempt to generalize each one of these steps with a step from the other expression that has a similar index child is made. If these attempts fail, the two expression steps are generalized together and a new index child with a union of the original two index children is created.

Algorithm 3 generalizeTreeStep(*genXPathTrees*, p_i, p_j)

```
1: if p_i==null or p_j==null then
2:     return genXPathTrees
3: else if !hasIndexChild(p_i) and !hasIndexChild(p_j) then
4:     return generalizeTree–NoIndexedChildren(genXPathTrees, p_i, p_j)
5: else if hasIndexChild(p_i) and !hasIndexChild(p_j) then
6:     return generalizeTree–OneIndexedChild(genXPathTrees, p_i, p_j)
7: else if !hasIndexChild(p_i) and hasIndexChild(p_j) then
8:     return generalizeTree–OneIndexedChild(genXPathTrees, p_j, p_i)
9: else if hasIndexChild(p_i) and hasIndexChild(p_j) then
10:    return generalizeTree–IndexedChildren(genXPathTrees, p_i, p_j)
11: end if
```

In order to accomplish the task of *generalizeTreeStep*, some helper functions are used. *hasIndexChild* is a boolean function that returns true if the current step of the expression has a left child, which is also an element to be included in the index. For example, *hasIndexChild*(`/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 *appendIndexChild*, which navigates to the last step of an expression and then appends an index child to it. For example, *appendIndexChild*(`/Security`, `/Yield`) returns `/Security`⟨`Yield`⟩. *generalizeNode*, which is described in Algorithm 7, takes two nodes and generalizes them to *newNode*, and returns *newNode*. To perform this generalization, we check if the two nodes have the same name test.

Algorithm 4 generalizeTree–NoIndexedChildren(*genXPathTrees*, p_i, p_j)

1: $p_{i_{next}} \Leftarrow p_i$
2: $p_{j_{next}} \Leftarrow p_j$
3: **if** $p_i.nameTest == p_j.nameTest$ **then**
4: *genTrees* $\Leftarrow$ *generalizeStepNoIndexChild*(*genXPathTrees*, p_i, p_j, null)
5: **else**
6: $p_{in} \Leftarrow$ first occurrence of first node of p_j in p_i
7: $p_{jn} \Leftarrow$ first occurrence of first node of p_i in p_j
8: **if** $p_{in} \neq$ null **then**
9: *genTree1* $\Leftarrow$ *generalizeStepNoIndexChild*(*genXPathTrees*, p_{in}, p_j, "*")
10: $p_{i_{next}} \Leftarrow p_{in}$
11: **end if**
12: **if** $p_{jn} \neq$ null **then**
13: *genTree2* $\Leftarrow$ *generalizeStepNoIndexChild*(*genXPathTrees*, p_i, p_{jn}, "*")
14: $p_{j_{next}} \Leftarrow p_{jn}$
15: **end if**
16: **if** $p_{in} ==$ null **and** $p_{jn} ==$ null **then**
17: *genTree3* $\Leftarrow$ *generalizeStepNoIndexChild*(*genXPathTrees*, p_i, p_j, null)
18: **end if**
19: *genTrees* $\Leftarrow$ *genTree1* $\cup$ *genTree2* $\cup$ *genTree3*
20: **end if**
21: **return** *generalizeTreeStep*(*genTrees*, $p_{i_{next}}.next$, $p_{j_{next}}.next$)

Algorithm 5 generalizeTree–OneIndexedChild(*genXPathTrees*, p_i, p_j)

1: $p_{j_{next}} \Leftarrow p_j$
2: $p_{jn} \Leftarrow$ first occurrence of first node of p_i in p_j
3: $p_{jpn} \Leftarrow$ first occurrence of first node of p_i in p_j where $p_{jpn}.indexChild == p_i.indexChild$
4: **if** $p_{jpn} \neq$ null **then**
5: *genTree1* $\Leftarrow$ *generalizeStepWithIndexChild*(*genXPathTrees*, p_i, p_jpn, "*")
6: $p_{j_{next}} \Leftarrow p_{jpn}$
7: **end if**
8: **if** $p_{jn} \neq$ null **and** $p_{jpn} ==$ null **then**
9: *genTree2* $\Leftarrow$ *generalizeStepNoIndexChild*(*genXPathTrees*, p_i, p_{jn}, "*")
10: $p_{j_{next}} \Leftarrow p_{jn}$
11: **end if**
12: **if** $p_{jn} ==$ null **and** $p_{jpn} ==$ null **then**
13: *genTree3* $\Leftarrow$ *generalizeStepNoIndexChild*(*genXPathTrees*, p_i, p_j, null)
14: **end if**
15: *genTrees* $\Leftarrow$ *genTree1* $\cup$ *genTree2* $\cup$ *genTree3*
16: **return** *generalizeTreeStep*(*genTrees*, $p_i.next$, $p_{j_{next}}.next$)

Algorithm 6 generalizeTree–IndexedChildren(*genXPathTrees*, p_i, p_j)

1: $p_{i_{next}} \Leftarrow p_i$
2: $p_{j_{next}} \Leftarrow p_j$
3: **if** $p_i.nameTest == p_j.nameTest$ **then**
4: $genTrees \Leftarrow generalizeStepWithIndexChild(genXPathTrees, p_i, p_j, \text{null})$
5: **else**
6: $p_{ipn} \Leftarrow$ first occurrence of first node of p_j in p_i where $p_{ipn}.indexChild == p_j.indexChild$
7: $p_{jpn} \Leftarrow$ first occurrence of first node of p_i in p_j where $p_{jpn}.indexChild == p_i.indexChild$
8: **if** $p_{ipn} \neq$ null **then**
9: $genTree1 \Leftarrow generalizeStepWithIndexChild(genXPathTrees, p_{ipn}, p_j, \texttt{"*"})$
10: $p_{i_{next}} \Leftarrow p_{ipn}$
11: **end if**
12: **if** $p_{jpn} \neq$ null **then**
13: $genTree2 \Leftarrow generalizeStepWithIndexChild(genXPathTrees, p_i, p_{jpn}, \texttt{"*"})$
14: $p_{j_{next}} \Leftarrow p_{jpn}$
15: **end if**
16: **if** $p_{ipn} ==$ null **and** $p_{jpn} ==$ null **then**
17: $genTree3 \Leftarrow generalizeStepWithIndexChild(genXPathTrees, p_i, p_j, \text{null})$
18: **end if**
19: $genTrees \Leftarrow genTree1 \cup genTree2 \cup genTree3$
20: **end if**
21: **return** $generalizeTreeStep(genTrees, p_{i_{next}}.next, p_{j_{next}}.next)$

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 *generalizeStepNoIndexChild*, outlined in Algorithm 8, to generalize two nodes and add their generalization to all the general expressions being constructed as a next step. A similar function is *generalizeStepWithIndexChild* (Algorithm 9), which generalizes two nodes and their index children and then appends the new generalized step and index children to all the general expressions being constructed.

Algorithm 7 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
```

For example, we generalize candidates CG1 (/Security⟨Yield⟩/SecInfo /*⟨Industry⟩) and CG2 (/Security⟨Yield⟩/SecInfo/*⟨Sector⟩) from Table 3.2 by applying our generalization rules as described in the following steps:

1. Initially, we make a call *generalizeTreeStep*(*null*, /Security⟨Yield⟩/SecInfo/*⟨Industry⟩, /Security⟨Yield⟩/SecInfo/*⟨Sector⟩). Since the first nodes of both expressions have index children, Condition on Line 9 of Algorithm 3 is triggered, and function *generalizeTree–IndexedChildren* (Algorithm 6) is called. Both expressions have Security as their first node, and there-

fore *generalizeStepIndexChild*(*null*, `/Security⟨Yield⟩`, `/Security⟨Yield⟩`) is called and a new expression `/Security⟨Yield⟩` is returned.

2. 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 index children, so the condition on Line 3 of Algorithm 3 is triggered, and *generalizeTree–NoIndexedChildren* is called. The two expressions have first steps with the same *nameTest* therefore a new node `/SecInfo` is appended to the current general expression.

3. Next, *generalizeTreeStep*(`/Security⟨Yield⟩/SecInfo`, `/*⟨Industry⟩`, `/*⟨Sector⟩`) is called. The condition on Line 9 of Algorithm 3 is triggered as the two expressions have index children. But the two index children `Industry` and `Sector` are not equal. Hence, a new index child node with a union of `Industry` and `Sector` is created. Finally a new expression of `/Security⟨Yield⟩/SecInfo/*⟨Industry ∪ Sector⟩` is returned.

4. The generalization process is terminated when we encounter null steps (Line 1, Algorithm 3). Figure 3.11 shows the expression tree representation of the generated index `/Security⟨Yield⟩/SecInfo /*⟨Industry ∪ Sector⟩`.

3.5 Estimating the Benefit of XML Indexes

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

Algorithm 8 generalizeStepNoIndexChild(*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*

on maintenance statements because the index has to be modified along with the data, and it is important to take this cost into account when estimating benefit. Techniques described in this section work for both linear and multipath indexes.

Relational index advisors leverage the query optimizer to estimate the benefit to a query workload of having a particular index configuration [23, 87]. To do the same in our XML Index Advisor, we employ a new query optimizer mode that we call the Evaluate XML Indexes mode. This mode relies on creating virtual indexes and estimating the cost of workload queries with these virtual indexes in place. To estimate the cost of queries when using these virtual indexes, we need to collect statistics on the XML data populated in the database (e.g., RUNSTATS command in DB2). The optimizer in Evaluate XML Indexes mode uses these data statistics to estimate for the virtual indexes the index statistics that are necessary for the optimizer cost model (e.g., the number of leaf nodes in a B-tree). The details of the index statistics that are needed depend on the implementation of the query optimizer. The optimizer can then include the virtual indexes with other existing real indexes when performing index matching to find the possible indexes to be

Algorithm 9 generalizeStepWithIndexChild(*genXPathTree*, p_i, p_j, *preNode*)

1: $genXPathTreeNew \Leftarrow \{\}$
2: $newNode \Leftarrow generalizeNode(p_i\ ,\ p_j)$
3: $newIndexChild \Leftarrow generalizeNode(p_i.indexChild\ ,\ p_j.indexChild)$
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** $newIndexChild \neq null$ **then**
12: $t_{new} \Leftarrow appendIndexChild(t_{new},\ newIndexChild)$
13: **else**
14: $t_{new} \Leftarrow appendUnionIndexChild(t_{new},\ p_i.indexChild\ ,\ p_j.indexChild)$
15: **end if**
16: $genXPathTreeNew \Leftarrow genXPathTreeNew \cup t_{new}$
17: **end for**
18: **return** *genXPathTreeNew*

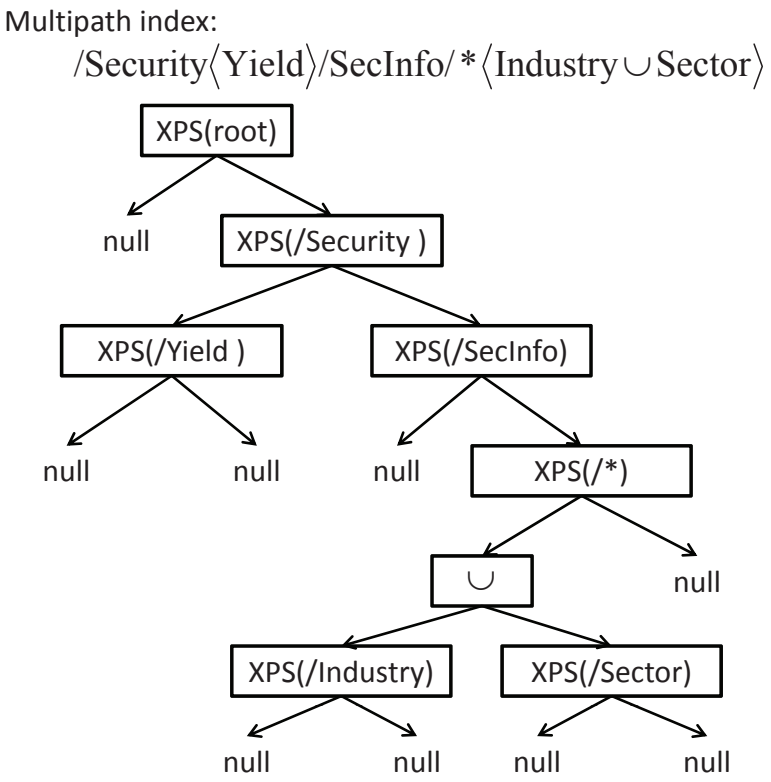


Figure 3.11: XPath XPS tree for the index pattern `/Security⟨Yield⟩/SecInfo /*⟨Industry ∪ Sector⟩`.

used in a query, and when determining a query execution plan for this query. After optimizing a query in Evaluate XML Indexes mode, the optimizer returns the set of indexes that were used, plus their index statistics and the new cost information of the evaluated query. This information is used by our index advisor to determine the benefit of using an index or a configuration consisting of multiple indexes.

The XML Index Advisor architecture allows us to rely completely on the query optimizer for cost estimation by using the Evaluate XML Indexes mode. This has the advantage of leveraging the optimizer's tuned, well-developed cost model. The XML Index Advisor does not attempt to influence or improve the query optimizer cost model and instead uses the cost model as is. XML cost modeling is an active area of research in its own right and is beyond the scope of this thesis. We describe the details of the Evaluate Index mode that we have implemented in the DB2 query optimizer in Section 3.7. Other database systems may use other XML statistics

and cost models and therefore require different implementation details, but the fundamental idea of the Evaluate XML Indexes mode is the same for all cost-based query optimizers. In the rest of this section, we describe how we use the information returned by the optimizer in our Index Advisor application.

3.5.1 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 XML 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 $C_{old}(q)$ (i.e. the cost of the query when any existing configuration is in place) and its cost after creating the index $C_{new}(q)$ (i.e. the cost of the query when the index is added to any existing configuration). Thus, the benefit of index x to query q is $\mathit{Benefit}(x;q) = C_{old}(q) - C_{new}(q)$. We use the Evaluate XML Indexes mode to evaluate the cost of a query when an index is in place without actually creating the index.

To evaluate the benefit of an index for a workload of queries, we generalize the above calculation to: $\mathit{Benefit}(x;W) = \sum_{q \in W}(C_{old(q)} - C_{new}(q))$. 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 XML Indexes mode to estimate their new costs. Thus, we have: $\mathit{Benefit}(x_1, x_2, \ldots, x_n; W) = \sum_{q \in W}(C_{old}(q) - C_{new}(q))$.

3.5.2 Estimating the Benefit of Update, Delete, and Insert Statements

Our workloads may contain update, delete, and insert (UDI) statements in addition to queries. Any index that we recommend must be maintained for each of the UDI statements in the workload. At the same time, update and delete statements may

benefit from an index that helps them identify the data that needs to be updated or deleted. 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. This is because updating the indexes is an operation that has to be performed regardless of the chosen query execution plan, so ignoring the cost of this operation will not affect the chosen query execution plan. 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 due to a UDI statement, we use the data statistics to estimate the number of XML documents that have changed 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 XML elements that the statement will affect in the index as follows:

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

In this equation we are making two simplifying assumption. We are assuming that the number of indexed XML elements from all 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. We assume that a B-tree is used as the index implementation. Therefore, for every element updated in the index, a *CPUCostPerNode* cost and an *IOCostPerNode* cost multiplied by the number of navigated levels of the B-tree are incurred. In many (or most) cases, some of the navigated B-tree nodes will be in the buffer pool, so no I/O is incurred to traverse them. The value of the calibration constant *IOCostPerNode* is chosen in a way that takes this

buffering into account. Thus, the total maintenance cost of an index x_i because of a statement s, is calculated as:

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

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 that contains queries $q_1, q_2, \ldots, q_l$ and maintenance statements $s_1, s_2, \ldots, s_k$:

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

3.5.3 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. This approach is inspired by the atomic configuration concept described in [23].

While we are generating the set of candidate indexes (basic and generalized), for each index, x, we keep track 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 3.1, 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}.

3.6 Searching for the Optimal Configuration

3.6.1 Search Problem

The XML Index Advisor needs to search the space of possible index configurations consisting of indexes from this candidate set to find the index configuration with the maximum benefit, subject to a constraint specified by the user on the disk space available for the configuration. This combinatorial search problem can be modeled as a 0/1 knapsack problem [87], 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*. The search problem goal is described as:

$$maximize_{p \in P}\{Benefit(W, p)\}$$

such that

$$\sum_{x \in p} Size(x) \leq DiskBudget$$

In the previous equation, p is an index configuration in the set of candidate configurations P, W is the workload, and x is an index in p.

Modeling the index search as a 0/1 knapsack problem gives us a spectrum of solutions that ranges from greedy approximation to dynamic programming. When considering the right algorithm for the search problem, we also need to take into account 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 also propose a *top down* search that chooses as many general indexes as it can fit into the disk budget. The goals of the greedy search with heuristics and the top down search are fundamentally different: The greedy search with heuristics attempts to find the best possible set of indexes for the given workload, without any consideration for the generality of these indexes, while the top down search attempts to find configurations that are as general as possible so that they can benefit not only the given workload but also any similar future workloads.

In our search algorithms, we model the relationship between queries in the workload, the extracted XML patterns, and the generalized candidates as *directed acyclic graph (DAG)*. Figure 3.12 presents an example of such a DAG. For queries $q_1, q_2, \ldots q_n$ we enumerate a basic set of candidates $p_1, p_2, \ldots p_m$ as described in Section 3.3. Each query and basic candidate is represented as a node in the DAG.

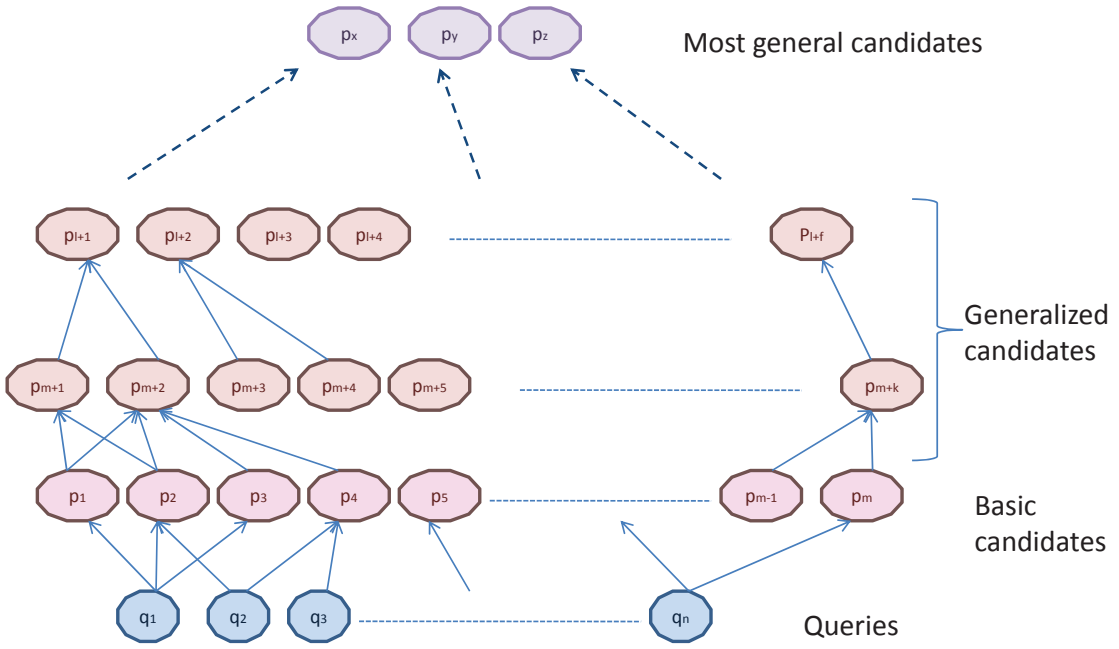


Figure 3.12: Relationship between workload queries and candidate XML patterns

One basic candidate can be enumerated because of more than one query, and one query can produce more than one candidate, so we associate with each candidate the set of queries that produced it via a set of edges in the DAG. We build the next levels in the DAG by generalizing the basic candidates using the Algorithm in Section 3.4.1, and we continue until we reach the most general candidates as shown in the figure. For each new candidate created during candidate generalization, we associate with it the set of XML patterns that were the cause of generating it through a set of edges in the DAG. Hence, by following these edges, we will have for any candidate index pattern a list of all candidates in the subtree rooted at this pattern, which we call the *coverage list*. The leaves of the subtree are the queries that can benefit from this candidate index pattern. Along with the coverage list, we keep a list of affected queries (described in Section 3.5.3). The affected list of a generalized pattern is the concatenation of the affected lists of its children. For example, in Figure 3.12, the XML pattern p_{l+1} has the coverage list $\{p_{m+1}, p_{m+2}, p_1, p_2, p_3, p_4\}$ and the list of affected queries $\{q_1, q_2, q_3\}$. Next, we present our two search algorithms.

3.6.2 Greedy Search with Heuristics

The greedy approximation of the 0/1 knapsack problem was not effective for our XML Index Advisor. The benefit of an index is highly dependent on the existence of other indexes in the configuration that answer the same query. Moreover, the greedy search can select general indexes that can be used for path expressions already covered by other indexes in the configuration. However, the 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 at the end of the index selection process that we never use again for adding more indexes, even though this space could be very useful. A similar approach is used for searching relational indexes in [87]. After selecting a configuration of indexes and eliminating redundant indexes from it, an iterative phase of random swapping of indexes is performed to try other variations of the configuration. In our proposed solution for searching candidate XML indexes, we take another deterministic approach.

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

This 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 replicate others already chosen. When a general index, $x_{general}$, is added to the recommended index configuration, it must be "better" than the indexes it generalizes, $x_1, x_2, \ldots, x_n$. We define $IB(X)$, the *improved benefit* of the set of indexes X, as the benefit of the recommended index configuration built to this point when X is

Algorithm 10 heuristicSearch(*candidates*)

```
1: sort candidates according to their benefit/size ratio
2: recommended ← φ, currSize ← 0, currCoverage ← φ
3: while currSize < diskConstraint do
4:    best ← pick the next best cand in candidates
5:    if currCoverage = φ or currCoverage ∩ best.coverage = φ then
6:       if currSize + best.size ≤ diskConstraint then
7:          recommended ← recommended ∪ best
8:          currCoverage ← currCoverage ∪ best.coverage
9:          currSize ← currSize + best.size
10:      end if
11:   else
12:      overlapConfig ← {cand ∈ recommended, cand.coverage ∩ best.coverage}
13:      configWithBest ← recommended ∪ best − overlapConfig
14:      configPredicted ← recommended after adding predicted candidates to it
15:      evaluate the benefit of configWithBest and configPredicted
16:      select the best configuration between configWithBest and configPredicted
         according to the benefit and size heuristic rules
17:      update recommended
18:   end if
19: end while
20: return recommended
```

added to it. A general index is added to the configuration only if the following two heuristic conditions are satisfied:

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

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.

Algorithm 10 illustrates the greedy search algorithm with the added heuristic rules. We start by sorting the list of candidate indexes (*candidates*) according to their *benefit/size* ratio and creating an empty configuration (*recommended*) with a zero size (*currSize*) and empty coverage (*currCoverage*). Next, we iterate through the sorted list of *candidates* by selecting the index with the highest *benefit/size* ratio (*best*), while the configuration size (*currSize*) has not reached the given budget (*diskConstraint*). Every iteration, we have two cases: (1) the new index is useful to new queries in the workload (Lines 6 - 10) and (2) the new index has more specific or more general forms of it that are already chosen in the *recommended* configuration (Lines 12 - 17). In the first case, when the index is adding a new query coverage, we add the index to the *recommended* configuration if the disk constraints allow it. In the second case, when the coverage of the index overlaps with the coverage with selected indexes, we apply our proposed heuristic rules (Line 16). In this case, we create two configurations: (1) *configWithBest*: the most updated *recommended* configuration after adding *best* index to it and removing any children indexes of *best* from it and (2) *configPredicted*: the *recommended* configuration selected in previous iterations of the algorithm after adding children indexes of *best* that we have not yet seen from *candidates* to it. We evaluate the benefit of these two configurations

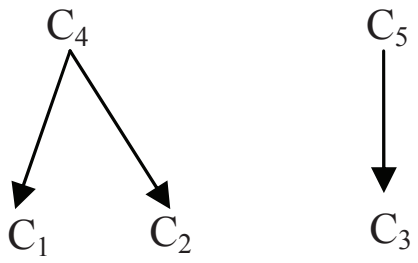


Figure 3.13: Directed acyclic graph of the candidates.

and compare them according to the heuristic rules we described above. Finally, according to the result of applying our heuristics, we decided whether to add the *best* index to the *recommended* configuration or not.

3.6.3 Top Down Search

The greedy search with heuristics attempts to recommend a configuration with the highest benefit 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, this might occur if the workload is all the queries in a particular application. However, another likely scenario is that the DBA has assembled a representative training workload, but the actual workload may be a variation on this training workload. This is true for relational data, but it is of added importance for XML, because the rich structure of XML allows users to pose queries that retrieve different paths of the data with slight variations. If this is the case, and the workload presented to the Index Advisor is a representative of a larger class of possible workloads, then we posit that the goal of the Index Advisor should be to choose a set of indexes that is as general as possible, while still benefiting the workload queries. We have developed a *top down search* algorithm to achieve this goal.

In the top down search, we construct a DAG, similar to the one shown in Figure 3.12, 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 3.13 illustrates the DAG constructed for the expanded set of candidates for our running example. We start with these roots of the DAG as our current configuration. Since general indexes are typically large in size, this starting configuration is likely to exceed the available disk space budget, but it likely has a higher benefit compared to specific indexes. General indexes can have zero or negative benefit for two reasons: (1) high maintenance cost because of update, delete, and insert statements in the workload, and (2) not being used in optimizer plans. To handle this, we add a preprocessing phase to remove any indexes with zero or negative benefit from our search space. Next, we iteratively replace a general index from the current configuration with its specific (and smaller) child indexes, and we repeat this step until the configuration that we have fits within the disk space budget. The top down search pseudocode is illustrated in Algorithm 11.

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:

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

Since our goal is to obtain the maximum total benefit for the workload with the most general configuration that fits in the disk space budget, we iteratively choose the general index with the smallest $\Delta B / \Delta C$ ratio, and we replace it with its (more specific) children in the DAG. That is, we replace general indexes whose additional

Algorithm 11 topDownSearch(*topCandidates*)

```
1: recommended ← topCandidates
2: currSize ← recommended.size
3: while currSize > diskConstraint do
4:    for all cand ∈ recommended do
5:       calculate ΔB/ΔC of cand
6:    end for
7:    recommended ←configuration after replacing the candidate with minimum
      (ΔB/ΔC) with its children
8:    currSize ← recommended.size
9: end while
10: return recommended
```

benefit per unit cost over their children is lowest. In case of ties, we select the index with the largest ΔC. If we run out of general candidates to replace and do not yet meet the disk space budget, we use greedy search. Note that in this case we do not need to apply our heuristics since none of the indexes we are searching is general.

3.7 Implementation in DB2

To implement our XML index recommendation techniques we need to extend the query optimizer of the database system with the new query optimizer modes described in Sections 3.3 and 3.5. In addition, the main functionality of the XML Index Advisor is implemented as a client-side application on top of the database system. We have implemented the server-side extensions required for our index advisor in a prototype version of IBM® DB2® 9.5 for Linux, Unix, and Windows (henceforth referred to simply as DB2). DB2 supports both relational and XML data. It 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 [71]. The indexes can be used to answer structural or value queries on the data. We have implemented and experimentally verified our recommen-

dation techniques for linear XML indexes. However, we have not implemented recommending multipath indexes in our current prototype.

We have extended the DB2 query optimizer with *Enumerate XML Indexes* and *Evaluate XML Indexes* modes. These are implemented as EXPLAIN modes in the DB2 optimizer. Our EXPLAIN modes have been integrated into the main line DB2 code base and have been available to all users since DB2 version 9.7. The client side XML Index Advisor is implemented in Java™ 1.6, and communicates with the prototype server via JDBC (DB2 JCC package). This application is available for download from the IBM alphaWorks web site [33]. We have used this implementation to verify the efficiency of our Index Advisor and the high quality of the index configurations that it recommends. A demo of our XML Index Advisor for DB2 was presented in [31]. Next, we describe more details of our implementation. The details for a database system other than DB2 would be different, but the fundamental ideas would remain the same.

3.7.1 Implementation of the Enumerate XML Indexes Mode

DB2 allows only XML indexes that are represented by index patterns expressed as *linear XPath path expressions* that do not include predicates. Hence, we focus on linear indexes in our implementation. We have extended the DB2 query optimizer with a new Enumerate XML Indexes EXPLAIN mode. In this mode, the optimizer creates *virtual universal indexes* (or `//*` indexes, described in Section 3.3) and returns the XPath patterns in the queries that were matched with these universal indexes. These XPath patterns are the basic candidate patterns for the XML indexes recommended by the XML Index Advisor. The virtual universal indexes are not physically created on disk, but are only created in the metadata and in the optimizer's internal data structures. To conform with the XPath standard, DB2 uses distinct indexes for distinct data types, and distinct indexes for elements and attributes [14, 15]. Thus, in our implementation of the Enumerate XML Indexes mode in DB2, we create several virtual universal 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 XML Indexes mode to recommend candidate XPath index patterns.

3.7.2 Implementation of the Evaluate XML Indexes Mode

The Evaluate XML Indexes EXPLAIN mode relies on the query optimizer for cost estimation. A detailed description of the cost model of the DB2 optimizer, which we use in our prototype, can be found in [11]. The DB2 query optimizer needs statistics about virtual indexes to estimate the execution time of queries using these indexes. Some of these statistics are *data statistics*, such as the number of 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 [11].

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 [11]. 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 [11], to estimate the XML column cardinality, a count of all linear rooted paths occurring in the documents of that column is collected. But because the number of occurring rooted paths can be very large, this count is only kept for the most frequently occurring paths. To estimate the cardinality of an XML pattern, we check the most frequently 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 all components needed: the size of the index key and the number of keys. While

the number of keys for an index is based on the data to be indexed and 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.

3.7.3 Search Algorithms

In the client-side XML Index Advisor application, we have implemented the two search algorithms described in Sections 3.6.2 and 3.6.3, that is, greedy search with heuristics and top down search, respectively. In addition, we have implemented a naive greedy search to show the benefit of adding our proposed heuristics. These three search algorithms find approximations to the optimal solution. To find a near optimal solution, we use a dynamic programming algorithm that searches the exponential space of possible configurations. However, the dynamic programming algorithm does not consider the possible index interactions and hence it sometimes misses the true optimal solution.

We implemented the dynamic programming algorithm given in [47], which has a time and space complexity of $O(min\{2^n, n\sum_{1\leq i\leq n} p_i, nm\})$, where p_i is the size of index i, n is the number of indexes, and m is the disk size allocated for building indexes. In this solution, optimal sub-configurations are kept in a table, and in every iteration, an index is considered to be added to all of these optimal sub-configurations. This is equivalent to the search algorithm starting with a small disk budget and increasing it in every stage until the available disk budget is exhausted. 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 overhead to the solution. This solution suffers from pruning some configurations 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 out to be

the best ones later on. Another problem with dynamic programming is that it focuses on the absolute reduction in execution time, not the reduction relative to the original execution time of a query. It therefore favors indexes that benefit longer running queries, even if the benefit is small in relative terms.

We have 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*. The main purpose of this version is to overcome the overhead of calculating the benefit of every index configuration we are examining during the search. In the second version, we evaluate the benefit of every configuration as described in Section 3.5.1. We also use the efficient configuration evaluation technique described in Section 3.5.3. We refer to this version of the search algorithm as *top down full*. In both versions of the top down search algorithm, if the total budget is less than the size of the basic indexes (we searched through the DAG till we reached the leaf nodes), we perform a greedy search on the specific indexes.

3.7.4 Implementation Requirements for Other Database Systems

Although this thesis uses DB2 for evaluating our index recommendation techniques, we believe that these techniques can be used for most database systems that support XML. In this section, we discuss the requirements for implementing our index recommendation techniques in a system other than DB2. In our proposed architecture of the XML Index Advisor, we divided our implementation into two main parts: a client-side application and optimizer extensions (Figure 3.8). The client-side application encapsulates all the communication details with the database system in one database communication package. For every database system, we only need to implement on the client side a new database connection package with the specific details of this database system. In addition, we need to extend the database query optimizer with two new optimizer modes to allow us to enumerate candidate

indexes and evaluate cost of queries when candidate indexes are existing in the database.

The database system that we can extend needs to have the following characteristics:

1. The database system supports building partial XML indexes.

2. An XML query optimizer that performs cost based query optimization for XQuery queries and/or SQL/XML queries.

3. The ability to add special optimization modes to the query optimizer. Most database systems allow such special optimizer modes, and we need to add two new optimizer modes as described in Sections 3.7.1 and 3.7.2.

To implement an Enumerate XML Indexes optimizer mode, the query optimizer needs to have the following features:

1. XML index matching is performed by the optimizer prior to plan enumeration.

2. It is possible to create a universal index that would match all the XPath path expressions in a query that can be answered by an index. However, a rudimentary Enumerate XML Indexes optimizer mode can be implemented using an XML query parser that can extract the XPath path expressions referenced in the query. In this case, some of the enumerated candidate indexes will have zero benefit because the optimizer cannot use them. However, these zero benefit candidates will be eliminated while searching for the best index configuration.

To implement the Evaluate XML Indexes optimizer mode, the optimizer needs to have the following features:

1. It is possible to collect data statistics prior to query optimization (e.g., using RUNSTATS).

2. The index statistics required for XML cost estimation can be derived or estimated from the data statistics.

3. The optimizer can use virtual indexes in the index matching and plan enumeration phases of query optimization.

4. The optimizer can estimate the cost of a query that uses virtual indexes.

We believe that all of these requirements are easily supported by most current database systems that handle XML, especially database systems that have sophisticated relational design advisors and advanced XML indexing capabilities.

3.8 Experimental Evaluation

3.8.1 Experimental Setup

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. As mentioned in Section 3.7, we use DB2 for our experimental evaluation.

We used two XML benchmarks for our experiments: the TPoX [70] and XMark [78] benchmarks. 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 15 XQuery queries for XMark. To illustrate the effectiveness of our generalization algorithm, we also use synthetic queries on the TPoX data in Section 3.8.3.

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. For some experiments, we report the geometric mean of the speedup achieved by the queries in the workload. The geometric mean of the speedup for a workload is defined as: $(\prod_{i=1}^{n} s_i)^{1/n}$ where n is the number of queries in the workload and s_i is the speedup of query i when the index configuration recommended by the XML Index Advisor is created in the

database. We use the geometric mean of the query speedups because the variance in execution time for queries in both the TPoX and XMark workloads is very large. Simply measuring the total speedup for all queries or computing the arithmetic mean will make long-running queries dominate the speedup metric, and we do not want that to happen. Using the geometric mean leads to all queries contributing equally to the metric regardless of their execution time. For this reason, benchmarks such as TPC-D [85] also use the geometric mean to measure performance. In addition, we also report the *actual speedup* for some workloads: The measured execution time of the workload with no XML indexes divided by the measured 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 index configurations that are useful beyond the training workload. Furthermore, we demonstrate the accuracy of the statistics we create for cost estimation in the Evaluate XML Indexes mode, and of our estimation of the cost of updating indexes.

3.8.2 Effectiveness of the Advisor Recommendations

We have implemented five different combinatorial search strategies in our Index Advisor: (1) greedy search (without heuristics), (2) greedy search with the heuristics, (3) top down lite, (4) top down full, and (5) dynamic programming. In our first experiment, we compare the index recommendations of these five strategies.

Figures 3.14 and 3.15 show the estimated speedup for the search strategies with varying disk space budgets for the TPoX and XMark benchmarks, respectively. For each benchmark, we create an index configuration in which we have XML indexes for every indexable XPath expression in the query workload (i.e., an index for every basic indexing candidate enumerated by the Enumerate XML Indexes optimizer mode). We call this the *All Index* configuration. This configuration is overfitted to

the workload, but it is also the configuration with the highest possible speedup for a workload that consists of queries with no updates. The size of this configuration is 96.4MB for TPoX and 132.6MB for XMark. The estimated speedup achieved by this configuration is 48.9 for TPoX and 7.6 for XMark. In these figures, we use the benchmark queries (11 for TPoX and 15 for XMark) for recommending the indexes and also for evaluating the recommendations.

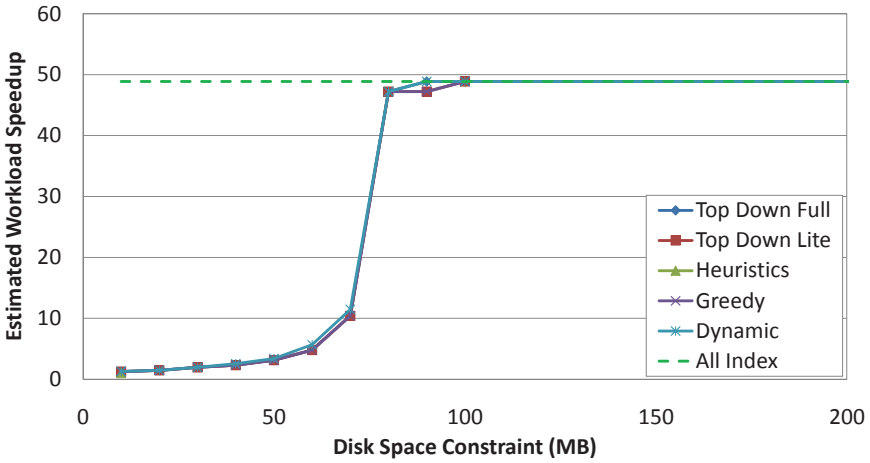


Figure 3.14: Estimated workload speedup (TPoX).

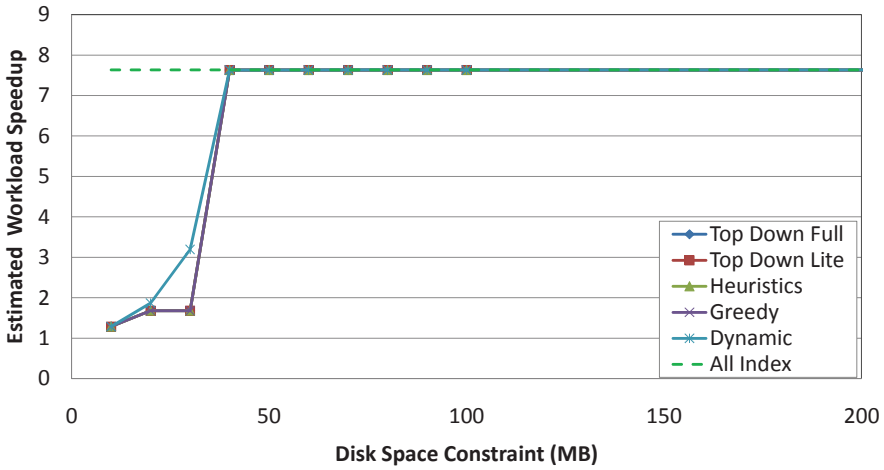


Figure 3.15: Estimated workload speedup (XMark).

Figures 3.14 and 3.15 show that our XML Index Advisor is able to recommend indexes that speed up workload execution for both workloads and all disk space

budgets. As expected, speedup increases as we increase the available disk space budget, until it reaches the best possible speedup of the *All Index* configuration. Dynamic programming recommends configurations that has an overall workload benefit that is higher than configurations recommended by other search strategies, but as we see later this comes at a cost since dynamic programming is the slowest search strategy.

One observation about the figures is that greedy search, greedy search with heuristics, and top down search all perform the same. Greedy search recommends the same indexes with and without heuristics because for the TPoX and XMark workloads, all the basic candidate indexes enumerated by the advisor have benefit/cost ratios that are less than any of the generalized indexes that are generated during the candidate generalization phase. This is because the generalized indexes generated for these workloads were all quite large in size. Both greedy search and greedy search with heuristics sort the candidate indexes and examine them in ascending order according to their benefit/cost ratio. In this case, all the basic candidate indexes (which are not generated by the generalization algorithm) will be selected before any general index is examined by the greedy and greedy with heuristics search algorithms. When the disk space constraint input to the advisor is smaller than the size of the *All Index* configuration, the disk budget is exhausted before examining generalized indexes. This means that the heuristics will never be applied (since they are only applicable to generalized indexes). When the disk space constraint input to the advisor is larger than the size of the *All Index* configuration, all the basic candidate indexes are selected by the greedy search algorithm in addition to other generalized candidate indexes. In this case, the heuristic rules are applied to select between basic and generalized candidate indexes by the greedy search with heuristics algorithm. In Appendix C, we show examples of our search algorithms in action, and we demonstrate cases where the heuristics would be used. We note that the heuristics were useful for the TPoX and XMark workloads when we used our Index Advisor with an earlier version of DB2 [32]. The optimizer of that previous version of DB2 overestimated the benefit of some general indexes,

which led to their benefit/cost ratios being smaller than other specific indexes, and they were considered by the search algorithm first. Therefore, the greedy search selected general and specific indexes for the same queries while ignoring other useful indexes. But the greedy search with heuristics applied the heuristic rules to avoid making such a selection.

Top down search performs the same as greedy search in Figures 3.14 and 3.15, also due to the fact that the generalized indexes are too large in size. For these large generalized indexes, and small disk space budgets, the top down algorithm eliminates all generalized indexes and still does not satisfy the disk space constraint. Hence, the algorithm resorts to greedy search on the basic candidate indexes.

Figure 3.16 shows the geometric mean of the speedups shown in Figure 3.14. We note that dynamic programming underperforms the other search algorithms at a disk budget of 70MB because it chooses an index that provides a small benefits (in relative terms) to a long running query over one that provides a larger relative benefit to a short query.

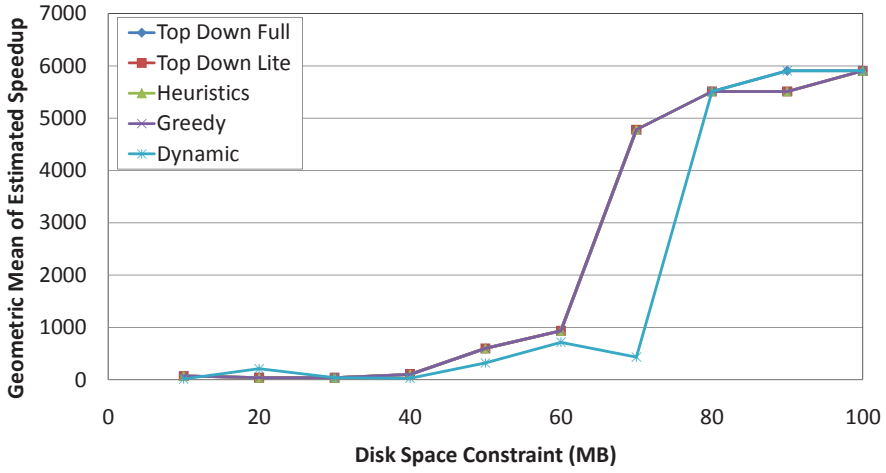


Figure 3.16: Geometric mean of estimated speedup, low disk budgets (TPoX).

To demonstrate the top down search algorithms, Figures 3.17 and 3.18 show the geometric mean of the estimated speedup for much larger disk space budgets for the TPoX and XMark benchmarks, respectively. The figures show that as the disk space budget increases, the top down algorithms choose index configurations that

are different from those chosen by greedy search. These configurations are not the best for the specific training workload (i.e, the speedup is not the highest), but this is expected since the goal of the top down algorithms is to enable generalization to unseen workloads and not to achieve the best possible performance on the training workload. We examine the generalization ability of the top down algorithms in the next section. Figure 3.17 shows some cases where top down full outperforms top down lite. This is due to the fact that top down full takes index interactions into account by re-evaluating the benefit of all unselected indexes given the selected indexes at every iteration, whereas top down lite does not modify the benefit of unselected indexes. In our workloads, this can result in a factor of 2 difference in performance between top down lite and top down full for queries where index interactions have an effect.

Figure 3.18 also shows the effectiveness of the heuristics that we add to greedy search. The figure shows that the greedy search algorithm sometimes chooses multiple indexes that answer the same query, thereby wasting some of the available disk space budget without gaining any benefit. Therefore, the greedy search does not achieve the same level of speedup as the greedy search with heuristics for the same disk space budget. Greedy search with heuristics avoids this problem by ensuring that any selected index does not replicate previously chosen indexes, and so it makes better use of the available disk space budget. As an example of the wastefulness of greedy search, we note that with a disk space budget of 1000MB, greedy search chooses a configuration with size 524MB, while greedy search with heuristics chooses a configuration with size 96MB, and both of these configurations have the same benefit (see Appendix C for details).

Figure 3.19 shows the actual speedup for the workload consisting of the 11 queries of the TPoX benchmark (the estimated speedup for this workload is presented in Figure 3.14). We can see that the actual speedup corroborates the conclusions drawn from the estimated speedup experiment. In addition, the figure shows that the benefit of some indexes is overestimated by the optimizer. For example, the dynamic programming search selects an index that turns out to be not as useful

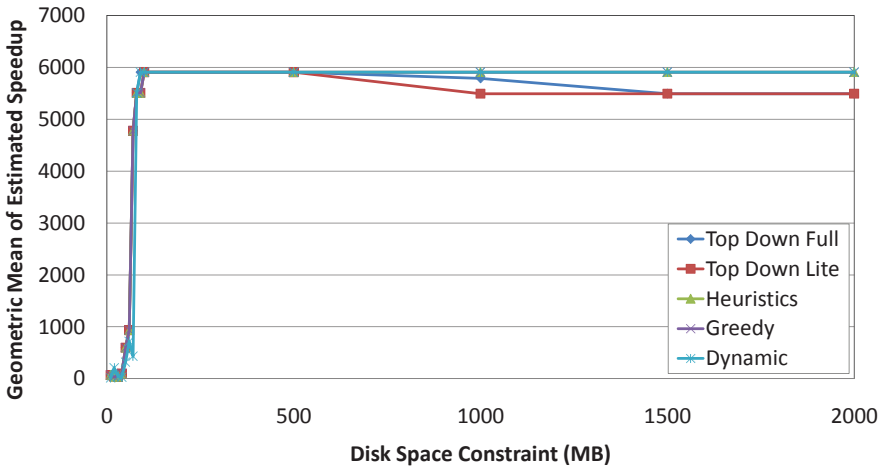


Figure 3.17: Geometric mean of estimated speedup (TPoX).

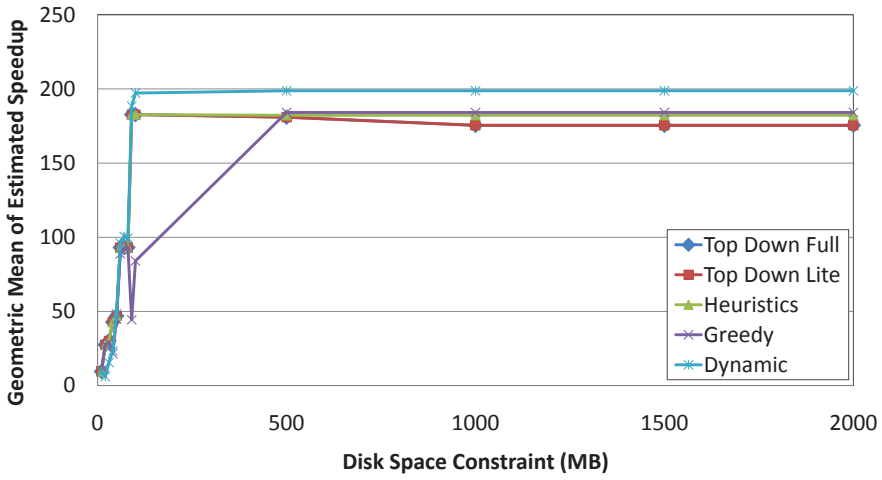


Figure 3.18: Geometric mean of estimated speedup (XMark).

as it was estimated by the query optimizer, which is why the actual speedup for dynamic programming search is much lower than that for top down search even though their estimated speedups are similar.

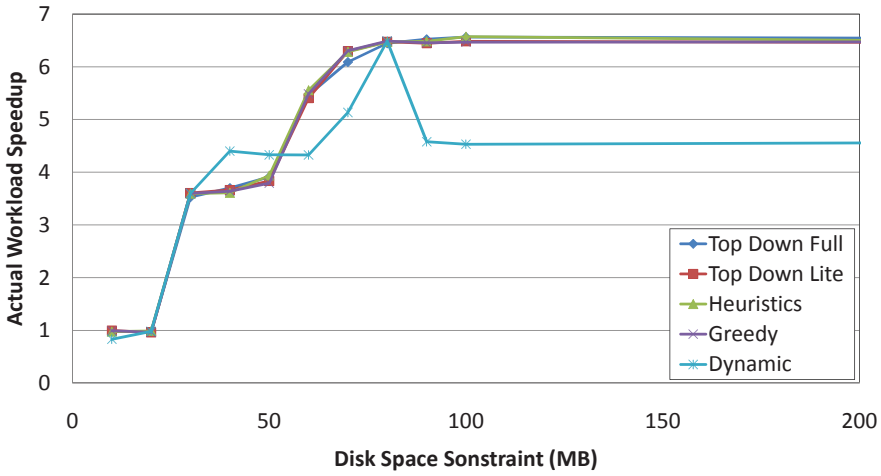


Figure 3.19: Actual speedup (TPoX).

3.8.3 Recommending General Indexes

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

The first question we address is how many generalized indexes can potentially be found in a workload. To address this question, we generated synthetic workloads consisting of random XPath path expressions that occur in the data. Table 3.4 shows for the TPoX workload the number of basic candidate indexes generated by the query optimizer in Enumerate XML Indexes mode for these workloads as the number of workload queries increases, and also the total number of candidate indexes after candidate generalization (Section 3.4). 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 25% to 50% by adding general candidate indexes.

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

Table 3.4: Number of candidate indexes after generalization (TPoX).

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

Tables 3.5 and 3.6 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 15 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. Figures 3.20–3.23 shows the selected indexes in the DAG of indexes generated for the TPoX workload for disk budgets 100MB, 500MB, 1000MB, and 2000MB (the selected indexes are shaded in a darker color). Appendix C has a detailed description of the indexes in the DAG generated for the TPoX workload.

To show the effect of recommending general indexes on the speedup of various workloads, we perform an experiment where the training workload used by the Index Advisor for recommending indexes is different from the test workload used to evaluate the recommended configuration. For TPoX, we used a workload of

Disk Budget	Heuristics	Top Down Lite	Top Down Full
100MB	G: 0, S:15	G: 0, S: 15	G: 0, S: 15
500MB	G: 0, S: 15	G: 1, S: 12	G: 2, S: 10
1000MB	G: 0, S: 15	G: 2, S: 8	G: 2, S: 9
1500MB	G: 0, S: 15	G: 4, S: 4	G: 4, S: 4
2000MB	G: 0, S: 15	G: 8, S: 0	G: 4, S: 4

Table 3.5: Number of generalized (G) and specific (S) indexes recommended (TPoX).

Disk Budget	Heuristics	Top Down Lite	Top Down Full
100MB	G: 1, S:10	G: 1, S: 10	G: 1, S: 10
500MB	G: 2, S: 9	G: 3, S: 5	G: 3, S: 5
1000MB	G: 2, S: 9	G: 2, S: 3	G: 2, S: 3
1500MB	G: 2, S: 9	G: 3, S: 0	G: 3, S: 0
2000MB	G: 2, S: 9	G: 3, S: 0	G: 3, S: 0

Table 3.6: Number of generalized (G) and specific (S) indexes recommended (XMark).

20 queries, the 11 TPoX queries followed by 9 synthetic queries generated as described above to increase workload diversity. For XMark, we used the 15 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 for TPoX and 15 for XMark). Figures 3.24 and 3.25 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. We choose 2GB to represent an infinite disk budget because any index configuration that can be recommended to either the TPoX or XMark workloads is guaranteed to have a size that is smaller than 2GB. A training workload with size n is the same as the training workload with size $n-1$ after adding one additional query to it. The figures show the speedup for

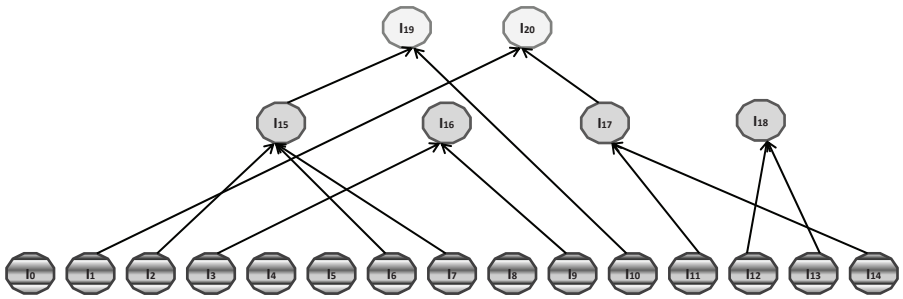


Figure 3.20: Selected indexes for disk budget of 100 MB (TPoX).

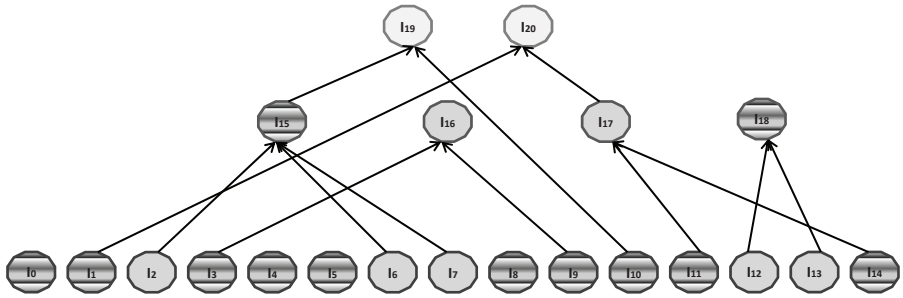


Figure 3.21: Selected indexes for disk budget of 500 MB (TPoX).

top down full search and greedy search with heuristics. In this case, the speedup of top down lite is similar to that of top down full, so we eliminate it from the figure for clarity. The figures show that as the advisor sees more and more of the test workload, it can recommend a configuration of indexes that can be useful to unseen queries. The figures also show that top down search is quite effective at using the available disk space to generalize from the queries seen in the training workload to the unseen queries in the test workload, whereas greedy search with heuristics is unable to perform such generalization.

Figures 3.24 shows the results of the experiment when we added TPoX queries to the training workload in one order. To confirm that the subset of queries chosen as the training workload does not affect the conclusion that we are making (i.e., the conclusion is not affected by the order of the queries), we repeat the above experiment for different orders of queries. In every run of this experiment, we change the order of the training queries input to the advisor. Figure 3.26 shows the average estimated speedup computed over five runs of the experiment using five

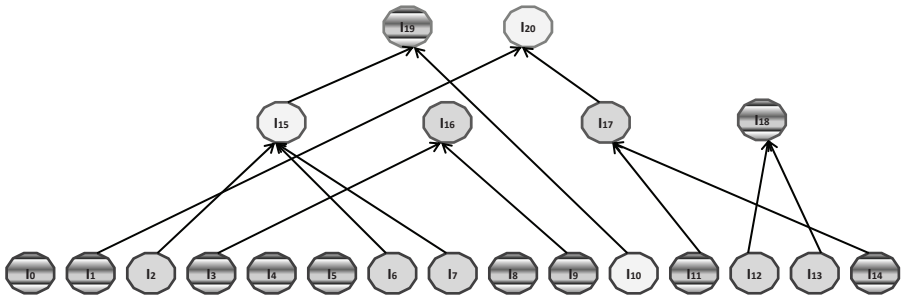


Figure 3.22: Selected indexes for disk budget of 1000 MB (TPoX).

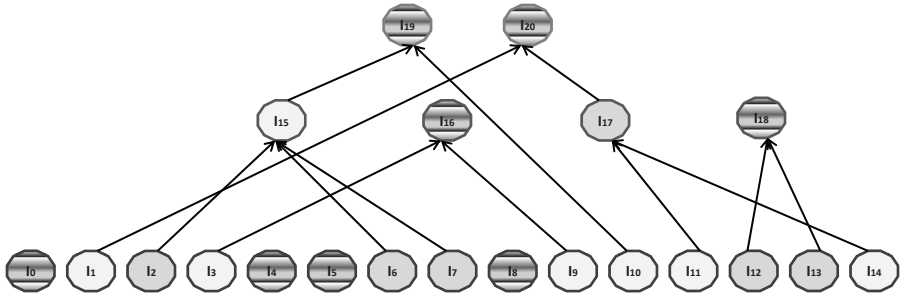


Figure 3.23: Selected indexes for disk budget of 2000 MB (TPoX).

different random permutations of the workload queries. The figure reaffirms the conclusions drawn from Figure 3.24: the top down search algorithm can generalize from the training workload to future unseen queries while the greedy search with heuristics cannot.

3.8.4 Advisor Run Time

Figure 3.27 shows the run time of the Index Advisor for varying disk space budgets on the TPoX workload. Top down full search takes up to 4 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 runtimes of the greedy search and the top down lite search are lower than all the other search algorithms and are not affected by changing the disk budget because both search algorithms check every candidate index at most once. In addition, the runtime of the dynamic programming search increases expo-

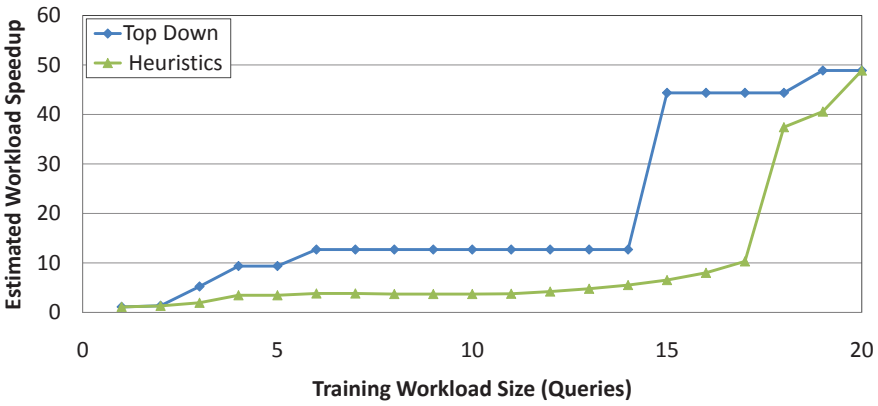


Figure 3.24: Generalization to unseen queries (TPoX).

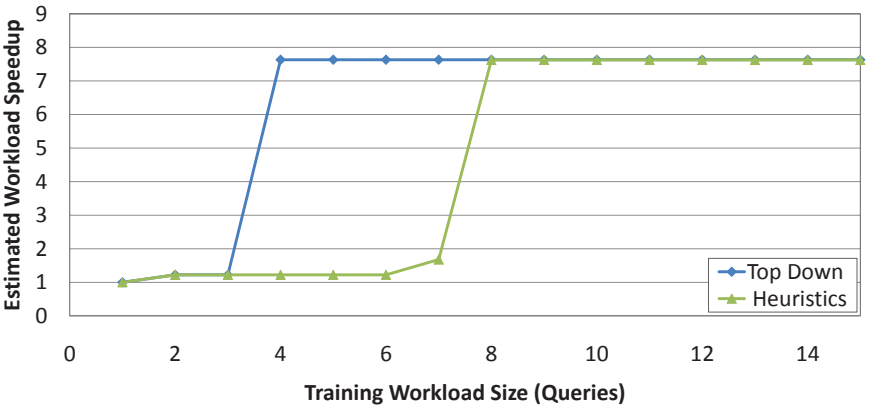


Figure 3.25: Generalization to unseen queries (XMark).

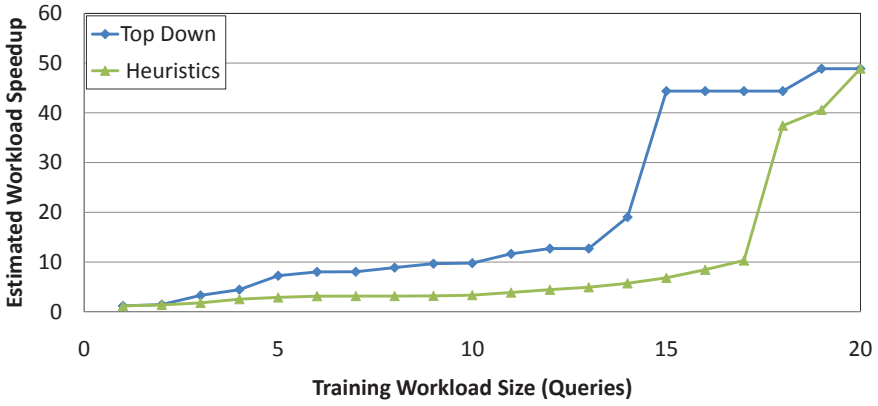


Figure 3.26: Generalization to unseen queries, average result of 5 workload permutations (TPoX).

nentially with the increase of the disk space budget. Thus, we can deduce that the recommendations of the dynamic programming search algorithm, which are sometimes better than the recommendations of other approximate search algorithms, come at a cost.

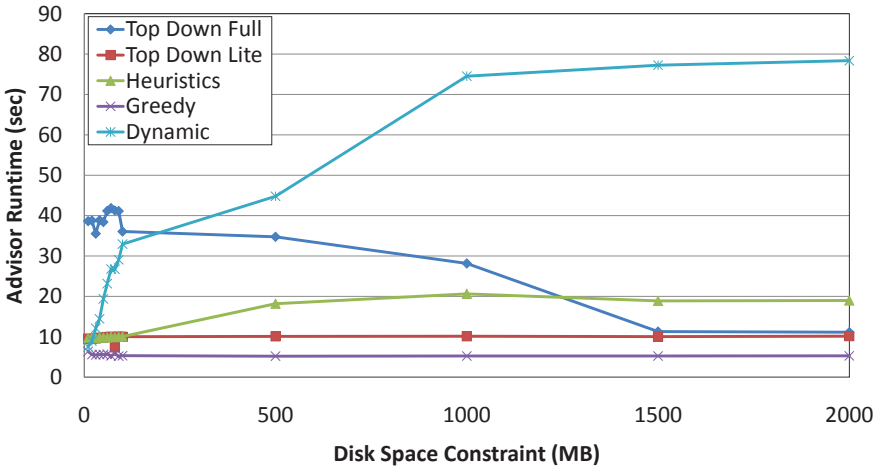


Figure 3.27: Advisor runtime (TPoX).

3.8.5 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 index configurations in the Evaluate XML Indexes optimizer mode, and in estimating in our client-side application the penalty of updating the index configuration when updating the database with update, insert, or delete (UDI) statements.

The key statistic used by Evaluate XML Indexes mode is the size of a virtual index. We have found that for the TPoX 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 are indexes on `/FIXML/Order/OrdQty/@*` and `/FIXML/Order//@*`, and we are able to estimate their size with 3.7% and 5.5% error, respectively.

Figure 3.28 illustrates the effect of estimating the penalty of updating candidate index configurations in response to UDI statements. We add to the TPoX workload a varying number of UDI statements that insert documents into one of the tables (the `ORDER` table), and we use the Index Advisor to recommend a configuration with a 100MB disk space budget. The figure shows the estimated execution time (in millions of optimizer time units, or *timerons* in DB2 terminology) as we vary the number of UDI statements. The figure shows the case where the design advisor ignores UDI statements while recommending an index configuration, and for the case where the design advisor takes UDI statements into account. The figure also shows the cost of the queries and the insert statements for the second case. As the number of UDI statements increases, workload execution time increases in all cases, but the advisor that takes into account UDI statements is able to reduce the increase in execution time by dropping indexes when the penalty for updating them exceeds their benefit (which happens when insertions are around 70% of the table size). The figure also shows that the queries in the workload suffer when indexes are dropped, but dropping the indexes saves time overall. Thus, from these experiments, we can see that we can effectively estimate the benefit of indexes and continue to do so in the presence of updates.

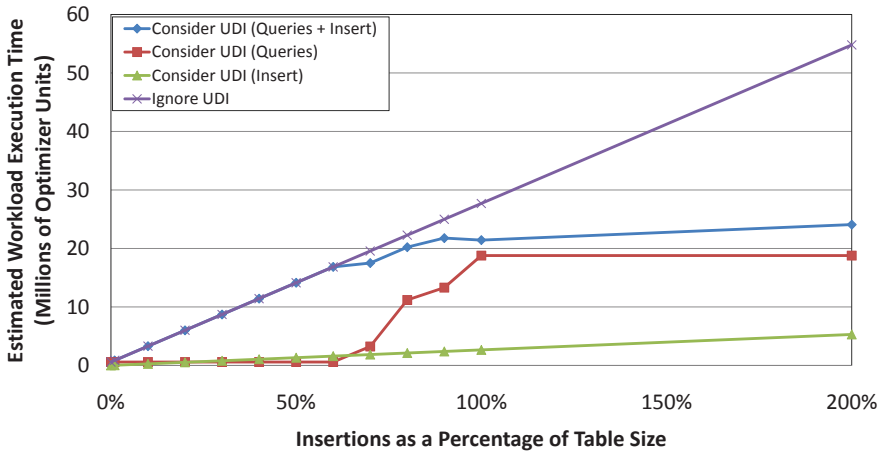


Figure 3.28: Effect of updates, budget=100MB (TPoX).

The overall conclusion of our experiments is that the XML Index Advisor can effectively recommend index configurations that significantly speed up workload execution while fitting within the available disk space budget. The DBA should decide on the search algorithm to use based on the goal of the search. If the goal is to recommend a configuration that is specific to the given workload, then greedy search with heuristics should be used. If the goal is to recommend a configuration that is useful for the given workload but can also benefit future, yet unseen queries, then the top down lite search algorithm should be used. The XML Index Advisor is a novel and powerful addition to the toolbox of the DBA of an XML database system. It deals with the structural complexity of XML data and workloads, and with the complexity of recommending partial indexes in this setting. Having described the Index Advisor, we turn our attention next to recommending materialized views for XML workloads.

Chapter 4

Recommending XML Views

In the previous chapter, we focused on indexes as one of the physical design structures for XML databases. We described an end-to-end XML Index Advisor that automatically recommends XML indexes that are represented by linear XPath path expressions, and we described techniques for extending the advisor to support multipath indexes that are represented by a set of XPath path expressions that can be used to construct a tree of the data. In this chapter, we study automatically recommending materialized views for XML databases.

4.1 Introduction

There are currently several types of materialized views for XML data. Many proposals have defined view languages for XML data and have studied matching these views with XML queries. In this chapter, we focus on enumerating and recommending XMLTable materialized views for a workload of XQuery queries. XMLTable views are relational views that use an SQL/XML extension, namely the XMLTable table function that is used to query XML data and export the result into a relational table[1]. One way to look at an XMLTable view is that it is a set of linear XPath path expressions where the results of executing these XPath queries are stored in a relational table. The main difference between what we propose in this chapter and

[1] A more detailed description of XMLTable views was given in Section 2.4.1

what we presented in the previous chapter for multipath XML indexes is that the XMLTable views that we focus on in this chapter represent an alternative *relational* access path of the XML data. This enables the query optimizer to use these views as it would use a regular relational table. We also deal with XMLTable views in this chapter as one unit, not as a set of linear XPath path expressions as we did for the multipath indexes in the previous chapter. Materializing XMLTable views helps the performance of XQuery queries by enabling the database system to pre-scan frequently queried XML data, perform complex operation in advance, and provide a normalized version of parts of the XML data.

Materialized views for XML databases are different from those for relational databases because the structure of the data is more complex and XQuery queries can reference any parts of the XML data in one row of the table using XPath path expressions. XML indexes are based on XPath path expressions and several indexes can be used together to answer a query. Materialized views for XML data are more complex (and more flexible) than indexes because one materialized view can capture not only several XPath path expressions in the XQuery query but also other operators such as join and projection. Furthermore, in this thesis we focus on using relational views of the XML data to answer XQuery queries, which adds extra complexity because it requires an intermediate translation of the XQuery queries to SQL queries.

The main issues that we need to address when recommending materialized views are: (1) determining the candidate physical structures (relational materialized views) that would be useful for an XQuery query or a workload consisting of a set of XQuery queries, (2) expanding the set of candidates by generating new ones that are useful for multiple queries in the workload as well as adding indexes that can help these views, and (3) searching the space of possible materialized view configurations for the optimal configuration that provides the maximum benefit to the workload while satisfying disk, schema, and other system constraints. In this chapter, we present novel techniques to address each of these challenges. We have implemented our XMLTable View Advisor in a prototype version of DB2 V9.7,

which supports both relational and XML databases, and we have used this implementation to verify the efficiency of our proposed advisor and the high quality of the view configurations that it recommends. Part of this work appeared in [34].

The rest of the chapter is organized as follows. In Section 4.2, we discuss the architecture of a database compiler/optimizer that translates XQuery queries into SQL/XML queries and matches and then rewrites the translated queries using materialized XMLTable views. We also discuss the architecture of our proposed XMLTable View Advisor. Next, we describe an end-to-end solution for an XMLTable View Advisor that recommends relational materialized views given an XML database and a workload of XQuery queries. Section 4.3 describes enumerating basic candidate materialized views. Section 4.4 describes generalizing these candidates. Section 4.5 discusses recommending relational indexes on the XMLTable views, and Section 4.6 presents our algorithm for searching for the best possible configuration of views. We describe how to translate XQuery queries into SQL/XML queries with XMLTable functions in Section 4.7, which is an extension that would be added to the database optimizer. Finally, we describe our prototype implementation of the XMLTable View Advisor in DB2 (Section 4.8) and present our experimental results (Section 4.9).

4.2 Overview of XMLTable View Recommendation

4.2.1 Database Compiler/Optimizer Architecture for Rewriting XQuery Queries with XMLTable Views

The XMLTable table function is an SQL extension that maps XML data into relational tables. Our advisor recommends XMLTable materialized views for improving the performance of XQuery queries. This requires the XQuery queries to be rewritten at run time by the query optimizer so that they use the XMLTable views. Thus, the query optimizer of a database system using our approach needs to

be extended with the ability to translate XQuery queries into SQL/XML queries that use XMLTable functions. This translation process allows the optimizer to match the translated query with the XMLTable views materialized in the database. Translating queries during compilation by the database engine to use the internal data structures is referred to as *native compilation*. The idea of natively compiling XQuery queries into SQL with extension operators to process them using a relational DBMS is described in [59]. In this thesis we use a similar (but more simplified) approach, described in detail in Section 4.7. The database compilation/optimization of XQuery queries to use materialized XMLTable views runs through the following steps, which are also illustrated in Figure 4.1:

1. **XQuery parsing.** The XQuery query is parsed into XQueryX [66], which is an XML representation of XQuery. This step helps the XQuery compiler analyze the clauses of an input query and prepare it for the following steps.

2. **XMLTable views enumeration.** Using the XQueryX format of the input XQuery query, the XQuery compiler examines all the clauses in the query and enumerates the possible XMLTable views that include all the XPath expressions that are referenced in the input query. We describe this process in more detail in Section 4.7.

3. **Generating SQL/XML query using enumerated views.** The XQuery compiler uses the XMLTable views enumerated in the previous step in translating the input XQuery query into an SQL/XML query that has these views in the FROM clause as sub-queries. Return constructs in the input XQuery are translated into SQL/XML publishing functions (Section 4.7).

4. **Matching query with materialized views and selecting the best matching materialized XMLTable views.** The optimizer matches the translated SQL/XML query with all the XMLTable views materialized in the database and applies a cost based function to select the best set of views to rewrite the query.

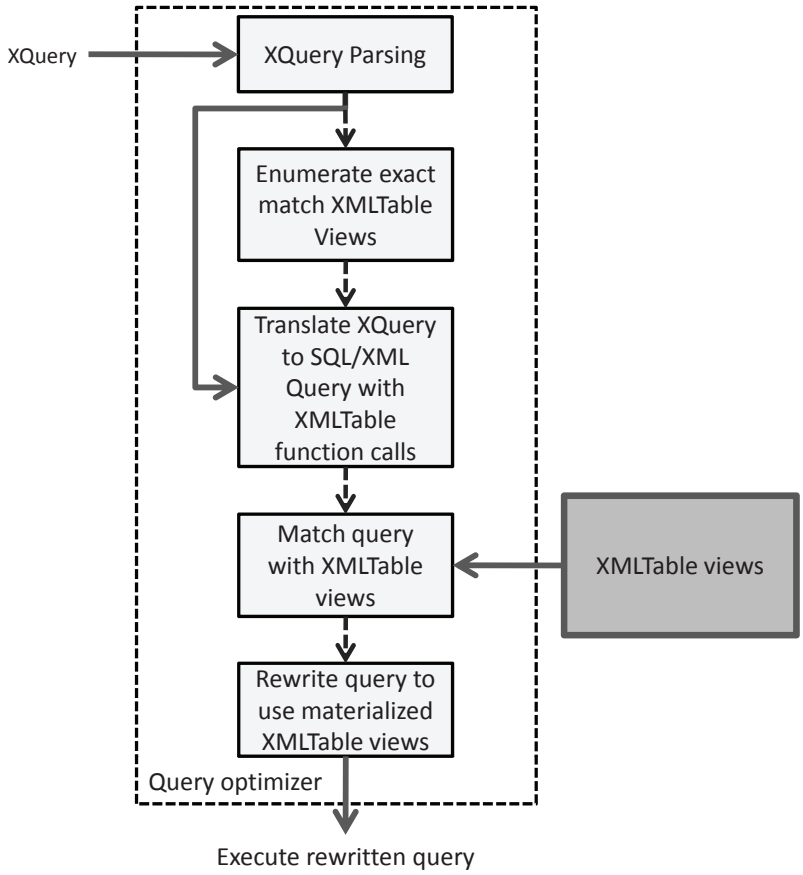


Figure 4.1: Query optimizer rewriting queries to use XMLTable materialized views.

5. **Rewriting query to use the selected views.** The final phase of the optimization process is to use the set of matched XMLTable views to rewrite the query.

To concretely demonstrate the kinds of materialized views that we aim to recommend, and how they would be used by a query optimizer, we give a brief example that illustrates how to translating an XQuery query to an SQL/XML query and rewriting it to use materialized views. A detailed example with a description of the translation rules is presented in Sections 4.3 and 4.7. Consider the XQuery query *XQeg* on the data of the TPoX [70] benchmark (shown below). The following steps summarize how the query optimizer would compile, translate, and rewrite *XQeg* to an SQL/XML query that uses XMLTable views:

Example query: XQeg: Return the name and the account information of the customer whose ID equals 9000.

```
for $cust in ("CUSTACC.CADOC")/Customer[@id = 9000]
return
    <customerInfo>
      <name>$cust/name</name> <accounts>$cust/Accounts/Account</accounts>
    </customerInfo>
```

1. First, query *XQeg* is parsed and the XMLTable views that encapsulate all the XPath expressions referenced in the query are enumerated. For *XQeg*, only one XMLTable view is generated in this step: view *Vrw* (shown below).

   Vrw: Generated XMLTable view that contains query *XQeg*.

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

2. Next, the query optimizer uses *Vrw* to translate *XQeg* to *SQeg* (shown below).

   SQeg: The rewriting of *XQeg* to use *Vrw*.

   ```
   select  XMLElement( NAME  "customerInfo" ,
               XMLElement( NAME  "name" , Vv0.cx1) ,
               XMLElement( NAME  "accounts" , Vv0.cx2))
   from  (  select v0.cx0, v0.cx1, v0.cx2  from CUSTACC, xmltable(
               '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
           columns
                   cx0  double path '@id',
                   cx1  string path 'name',
                   cx2  xml path 'Accounts/Account') as v0  ) as Vv0
    where  (  Vv0.cx0  =  9000  )
   ```

3. The optimizer then matches the translated query *SQeg* with all the XMLTable views that are available as materialized views in the database. In this example, we assume that there are two materialized views *Veg1* and *Veg2* (shown below) that match the translated query *SQeg*. Our aim in this chapter is to recommend materialized views that are similar to *Veg1* and *Veg2*.

   Veg1: Example XMLTable materialized view.

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

   Veg2: Example XMLTable materialized view.

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

4. Finally, the optimizer chooses one or more materialized view to rewrite the query. *Veg1* is similar to view *Vrw* which means that we can rewrite *SQeg* to use *Veg1* by replacing the sub-query *Vv0* in the FROM clause and all its references in the query with a reference to *Veg1*. The rewritten query using *Veg1* is *RSQeg1*. *Veg2* is a general form of *Vrw* and hence compensation is needed to rewrite the query to use the view. The rewritten query using *Veg2* is *RSQeg2*. In this thesis, we study rewriting XQuery queries into equivalent SQL/XML queries that use XLMTable views, but we do not consider

any rewriting for these SQL/XML queries that use XMLTable views. Thus, we would not be able to automatically generate the query *RSQeg2* since it requires a compensating expression on top of *Veg2*.

RSQeg1: Query *SQeg* after rewriting it using view *Veg1*.

```
select  XMLElement( NAME  "customerInfo" ,
            XMLElement( NAME  "name" , Veg1.cx1) ,
            XMLElement( NAME  "accounts" , Veg1.cx2))
from  Veg1
where  (  Veg1.cx0  =  9000  )
```

RSQeg2: Query *SQeg* after rewriting it using view *Veg2*.

```
select  XMLElement( NAME  "customerInfo" ,
            XMLElement( NAME  "name" , Veg2.cx1) ,
            XMLElement( NAME  "accounts" , Vv0.cxv2))
from  Veg2 ,
    (
        select v0.cxv2
        from Veg2, xmltable(
            '$cx2' passing Veg2.cx2 as "cx2"
        columns
            cxv2  xml path 'Account') as v0
    ) as Vv0
where  (  Veg2.cx0  =  9000  )
```

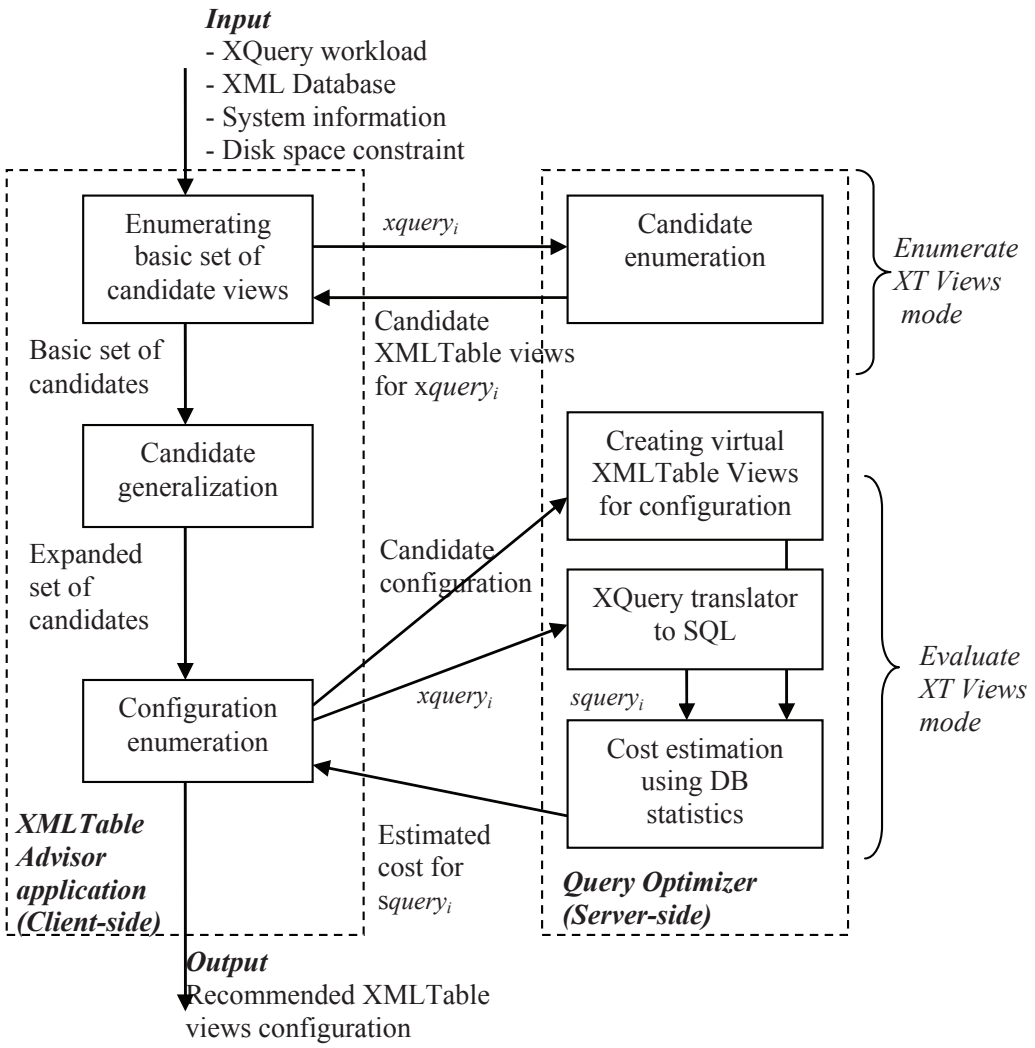


Figure 4.2: Architecture of the XMLTable View Advisor.

4.2.2 XMLTable View Advisor Architecture

We now present the architecture of our XMLTable View Advisor. Our view advisor architecture follows the same general architecture described in Chapter 1 (Figure 1.1). Figure 4.2 illustrates the specifics of the XMLTable View Advisor architecture. First, the XMLTable View Advisor analyzes each query in the workload and enumerates its possible XMLTable view candidates. The set of XMLTable views enumerated for all queries in the workload constitutes the basic set of candidate views for the entire workload. Next, the advisor expands the set of candidate views by recommending more general views that can answer multiple queries in the

workload. Then, for each candidate view, the advisor invokes the query optimizer in a special mode to estimate the benefit of the view to the queries in the workload. Finally, the advisor searches the space of candidates to find the best configuration of views that has the highest benefit to the workload and fits into the given disk space budget.

At a high level, the goal of the XMLTable View Advisor is to identify common access patterns in the input XQuery workload, and to extract the XML data accessed by these patterns into XMLTable views. For example, if the queries in the input workload frequently access the value of an element in the XML data (an ID element for instance), then it is beneficial to extract this element as a separate column in an XMLTable view. Moreover, for elements or attributes that are usually accessed together in the input workload, it is useful to store them as an XML construct in the XMLTable view. For example, if customers' phones are usually retrieved together in the input queries, then we deduce that it is beneficial to store them as an XML construct in the recommended views.

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

```
FLWORExpr      ::=     (ForClause  | LetClause)+ WhereClause?
                        ReturnClause
ForClause      ::=      "for" "$" VarName "in" Expr
                        ("," "$" VarName "in" Expr)*
LetClause      ::=      "let" "$" VarName ":=" Expr
                        ("," "$" VarName ":=" Expr)*
WhereClause    ::=      Expr

ReturnClause   ::=      Expr  | ConstructExpr
```

```
ConstructExpr  ::=   "<" ConstructName ">"
                     (Expr  | ConstructExpr)+
                     "</" ConstructName ">"
Expr           ::=   OrExpr
OrExpr         ::=   AndExpr  ( "or" AndExpr  )*
AndExpr        ::=   ComparisonExpr ( "and" ComparisonExpr )*
ComparisonExpr ::=   SingleExpr  ( ComparisonOp  SingleExpr )?
SingleExpr     ::=   PathExpr  | Literal
```

`VarName` is a valid XQuery variable name. `ConstructName` is an XML tag name of an XML constructed return value. `PathExpr` is any valid XPath path expression. `Literal` can be a numerical or string literal. `ComparisonOp` can be: (1) a general comparison: "=" | "≠" | "<" | "≤" | ">" | "≥", (2) a value comparison: "eq" | "ne" | "lt" | "le" | "gt" | "ge", or (3) a node comparison: "is" | "≪" | "≫".

Next, we describe the phases of the view recommendation process in detail. We use the following query *Q1*, which is the same as query *Qeg* discussed above after adding more clauses to it, as a running example:

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

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

4.3 Enumerating Candidate Views

In Chapter 3, we enumerated candidate indexes by completely relying on the optimizer index matching algorithm to identify the indexable expressions in the XQuery query. XMLTable views are more complex physical structures, since views can include XML data from multiple XML paths. Therefore, it would be more difficult for the query optimizer to enumerate candidate views due to the fact that there is no simple equivalent to the `//*` index for materialized views. Because of that, we decided to develop a process for enumerating candidate XMLTable materialized views that does not rely on the query optimizer. The process for enumerating XMLTable views relies on translating the input XQuery queries into SQL/XML queries that use XMLTable functions, as described briefly in Section 4.2. During this translation process, potentially useful XMLTable views are identified and added to the set of candidate views. We describe the XQuery-to-SQL/XML translation algorithms that we use to enumerate candidate views in this section. In Section 4.7, we revise these algorithms to enable translating XQuery queries into SQL/XMLTable queries that use the recommended materialized views.

To enumerate candidate views for an XQuery query, we parse the query and break it down into its FOR, LET, WHERE, and RETURN clauses. Then, for each one of these clauses we further break it into its components. The FOR and LET clauses in the FLWOR expression are used to produce a tuple stream in which each tuple consists of one or more bound variables. This behavior resembles the row generator in the XMLTable function. Therefore, for every FOR or LET clause in the input XQuery, we create a new candidate XMLTable view. Every reference to the bound variable declared in a FOR or LET clause represents a navigation to an element in its expression subtree, and hence we add a column navigator in the corresponding view. We describe next the details of how we handle each clause in the candidate enumeration process (Algorithm 12 and the helper functions described in Algorithms 13–17).

Algorithm 12 enumerateCandidates(*xquery*)

1: **for** *clause* ∈ *xquery* **do**
2: **if** *clause is forClause* **then**
3: call *enumerateCandidates–handleForClause*(*clause*)
4: **else if** *clause is letClause* **then**
5: call *enumerateCandidates–handleLetClause*(*clause*)
6: **else if** *clause is whereClause* **then**
7: call *enumerateCandidates–handleWhereClause*(*clause*)
8: **else if** *clause is returnClause* **then**
9: call *enumerateCandidates–handleReturnClause*(*clause*)
10: **end if**
11: **end for**

Algorithm 13 enumerateCandidates–handleForClause(*forClause*)

1: break *forClause* into *forVarName* and *forExpr*
2: *view* ← *enumerateCandidates–handleExpr*(*forVarName*, *forExpr*)

Algorithm 14 enumerateCandidates–handleLetClause(*letClause*)

1: break *letClause* into *letVarName* and *letExpr*
2: *view* ← *enumerateCandidates–handleExpr*(*letVarName*, *letExpr*)
3: **if** *letClause* has *aggFn* **then**
4: add an SQL GROUP BY clause to *view* with all columns except the expression that appears in the *aggFn*
5: update the SELECT clause of *view* to reflect applying the *aggFn*
6: **end if**

Algorithm 15 enumerateCandidates–handleWhereClause(*whereClause*)

1: **for** *comparisonExpr* found in the *whereClause* **do**
2: **for** *pathExpr* found in the *comparisonExpr* **do**
3: find *refView* associated with the *varRef* referenced in *pathExpr*
4: add *pathExpr* to *refView* as a column navigator
5: **end for**
6: **end for**

Algorithm 16 enumerateCandidates–handleReturnClause(*returnClause*)

1: **for** *pathExpr* found in the *returnClause* **do**
2: find *refView* associated with the *varRef* referenced in *pathExpr*
3: add *pathExpr* to *refView* as a column navigator
4: **end for**

Algorithm 17 enumerateCandidates–handleExpr(*varName*, *expr*)

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

FOR Clause. We divide the FOR clause into a variable, the path expression associated with the variable (the binding sequence for that variable), and the optional predicates. A FOR clause produces a tuple stream for every variable and iterates over the binding sequence of that variable. We deduce that the behavior of the binding sequence of the FOR clause is similar to the behavior of the row generator of the XMLTable function. Therefore, for every FOR clause: (1) we create a new candidate view and assign its row generator to be the binding sequence in the FOR clause (i.e. the path expression after removing any predicate values from it, e.g., `/Customer` in the FOR clause of *Q1*), (2) we record the variable name and the created view so that we can add any expression that references the variable to the view as a column navigator, and finally (3) for every predicate expression appearing in the binding sequence of the FOR clause, we add it as a column navigator path expression to the view. For example, when we parse the FOR clause of *Q1*, we create a new view *V1* that has the row generator `/Customer` and the column `@id`:

V1:

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

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

For a binding sequence that references another variable (e.g., the expression `$cust/Accounts/Account` in *Q1*), we look up the expression referenced by this variable (`$cust` references `/Customer` in the FOR clause, which is also associated to the already generated view *V1*) and concatenate it with the rest of the expression to form the path expression that we use as a row generator when creating the XMLTable view (`/Customer/Accounts/Account` is used as the row generator for *V2* in this example). We then add a column in each of the views: (1) a column in the newly generated view (*V2*) to backward navigate the row generator of the view that represents the referenced variable in the binding sequence (the column `parent::Accounts/parent::Customer` in *V2* references `/Customer` in *V1*) and (2) a `"."` column in the referenced view (*V1*). These columns are used for joining the two views in the translated query. Additionally, a LET clause might have an optional aggregation function that we handle by adding the aggregation of the `"."` column to the SELECT clause of the XMLTable view (`count(u.cy0)` in *V2*). The updated version of *V1* and the newly generated *V2* will be as follows:

V1:

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

V2:

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

WHERE Clause. For every predicate appearing in a WHERE clause, we extract the XPath expressions appearing in this predicate. For each XPath path

expression, we lookup the view associated with the referenced variable in this expression and add a column to that view to correspond to this navigation. For example, to account for the predicate on `age` in *Q1*, we add a column navigator in view *V1*, which will be updated to look as follows:

V1:

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

RETURN Clause. For all the XPath path expressions that appear in the RETURN clause, we find all the views that are associated with the reference variables that appear in these expressions and we then add a column for each expression to the corresponding view. For example, the expression `$accounts` in the RETURN clause of *Q1* references an existing column in *V2* and hence no change is needed to the view. However, for the expression `$cust/name`, we add the column `name` to *V1*. The updated version of view *V1* is as follows:

V1:

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

After parsing queries in the workload, we have a set of enumerated views for this workload. Next, we expand this set of candidates through generalization rules.

4.4 Generalizing the Set of Enumerated Views

Recall that the XML Index Advisor generalizes the index patterns to make them useful for queries not seen in the input workload that is used for recommendation. Similarly, creating XMLTable views that answer multiple queries in the workload and potential unseen queries can increase the usefulness of our recommendations. Since our proposed view definition involves both XPath expressions and SQL query definitions, generalization can benefit from the index generalization techniques proposed in Chapter 3 and the query merging techniques proposed in [57, 92]. The possible generalization techniques include generalizing the row generator or the column navigators of the view and merging views. We describe the forms of query generalization that we have explored in this section. The XMLTable View Advisor applies these generalization rules to the basic set of candidate views to generate an expanded set of candidate views.

4.4.1 Generalizing Column Navigators to Include Subtrees

Most of the XMLTable views that we recommend in the enumeration phase are a normalization (flattening) of all the values that are being accessed in the workload queries. An alternative approach is to recommend views that store sub-trees of the data as XML columns. A recommended XMLTable view can now have one column navigator with a `"."` path expression to represent all the subtrees reachable by the row generator. For example, *V3* (below) is a generalization of *V1*. This approach is useful when the query requires reconstructing the XML tree. This general view requires that the matching infrastructure allow matching multiple columns in the query with one column in the view and is also capable of performing XPath compensation. For example, matching view *V3* with sub-query *Vv0* in the translated query *RQ1* (listed in Section 4.7) means matching columns *cx0*, *cx1*, *cx2* and *cx3* in *Vv0* with *cx0* in *V3*. Compensation is required in this case to navigate to *@id*, *age*, and *name* for columns *cx0*, *cx2*, and *cx3*, respectively, to rewrite the translated query to use the view *V3*.

Instead of replacing the column navigators of a view with a "." column, a less aggressive approach is to generalize pairs of column navigators that come from a different pair of views that share the same row generator using the index generalization algorithms proposed in Section 3.4. Since our XMLTable View Advisor does not consider rewriting SQL/XML queries (and hence does not consider generating compensating queries), we do not implement these generalization rules and we leave the details of generating the required compensating queries to future work.

V3:

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

4.4.2 Merging Views

A common generalization approach used in relational advisors is view merging [92]. For XMLTable views, we merge views that have the same row generator to produce a new view that has the set of column navigators that appear in the merged views after removing duplicates. The goal of this approach is to decrease the disk space required for views by removing duplicate columns from the merged views, while still achieving the same performance. This approach is a special case of the approach we discussed in Section 4.4.1, since we are keeping the normalization state (flat or nested) of the column navigator. For example, view *V5* is a merging of *V1* (listed in Section 4.3) and *V4* (shown below).

V4:

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

V5:

```
select u.cx0, u.cx1, u.cx2, u.cx3, u.cx4
from CUSTACC, xmltable(
    '$cadoc/Customer' passing CUSTACC.CADOC as "cadoc"
     columns
            cx0  double path '@id',
            cx1  xml path '.',
            cx2  double path 'age',
            cx3  varchar(100) path 'name',
            cx4  varchar(100) path 'occupation') as u
```

4.5 Indexes on XMLTable Views

One approach to make XMLTable views more useful is to build relational indexes on their columns. This is possible since the XMLTable views are regular relational SQL tables with indexable columns that happen to originate from XML data. There can be many possible indexes that can be built on the columns of an XMLTable view to help the view perform better. In this thesis, we use a heuristic approach to select only one index for each view. The chosen index has all the columns of the view that appear in a predicate in the XQuery that caused this view to be recommended. This guarantees that these columns have relational values that are used for lookup in the query. The index follows the same order of the columns in the view. For example, the index that we build for view *V1* is index *I1*. For every candidate view, we add to the search space another alternative structure that consists of the view with a relational index on its columns.

I1: Generated relational index for view V1.

```
create index index1 on V1(cx0, cx2)
```

4.6 Searching for the Optimal View Configuration

The techniques presented in the previous two sections result in a set of candidate views being enumerated: basic candidate views generated directly from the input queries and generalized candidate views obtained by applying the generalization rules. To choose some of these enumerated XMLTable views (a view configuration) to recommend for a workload, we search the space of enumerated candidate views to find the best set of views that fits in a given disk space budget. We generalize the search algorithms in Section 3.6 to be able to search any physical structure (indexes, views, views with indexes over them, etc.). The search problem can be modeled as a 0/1 knapsack problem similar to that discussed in Section 3.6. We compute the benefit of a physical structure as the difference between the workload cost as estimated by the query optimizer before and after creating this structure. In this section, we focus on developing rules that are similar to the ones described in Section 3.6.2 for the greedy search with heuristics algorithm. The top down search algorithm described in Section 3.6.3 can be used without any changes for searching the candidate XMLTable views.

XMLTable views can interact with each other in ways that affect their total benefit for a query workload. Our search algorithm takes such interactions into consideration. The main types of interaction affecting the selection of views are: (1) views that can be used together to rewrite a query and (2) views that are generated by merging other views and therefore subsume those views. These interactions are similar to the ones encountered when searching the space of XML indexes, so we use a greedy search algorithm similar to the one described in Section 3.6, but we modify the heuristic rules used in this search to deal with the new types of interactions so that they suit the view search problem.

The high-level outline of the greedy search algorithm is as follows. First, we estimate the size of each candidate view, and the total benefit of this view for the workload. We then sort the candidate views according to their benefit/size ratio.

Finally, we add candidates to the output configuration in sorted order of benefit/size ratio if they agree with the heuristic rules, starting with the highest ratio, and we continue until the available disk space budget is exhausted. In Section 3.6, we proposed heuristic rules that are based on *index coverage*. We define the *view coverage* of a view as its view ID as well as the IDs of the views that it subsumes (i.e., the views that it was generated from using the generalization algorithm and the views that have the same row generator and column navigators that were enumerated for other queries). The coverage of a configuration of views is defined as the combination of the view coverage of its constituent views. For example, if *V5* is generated by merging *V1* and *V4*, then the coverage of *V5* is the set {1, 4, 5}. We refer to the coverage of a candidate view (*cand*) or a group of views (*config*) as *cand.coverage* and *config.coverage*, respectively. We also refer to the size of a candidate view (*cand*) or a group of views (*config*) as *cand.size* and *config.size*, respectively. Algorithm 18 outlines the search algorithm. We use the following functions to perform the search and apply the heuristics:

- *benefit*(*config*) returns the estimated benefit of the workload when this configuration of views (or views with relational indexes on them) is created. It is based on calling the query optimizer with and without the views in place and computing the reduction in the optimizer's estimated cost when the views are in place.

- *addCandIfSpaceAvl*(*cand*, *config*) adds the candidate *cand* to the set *config* if $cand.size + config.size \leq diskConstraint$. In addition, if the condition holds, *addCandIfSpaceAvl* updates the *size* and *coverage* of *config*.

- *replaceCandIfSpaceAvl*(*cand*, *subConfig*, *config*) replaces the *subConfig* in *config* with *cand* if the new configuration after performing the replacement *newConfig* has a higher benefit than *config* and the added size is below a threshold β. This is the heuristic that we add to the greedy search to deal with view interactions. The value β is a threshold that specifies how much increase in size we are willing to allow. We have found $\beta = 10\%$ to work well in our

experiments. Finally, if the condition holds and there is enough disk space to do the replacement, *replaceCandIfSpaceAvl* updates the *size* and *coverage* of *config*.

- *overlapCoverage*(*cand*, *config*) scans a configuration *config* and returns the maximal set of candidates *overlapConfig* that has the view coverage of *cand* or has a view coverage that is less than *cand.coverage*.

Algorithm 18 heuristicViewSearch(*candidates*, *diskConstraint*)

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

4.7 Translating XQuery Queries into SQL Queries that Use XMLTable Views

The search algorithm described in the previous section results in a recommended configuration of XMLTable views that would help the performance of future XQuery queries. At run time, the query optimizer needs to translate the XQuery queries to

SQL queries with XMLTable functions to be able to match these queries with the XMLTable views. Translation of XQuery queries to SQL queries with XMLTable functions during query compilation is studied in [59]. We adopt a similar approach that we describe in this section. The translation involves using XMLTable views that are similar to the ones being enumerated for the XQuery queries using Algorithm 12 (Section 4.3). This ensures that the XMLTable views in the translated XQuery queries will match the recommended XMLTable views.

During the XQuery to SQL translation, we examine the parsed XQuery, generate XMLTable views that encapsulate all referenced XPath expressions in the query, and then construct an SQL query based on this information. We add all the generated views to the FROM clause of the SQL query. We then construct the SELECT and WHERE clauses in the translated query by referring to the columns of the views to reflect how their associated expressions appear in the original query. We also add joins between the views that are used to rewrite the query when needed. These joins are needed to link two FOR or LET clauses where one references the other (this is equivalent to joining two tables in SQL) to make sure that the data referenced by both clauses at any iteration is the same. For example, the binding sequence of the LET clause in *Q1* (`$cust/Accounts/Account`) references the binding sequence of the FOR clause (`/Customer`). Therefore, we add an equality predicate for the expressions represented by `$cust` referenced in the FOR and LET clauses to make sure that the XML data is the same at any iteration (i.e., we are aggregating the accounts of the same customer at any iteration).

Algorithm 19 and its helper functions outlined in Algorithms 20–22 illustrate the extensions we made to Algorithm 12 and its helper functions to build the three lists *selectElementsList*, *fromViewsList*, and *wherePredicatesList* that we use to construct the translated query. The lines added in this section to the algorithms of Section 4.3 are underlined. First, we initialize the three lists in Lines 1, 2, and 3 in Algorithm 19. For every FOR or LET clause in the query, we record the views that we create by adding them to *fromViewsList* in Lines 3 (Algorithm 20) and 7 (Algorithm 21). For every predicate we encounter during the parsing either in an

Algorithm 19 translateXQuery(*xquery*)

1: *selectElementsList* $\leftarrow \phi$
2: *fromViewsList* $\leftarrow \phi$
3: *wherePredicatesList* $\leftarrow \phi$
4: **for** *clause* $\in$ *xquery* **do**
5: **if** *clause is forClause* **then**
6: call *enumerateCandidates–handleForClause*(*clause*)
7: **else if** *clause is letClause* **then**
8: call *enumerateCandidates–handleLetClause*(*clause*)
9: **else if** *clause is whereClause* **then**
10: call *enumerateCandidates–handleWhereClause*(*clause*)
11: **else if** *clause is returnClause* **then**
12: call *enumerateCandidates–handleReturnClause*(*clause*)
13: **end if**
14: **end for**
15: generateQuery(*selectElementsList*, *fromViewsList*, *wherePredicatesList*)

Algorithm 20 translateXQuery–handleForClause(*forClause*, *fromViewsList*)

1: break *forClause* into *forVarName* and *forExpr*
2: *view* $\leftarrow$ *enumerateCandidates–handleExpr*(*forVarName*, *forExpr*)
3: add *view* to *fromViewsList*

Algorithm 21 translateXQuery–handleLetClause(*letClause*)

1: break *letClause* into *letVarName* and *letExpr*
2: *view* $\leftarrow$ *enumerateCandidates–handleExpr*(*letVarName*, *letExpr*)
3: **if** *letClause* has *aggFn* **then**
4: add an SQL GROUP BY clause to *view* with all columns except the expression that appears in the *aggFn*
5: update the SELECT clause of *view* to reflect applying the *aggFn*
6: **end if**
7: add *view* to *fromViewsList*

Algorithm 22 translateXQuery–handleExpr(*varName*, *expr*)

```
1: create a new view view and associate it with the variable name varName
2: break expr into pathExpr and predicateList
3: if pathExpr has a variable reference varRef then
4:    find refView associated with varRef
5:    set the row generator of view to be the concatenation of the row generator
      of refView and pathExpr
6:    add column "." to refView and a backward navigation path "refCol" to view
7:    construct predicate joinPred to join columns "." in refView and "refCol" in view
8:    add predicate joinPred to wherePredicatesList
9: else
10:   set the row generator of view to be pathExpr
11: end if
12: for p ∈ predicateList do
13:   for pathExpr found in the p do
14:      add pathExpr to view as a column navigator
15:   end for
16:   add predicate p to wherePredicatesList
17: end for
    return view
```

Algorithm 23 translateXQuery–handleWhereClause(*whereClause*)

```
1: for comparisonExpr found in the whereClause do
2:    for pathExpr found in the comparisonExpr do
3:       find refView associated with the varRef referenced in pathExpr
4:       add pathExpr to refView as a column navigator
5:    end for
6:    add predicate comparisonExpr to wherePredicatesList
7: end for
```

Algorithm 24 translateXQuery–handleReturnClause(*returnClause*)

1: **for** *pathExpr* found in the *returnClause* **do**
2: find *refView* associated with the *varRef* referenced in *pathExpr*
3: add *pathExpr* to *refView* as a column navigator
4: **end for**
5: construct return value *returnVal*
6: add *returnVal* to *selectElementsList*

expression appearing in a FOR or LET clause or in a WHERE clause, we add a reference to it in the *wherePredicatesList* (Line 16 in Algorithm 22 and Line 6 in Algorithm 23). When a binding sequence references a previously defined variable, we interpret this occurrence as a join between the referenced view and the new view. The predicate added for this join is illustrated in Lines 7 and 8 (Algorithm 22). Finally, we call the function *generateQuery* to construct the translated query from the three lists *selectElementsList*, *fromViewsList*, and *wherePredicatesList*. The *generateQuery* function uses a template of an SQL query with SELECT, FROM, and WHERE clauses to construct the translated query as follows: (1) simple elements or XML constructs in *selectElementsList* are added to the SELECT clause of the query, (2) references to views in *fromViewsList* are added to the FROM clause of the query, and (3) all predicates in *wherePredicatesList* are added to the WHERE clause. If the return value is a simple XPath expression, then the corresponding column name is used, otherwise if an XML fragment is constructed, an XQuery construction is done using the XMLElement SQL function. The pseudocode of *generateQuery* that is used to generate the translated query *TQ* is given in Algorithm 25.

To illustrate our translation process, we present the following example which shows translating query *Q1*. The two views *V1* and *V2* are recommended for query *Q1*, so we construct the FROM clause in the translated query as `from V1, V2`. Next, we examine the return clause and construct the SELECT clause of the rewritten query. A return XML fragment with two elements `name` and `accounts_number`

Algorithm 25 generateQuery(*selectElementsList*, *fromViewsList*, *wherePredicatesList*)

```
1: initialize query TQ
2: for selectElement ∈ selectElementsList do
3:    if selectElement is simple then
4:       find selectElementColRef as the column reference of selectElement in the
         list of views fromViewsList
5:       concatenate selectElementColRef to TQ.SELECT
6:    else if selectElement is construct then
7:       build selectElementConstruct using SQL/XML publishing functions (For
         example, XMLElement)
8:       concatenate selectElementConstruct to TQ.SELECT
9:    end if
10: end for
11: for view ∈ fromViewsList do
12:    concatenate view to TQ.FROM
13: end for
14: for predicate ∈ wherePredicatesList do
15:    concatenate predicate to TQ.WHERE
16: end for
```

is created via the XMLElement function and added to the SELECT clause of the translated version of the query. Finally, we construct the WHERE clause as a conjunction of all the predicates that appear in the XQuery and those that correspond to joins between views. The final translated query for *Q1* is as follows:

Translated Query: SQ1

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

4.8 Implementation

We have used a prototype version of IBM DB2 V9.7 that was modified to support creating materialized views using the XMLTable function as well as matching them

with queries [1]. The client side of the XMLTable View Advisor is implemented in Java 1.6 and communicates with the prototype server via JDBC. The optimizer infrastructure that supports rewriting XQuery queries to use materialized XMLTable views described in Section 4.2.1 is not fully implemented in DB2. For example, the XQuery to SQL/XML translation described in Section 4.7 is not currently implemented inside the engine. Therefore, we implemented the translation algorithm in our client side application. The XMLTable View Advisor sends workload queries after translating them into SQL/XML to the database optimizer to match them with materialized XMLTable views.

The current infrastructure of DB2 did not allow us to implement the candidate enumeration algorithm described in Section 4.3 inside the database engine. Hence, we have implemented the candidate enumeration algorithm in the client side application of the XMLTable View Advisor (in contrast to the *Enumerate XML Indexes* mode that we added as an extension to the DB2 optimizer for the XML Index Advisor). Enumerating candidate physical structures in the advisor and not in the query optimizer is an approach that has been used by prior design advisors such as Microsoft's Database Tuning Advisor (DTA) [3, 4, 23].

To evaluate the benefit of an XMLTable view, we need to create it as a virtual view and call the query optimizer in what-if mode to optimize the workload queries with this virtual view in place. For XML indexes, we added the ability to create virtual XML indexes to DB2 and implemented the Evaluate XML Indexes query optimizer mode as an EXPLAIN mode fully integrated in DB2. For XMLTable views we adopted a simpler but not fully integrated approach. Instead of implementing virtual XMLTable views in DB2, we implement our what-if infrastructure for estimating the cost of a query in the presence of an XMLTable view as follows. We actually create the XMLTable view (or configuration of views) whose benefit we want to estimate, populate it with data from the database, and then estimate the query execution time using the query optimizer's regular relational EXPLAIN mode (this EXPLAIN mode is an integral part of DB2 since its early versions). The estimated execution times obtained using this approach are fairly

accurate since they are based on regular, fully-populated XMLTable views. Thus, this approach enables us to estimate the benefit of XMLTable views with a high degree of accuracy. Hence, we can study the effectiveness of our XMLTable View Advisor at recommending view configurations that benefit the workload, and we can accurately measure the benefit of the recommended configurations.

What we cannot do with our current implementation is to obtain meaningful measurements of the run time of the XMLTable View Advisor since we use actual materialized views instead of virtual materialized views. We do not believe this to be a major concern for two reasons: First, creating virtual physical structures has been a staple of physical design advisors for a long time, so creating virtual XMLTables should be straightforward (although it does require access to source code and a significant effort in coding). Second, our advisor uses a greedy search algorithm, and greedy search is well-known to be fast, so we do not anticipate the time taken by the search algorithm to be a bottleneck in a production version of our advisor. The run time of our prototype View Advisor was not a bottleneck during our experiments. A typical run of the advisor on the TPoX benchmark would take around 70–85 minutes.

In both candidate enumeration and XQuery translation, we use the XQuery Normalizer and Static Analyzer (XQNSTA) [82] to normalize XQuery queries into XQueryX format. We then, parse the resulting XQueryX for every query to examine its blocks and enumerate its candidate XMLTable views.

Our XMLTable View Advisor implementation has some limitations due to the existing DB2 infrastructure. We can only use SQL data types for columns that appear in the XMLTable functions, since casting XML data elements into their corresponding relational data types fails in some cases. However, we use XML type for column navigators that represent an XML fragment of the data. In addition, we limit expressions that appear in column generators in the XMLTable functions to only select one element because if multiple elements are reachable by an XPath path expression then the values of all these elements are concatenated into one string value, and we lose their XML structure, whereas XQuery queries require this struc-

ture during execution. Additionally, our implementation does not support more than two joins per query. We have also left adding support for structured queries, which are XQuery queries with a sub-query in the return clause, for future work. However, these limitations have not prevented us from verifying the usefulness of XMLTable views to answer XQuery queries.

4.8.1 Implementation Requirements

The basic requirement for a database system to support our XMLTable View Advisor is that it has to be able to match XMLTable materialized views with SQL/XML queries that use the XMLTable function in their FROM clause (which is the support provided by DB2). The XMLTable View Advisor requires extending such a database system to support the following functionality: (1) the ability to translate XQuery queries to SQL/XML queries that use the XMLTable function, (2) the ability to enumerate basic candidate XMLTable views for the XQuery queries based on this translation, and (3) a what-if mode to estimate the cost of XQuery queries when they are translated to use candidate XMLTable views.

In this chapter, we implemented all this required functionality outside the database system, in the client side application. A full-fledged implementation of the XMLTable view advisor requires the database compiler/optimizer to be extended to translate XQuery queries into SQL/XML queries with XMLTable functions in a way similar to that described in [59]. In addition, two optimizer modes similar to the ones described in Chapter 3 (Sections 3.7.1 and 3.7.2) are required. We call these modes the Enumerate XMLTable Views and Evaluate XMLTable Views modes. In the Enumerate XMLTable Views, the optimizer would take an XQuery query, parse it to extract all the XPath expressions in this query, and finally organize them into the enumerated candidate XMLTable views. In the Evaluate XMLTable Views mode, the optimizer would create virtual XMLTable views, populate its internal structures with statistics about these views, and estimate the execution time of queries as if these views are present in the system. Furthermore, improving the query optimizer view matching and rewriting algorithms to support generating

compensating queries would allow the XMLTable View Advisor to generate more candidate XMLTable views. Generalizing the enumerated views and searching for the best possible configuration, including the necessary calls to the what-if API, would still be part of a client-side application.

4.9 Experimental Evaluation

4.9.1 Experimental Setup

Our experimental setup is the same as that used for evaluating the XML Index Advisor in Chapter 3. 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. The database system is the prototype version of DB2 V9.7 described in Section 3.8. The client application is written in Java and runs on a separate machine. When measuring the actual execution times of queries, all the measured times are on the server machine, where the database is stored, to make sure that network traffic does not affect these measurements.

We used the TPoX [70] benchmark in our experiments, and we generated the data using a scale factor of 1GB. We evaluate our advisor on the standard 10 queries that are part of the benchmark specification (we ignore one query that has multiple joins). We have made minor changes to the workload queries to account for some implementation limitations. These changes can be summarized as follows: (1) we add type casting to all XPath path expressions referenced in the queries to avoid duplicating the type checking module in our advisor application, and (2) we changed some XPath path expressions in the queries because they referenced multiple nodes in the data, which is currently not supported in our implementation. The queries used for our evaluation are given in Appendix B.

As in Chapter 3, our metric for evaluating the recommendations of the XMLTable View Advisor is *estimated workload speedup*: the estimated execution time of the workload with no XMLTable views created in the database divided by the estimated execution time of the workload with the view configuration recommended by our advisor in place. In addition, we show figures with the *actual workload speedup*: the measured execution time of the workload with no XMLTable views created in the database divided by the measured execution time of the workload with the view configuration recommended by our advisor in place. We also report the estimated execution time in optimizer units (timerons) and the actual execution time in seconds for queries in the workload.

In the following sections, we show that our XMLTable View Advisor recommends configurations of XMLTable views that effectively reduce the execution time of queries in the input workload. We focus on evaluating the effectiveness of the greedy search with heuristics. We did not evaluate the top down search algorithm in our experiments because the DAG (similar to that described in Section 3.6) that we constructed for the candidate XMLTable views enumerated for the TPoX workload had only two levels. The top down search algorithm is mainly effective when there are many DAG levels to navigate and hence minor configuration changes are made in every iteration of the algorithm. Furthermore, we demonstrate the effectiveness of recommending generalized XMLTable views that are generated by merging other views and relational indexes over XMLTable views.

4.9.2 Effectiveness of the XMLTable View Advisor Recommendations

Figures 4.3 and 4.4 show the estimated (based on query optimizer estimates) and actual (based on measured execution time) speedups for the TPoX workload. Both figures show that a maximum ratio of 1.6 (for the estimated workload execution speedup) and 1.3 (for the actual workload execution speedup) is achieved when we create the recommended views. Some queries in the workload did not benefit from

the views (speedup of 1), while others benefited significantly from the views. To show the effectiveness of our recommendations for queries that did benefit from the recommended views, we show the estimated and actual execution time of each query in Figures 4.5 and 4.6. Figures 4.5 and 4.6 show the estimated and actual execution time per query for a configuration with no views and the recommended view configurations with different disk space budgets. Queries Q1, Q2, Q7, Q8, Q9, and Q10, which range from simple navigation to join queries, benefit from the recommended XMLTable views. The actual speedup exceeds 1500 for some queries, for example Q1 and Q7. The execution times of Q1 and Q7 without the views were 21 seconds and 125 seconds, respectively. With the recommended views, these times were 0.016 seconds and 0.04 seconds. The configuration that consists of all useful views has a size of 115 MB, which also helped us to achieve an average speedup per query of 639 (the speedup of queries that did not benefit from views is 1). Even for a configuration size of 9.8 MB, the average speedup per query is 134. This shows that XMLTable views can be useful for many query types, and that our XMLTable View Advisor is quite effective at recommending these views.

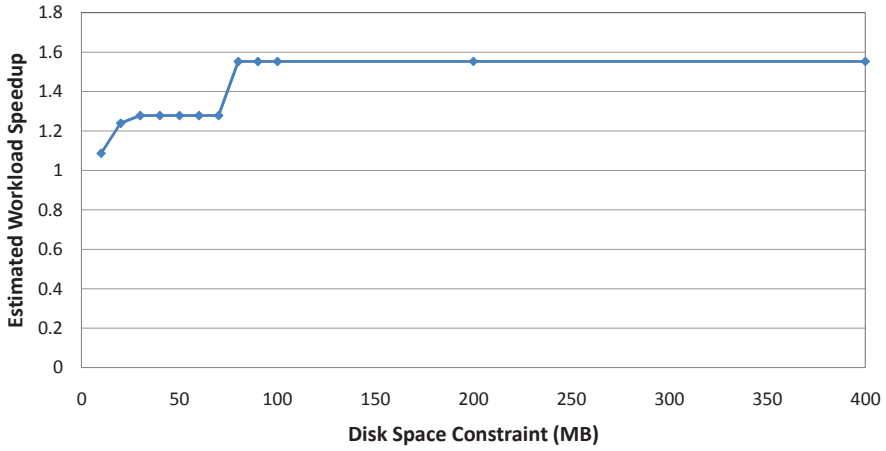


Figure 4.3: Estimated workload speedup for the recommended XMLTable views.

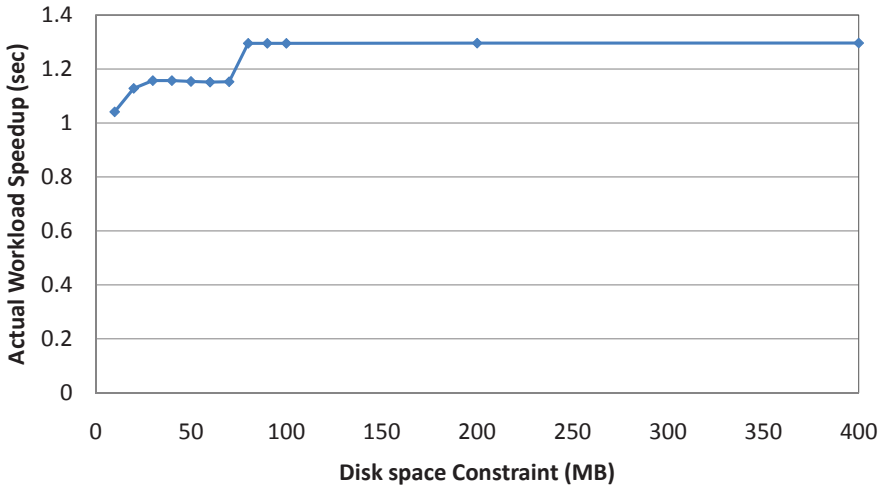


Figure 4.4: Actual workload speedup for the recommended XMLTable views.

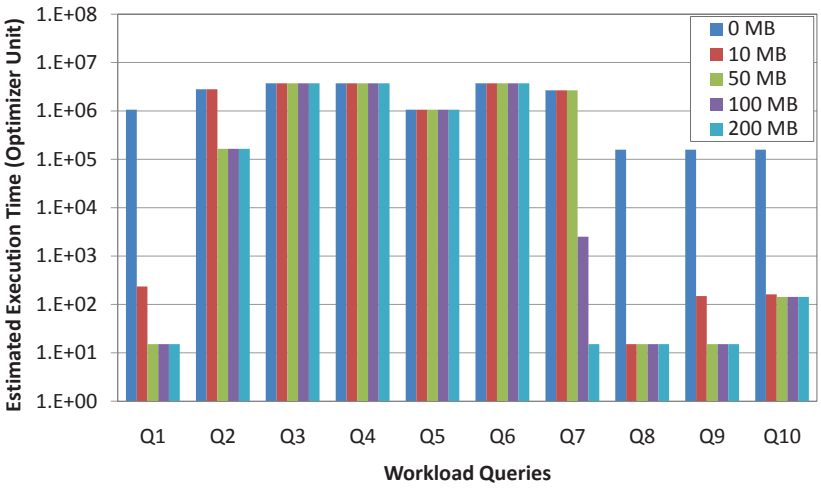


Figure 4.5: Estimated query execution time per query for the recommended XMLTable views.

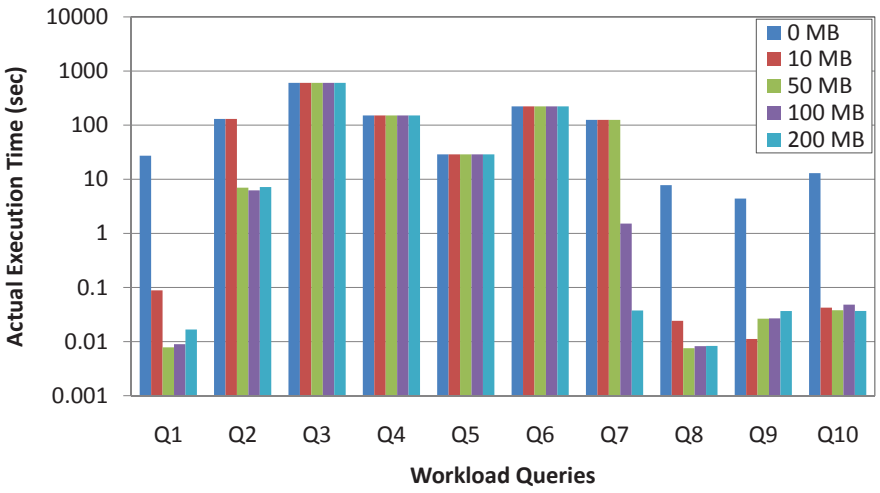


Figure 4.6: Actual query execution time per query for the recommended XMLTable views.

4.9.3 Recommending Merged XMLTable Views

To evaluate the performance of generalized views, we compare the effect of two configurations on the execution time of the TPoX workload. The first configuration (*Basic*) contains all the views enumerated for all the queries in the workload. The second configuration (*Generalized*) contains a new set of generalized views generated by merging views in the first configuration using the generalization rules presented in Section 4.4.2. In this experiment, we only use the queries that can be helped by at least one view from one of the two configurations. That is, we omit the queries that do not benefit from views (Q3-Q6, as seen in Figures 4.5 and 4.6). We measured the execution time of all queries in the workload after materializing each configuration. Table 4.1 summarizes the results. The table shows that 16% of the total size of the configuration is saved by merging views. The measured speedup with the generalized configuration is lower than the speedup with the basic configuration. However, the benefit/size ratio achieved is higher for the generalized configuration. From this we conclude that using generalized views reduces the execution cost of queries, but the benefit is lower than using basic views that only contain data referenced by the queries. However, if we also consider the reduction

in disk size needed to create the generalized configuration, the merged views are a more efficient alternative.

Configuration	Size	Measured speedup	Benefit/size ratio
Basic	58.2MB	354.6	5.3
Generalized	48.8MB	198	6.3

Table 4.1: Effect of merging views on performance.

4.9.4 Recommending Relational Indexes on XMLTable Views

In this section we investigate the benefit of building relational indexes on XMLTable views. For this experiment, we let the advisor choose a configuration consisting only of XMLTable views (with no relational indexes on them) in one case. In the other case, we let the advisor choose a configuration from a set of candidates consisting of XMLTable views and XMLTable views with relational indexes on them. The disk space budget was 2GB in both cases. Figure 4.7 shows actual execution time in both these cases, and when there are no views. We omit the TPoX queries that do not use XMLTable views from the figure. In all the remaining queries, using relational indexes over the XMLTable views reduces the execution time of the queries in the workload. The speedup achieved due to using relational indexes (compared to using views with no indexes) ranges from 1.5 to 32.5. This demonstrates the effectiveness of our approach to recommending relational indexes on the XMLTable views.

Overall, our experiments show that our XMLTable View Advisor can effectively recommend XMLTable materialized views that significantly benefit the queries that can take advantage of these views. Since the physical structure of the views is simply a relational table, view matching is simpler than full-fledged XQuery view matching. Furthermore, we can take advantage of relational query processing technology, for example by creating relational indexes on the recommended materialized views.

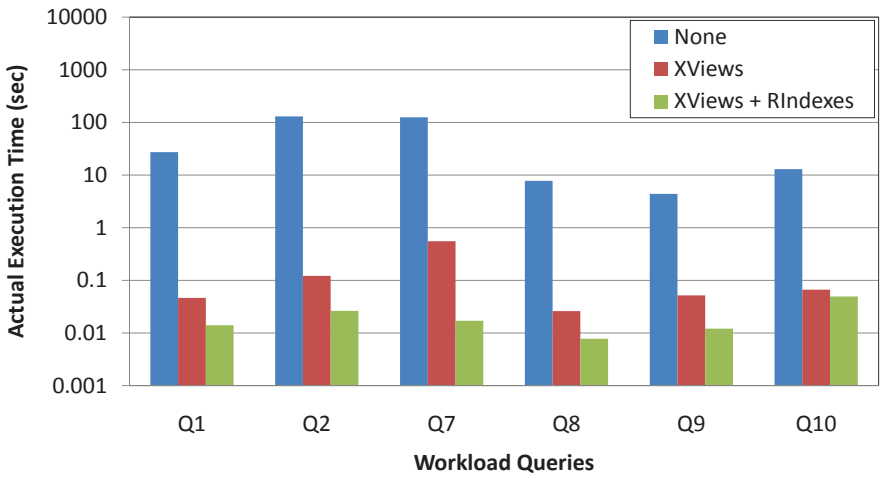


Figure 4.7: Actual query execution time per query for the recommended XMLTable views and XMLTable views with relational indexes on them.

Chapter 5

Integrated Recommendation of Indexes and Views

In the previous two chapters, we described two physical design advisors to recommend XML indexes and XMLTable views. In this chapter, we integrate these two advisors into one Integrated Index-View Advisor that recommends both XML indexes and XMLTable views for a workload of XQuery queries.

5.1 Introduction

In Chapters 3 and 4, we have shown that the recommendations of the XML Index Advisor and the XMLTable View Advisor reduce the execution time of the training workload queries as well as other queries similar to them. The magnitude of the reduction in execution time of a query due to using indexes or materialized views depends on the complexity of the query and the selectivity of its referenced path expressions. For example, a query with a low selectivity path expression which retrieves all the rows of the XML table might not benefit from an index on that predicate, but may benefit from a materialized view that contains all the elements referenced by the path expression. Similarly, if the view that is recommended for a query is as large as the data, this view will not enhance the performance of the query, but the query may still benefit from an XML index. Moreover, the execution

time of queries that are rewritten to use indexes and/or materialized views depend on the implementation of these indexes and materialized views. In this chapter, we propose an integrated version of the advisors described in the previous two chapters that recommends both XML indexes and XMLTable materialized views for a given XML database and XQuery query workload.

The Integrated Index-View Advisor recommends the set of XML indexes and XMLTable materialized views that leads to the best workload execution time when created in the database. The advisor also ensures that the recommended configuration satisfies the given disk space constraint. The integrated advisor initially enumerates all possible XML index and XMLTable view candidates for the given workload. Next, we search the space of candidate XML indexes and XMLTable materialized views together to find the best configuration to recommend. However, for a given query in the workload, the optimizer can use either XML indexes or XMLTable views in its execution plan because any XQuery query can either be rewritten to use XML indexes or be translated into an SQL/XML query to be rewritten using XMLTable views. Therefore, we revisit the search algorithms described in the previous two chapters to account for this.

The rest of the chapter is organized as follows. We motivate integrating the search for XML indexes and XMLTable views by studying their effect on query execution plans in Section 5.2. We then give a high-level description of the architecture of the Integrated Index-View Advisor in Section 5.3. Next, we describe the new search algorithm in Section 5.4. Finally, we describe our implementation in DB2 (Section 5.5) and present our experimental results (Section 5.6).

5.2 Motivation

5.2.1 Comparing Query Execution Plans that Use XML Indexes and XMLTable Materialized Views

XMLTable materialized views are considered alternative relational access paths to the XML data in the database. Yet, they can grow as large as the data, and the query execution plans that use them might not be better than the query execution plans without them. In contrast, partial XML indexes are usually smaller in size and can drastically reduce the execution time of queries. In this section, we show that XMLTable views are specifically useful for certain types of queries. We also show that XML indexes are not useful all the time. We study the usefulness of XML indexes and XMLTable views to query execution plans by comparing these different plans. We highlight three usage patterns for indexes and views: pre-navigation, joining tables, and aggregation.

Pre-navigation

Pre-navigation to the XML elements that are needed during query execution and storing them in a format that is easily accessible can save a huge amount of query execution time. The XMLTable function allows pre-navigation and stores the resulting pre-navigated values in a relational table format. By using XMLTable functions, we create new relational views of some of the fragments of the XML data that are accessed by the queries in the workload. Therefore, we can now translate complex XQuery queries into simple select statements. XML indexes are also useful in navigating to the nodes (or their values) referenced in the query. To evaluate the benefit of pre-navigation, we compare four optimizer query plan alternatives for query *Q1* (below): (1) the execution plan when no physical structures are used, (2) the execution plan when indexes are used, (3) the execution plan when XMLTable views are used, and (4) the execution plan when XMLTable views and relational indexes on them are used. In these plans we use the following abbreviations:

- **DFetch** refers to fetching a document from an XML column.
- **XSCAN** refers to scanning an XML document, which consequently means parsing or navigating an XML document depending on how the XML data are stored in the database.
- **TBFetch** refers to fetching specific rows in the table.
- **TBSCAN** refers to scanning an entire table to examine its entries.

Q1: Return order IDs which have their status `OrdStatus` equal to P

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

Figure 5.1 shows four possible query execution plans for *Q1*[1]. Figure 5.1(a) illustrates a typical query execution plan when no physical structures are used. In this plan, all the documents in the table are read and scanned to find the qualifying predicate(s) and the return value(s). The total cost of this plan essentially equals the cost of navigating all documents in the table. To reduce the execution cost, there are three alternatives:

1. We can use an XML index (for example, an index that includes the XML nodes that are reachable by the XPath expression `/Order/OrdStatus`) to select the XML subtrees rooted by nodes that satisfy the predicate(s) in the query. We would then navigate to these subtrees to find the return value(s). In this execution plan, the execution cost is equal to the sum of the index navigation cost and the navigation cost of the selected documents (Figure 5.1(b)).
2. We can use an XMLTABLE view such as *V1* (below). The execution plan for the rewritten query that uses this view (*RQ1* below) includes scanning all the rows of the view to find qualifying tuples. The cost of this execution plan is equal to the cost of scanning the entire view (Figure 5.1(c)).

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

3. We can use an XMLTABLE view and a relational index on the columns that represent all predicates in the original XQuery (for example, an index on the column *cx1* in view *V1*). In this case, the cost of executing the plan is equal to the sum of the index navigation and fetching the qualified tuples (Figure 5.1(d)).

V1: Create a view on the `ORDER` table that contains the `ID` and `OrdStatus` values for all the order documents that are stored in the table

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

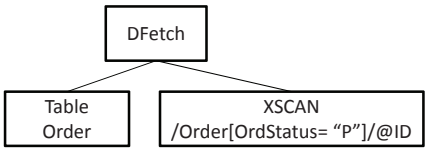
RQ1: A rewritten version of query *Q1* to use view *V1*

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

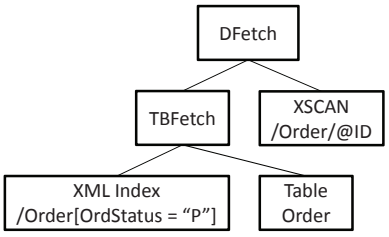
Depending on the structure of the XML documents and the selectivity of the predicates in the query, various situations will lead to different possible plans having the lowest cost.

Joining Tables

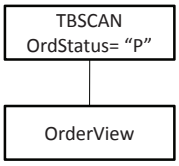
Joining tables is a common and important operation in XQuery queries. The execution cost of XQuery queries with table joins can be reduced by pre-navigating the values to be joined, storing them in relational tables, and then joining these relational tables. For example, we show in Figure 5.2 the execution plans for Query *Q2* (shown below), which has a join between tables `ORDER` and `SECURITY`. Figures 5.2(a) and 5.2(b) show the execution plans for query *Q2* using XML indexes and XMLTABLE views, respectively. The number of elements in the join operator's



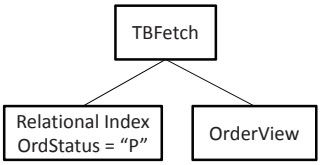
(a) Query execution plan with no physical structures.



(b) Query execution plan with XML indexes.



(c) Query execution plan with XMLTABLE views.



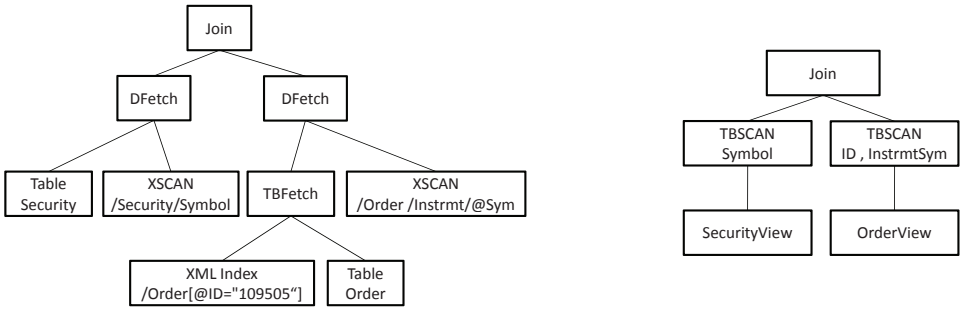
(d) Query execution plan with XMLTABLE views and relational indexes on them.

Figure 5.1: Query execution plans for query *Q1*.

two inputs is the same in both execution plans. The total execution cost can be lower in the execution plan with XMLTABLE views because of the following: (1) the relational optimizer can use a larger variety of join operators, hash joins for instance, and (2) the table scan of the XMLTable materialized view is cheaper than scanning a table with XML documents stored in one of its columns. In the latter case, it is required to further scan (parse or navigate) the XML documents.

Q2: Return current open price of a particular order

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



(a) Query execution plan with XML indexes. (b) Query execution plan with XMLTABLE views.

Figure 5.2: Query execution plans for query *Q2*.

Aggregation

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

5.2.2 Comparing Query Execution Times of Plans that Use XML Indexes and XMLTable Materialized Views

Having qualitatively described and contrasted the benefit of XML indexes and XMLTable views, we now compare the execution time of queries when creating the recommendations of the XML Index Advisor and the XMLTable View Advisor for large space budget (2 GB). Figures 5.3 and 5.4 show the estimated execution times and actual execution times, respectively, of queries in the TPoX workloads for the following three cases: (1) no physical structures are used, (2) XML indexes recommended by the XML Index Advisor are created, and (3) XMLTABLE views recommended by the XMLTable View Advisor are created. We note that we could not use views with four TPoX queries, Q3-Q6 (Section 4.8). We observe that the execution times of four out of the remaining six queries of the TPoX workload when rewritten to use XMLTable views are less than the execution times of these

queries when rewritten to use XML indexes. However, the benefit/size ratios of the XMLTable views used in rewriting these queries are less than the benefit/size ratios of indexes used for rewriting the same query because views usually have larger sizes compared to indexes. Table 5.1 shows the total sizes in MB of both the XML indexes and the XMLTable materialized views recommended for every query in the workload.

Figure 5.3: Estimated execution time per query for advisor recommendations.

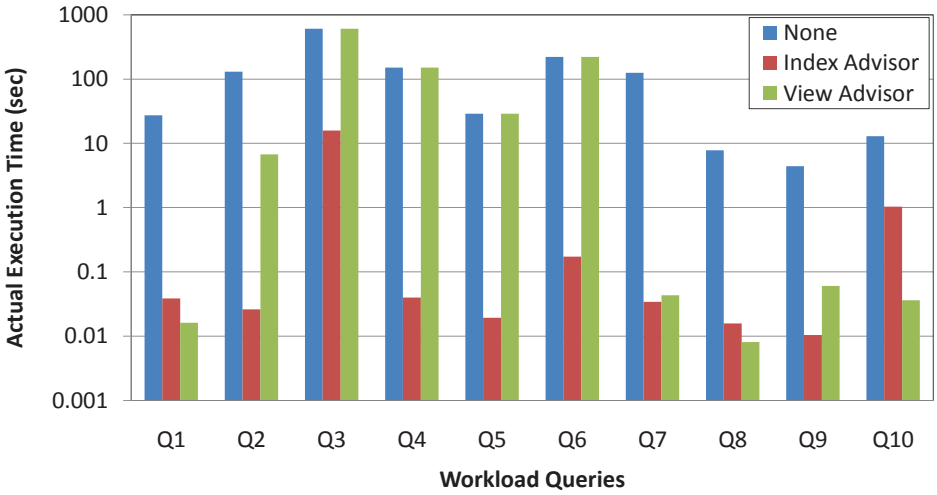


Figure 5.4: Actual execution time per query for advisor recommendations.

Queries	XML indexes	XMLTable views
Q1	2.7	2.9
Q2	29.5	14
Q3	30.6	None
Q4	23.3	None
Q5	2.7	None
Q6	37.2	None
Q7	14.4	35.8
Q8	0.8	1.3
Q9	0.8	2.1
Q10	3	2.1

Table 5.1: Sizes in MB of recommended XML indexes and XMLTable materialized views for TPoX queries.

5.2.3 Summary

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

5.3 Design of the Integrated Index-View Advisor

To build an integrated Index-View Advisor, we incorporate the two advisors presented in Chapters 3 and 4. Figure 5.5 shows the architecture of our integrated advisor. Since the candidate enumeration processes for XML indexes and XMLTable materialized views are different, we enumerate and generalize candidates of each type separately using the candidate enumeration and generalization algorithms described in Chapters 3 and 4. This results in candidates of three types: XML indexes, XMLTable materialized views, and XMLTable materialized views with relational indexes on them. We combine all these candidates into one pool of candidates, and we search for the best configuration among all the candidates. The search algorithm is different from the search algorithms in Chapters 3 and 4 because the space of candidates contains different types of physical structures, which introduce new types of interactions. In the next section, we generalize the search algorithms described in Chapters 3 and 4 to an algorithm that considers different types of interactions between different physical structures.

5.4 Search Algorithm

The search algorithms that we have proposed in Chapters 3 and 4 take into account two types of interactions between candidates: (1) interaction between candidates that can be used to rewrite the same query, and (2) candidates where one is a general form of the other. While the former type of interaction affects the benefit of candidates due to the existence of other candidates, the latter type of interaction poses a restriction that at most one candidate is to be chosen. When searching the combined space of XML indexes and XMLTable views, we also consider that a query can either be rewritten to use XML indexes or XMLTable views, but not both, because of their different rewriting algorithms. The notion of candidate coverage is not valid any more, as there is no clear relation between the XML indexes and the XMLTable views that can be used for the same query, and we

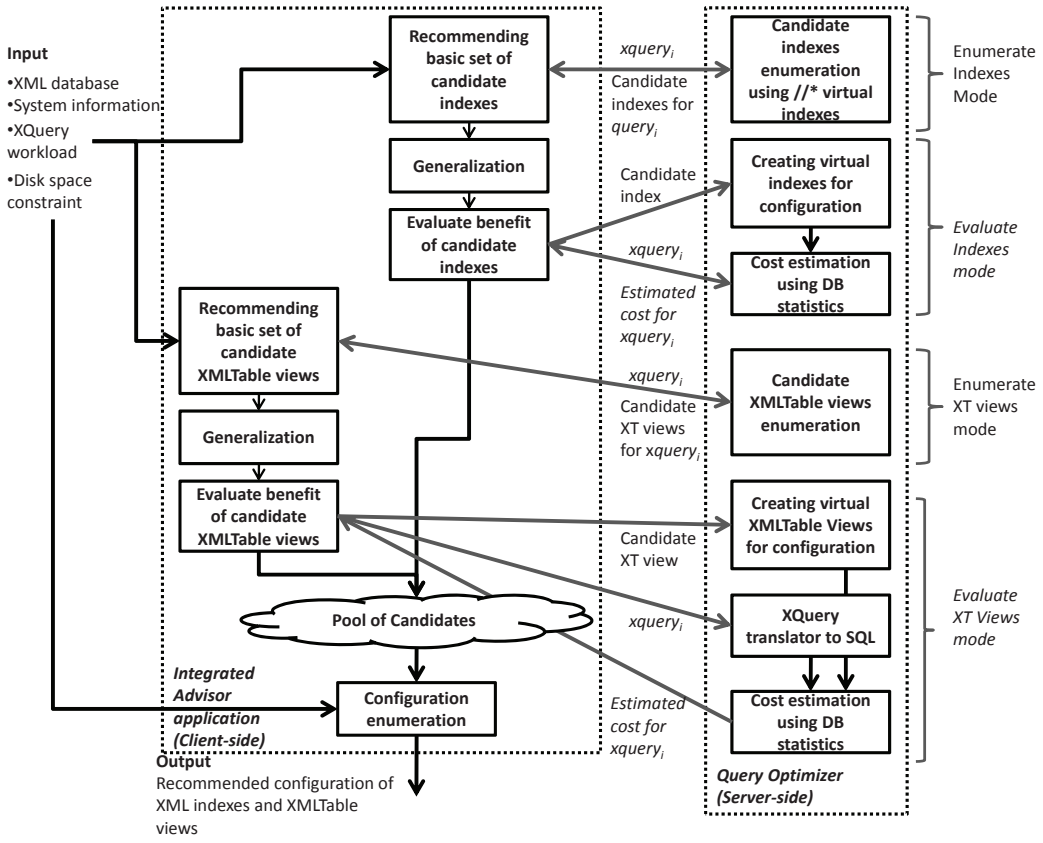


Figure 5.5: The Integrated Index-View Advisor architecture.

also want to consider using either of them for each query. We choose to define the coverage in the integrated search algorithm based on query coverage, and we introduce new rules to handle special cases. We treat the case of an XMLTable view with a relational index on it as a separate structure from its corresponding XMLTable view. Hence, we have three types of candidates in the search space: (1) XML indexes, (2) XMLTable views, and (3) XMLTable views with relational indexes on their columns.

The high level outline of the search algorithm is similar to the one we use to search the space of views (Section 4.6), with different rules for the various types of candidates. Algorithm 26 presents the integrated search algorithm. The first step of the search algorithm is to sort all of the physical structures according to their benefit/size ratio. We then iteratively consider candidate physical structures:

Algorithm 26 searchIntegrated(*candidates*, *diskConstraint*)

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

XML indexes (XINDEX), XMLTable views (XVIEW), and XMLTable views with relational indexes on them (XVIEW_RINDEX) and try to add them to the set of recommended structures (*recommended*). In every iteration, if the recommended set of candidates is empty or the candidate that we are considering in this iteration (*bestCand*) adds new coverage (i.e., it helps a query in the workload that is not yet helped by any of the structures already selected by the advisor), we add *bestCand* to our set of recommended physical structures if enough disk space is available. Otherwise, if there is overlap between the queries that are already covered by physical structures in the *recommended* configuration and the candidate's (*bestCand*) coverage, we apply heuristic rules to decide whether to add *bestCand* to our set of recommended physical structures or not. First, we find the set of physical structures in the *recommended* configuration that help some or all the queries that are covered by the candidate physical structure *bestCand*. We call this set of candidates the *overlapConfig*. We then apply the following rules depending on the type of *bestCand*:

1. *bestCand* **is an XML index:** We build an alternate configuration (*replaceConfig*) consisting of the set of physical structures that belong to the *overlapConfig* set and that satisfy one of these conditions:

   (a) The physical structure is an XML index and is a general form of *bestCand*. In this case, the physical structure *cand* can replace *bestCand* in its query execution plans.

   (b) The physical structure is an XMLTable view or XMLTable view with a relational index on it. In this case, choosing an XMLTable view to answer a query in the workload means that we cannot use XML indexes for rewriting it, because the query rewriting algorithm can use either XML indexes or XMLTable views to rewrite a given query, but not both.

2. *bestCand* **is an XMLTable view:** We build an alternate configuration (*replaceConfig*) as the set of physical structures that belong to the *overlapConfig* and that satisfy one of these conditions:

(a) The physical structure is an XML index. Hence, we either choose the XML index that is already selected or the new XMLTable view that we are currently considering.

(b) The physical structure is an XMLTable view, and it is a general form of *bestCand*. We add this candidate to *replaceConfig* because it can replace *bestCand* in rewriting the queries that *bestCand* can help.

(c) The physical structure is an XMLTable view with a relational index on it. We add this candidate to *replaceConfig* if its view *cand.view* is the same as *bestCand* or a general form of it.

3. *bestCand* **is an XMLTable view with a relational index on it:** We build an alternate configuration (*replaceConfig*) as the set of physical structures that belong to the *overlapConfig* and that satisfy one of these conditions:

(a) The physical structure is an XML index. Hence, we either choose the XML index that is already selected or the new XMLTable view with a relational index on it that we are currently considering.

(b) The physical structure is an XMLTable view. Whether *cand* is the same as *bestCand.view* or is a general form of it, we add *cand* to *replaceConfig*.

(c) The physical structure is an XMLTable view with a relational index on it. We add this candidate to *replaceConfig* if its view *cand.view* is a general form of *bestCand.view*.

The next step in the algorithm is to check the alternate configuration *replaceConfig*. If it is empty, this means that *bestCand* can be used together with already selected physical structures to answer queries in the workload and that we can safely add *bestCand* to the *recommended* configuration if there is enough disk space. Otherwise, we check the following two configurations: (1) $bestCand \cup (recommended - replaceConfig)$: the configuration that includes *bestCand* in addition to the structures that we have already selected after removing the ones in *replaceConfig* from it, and (2) *recommended*. If the new configuration

($bestCand \cup (recommended - replaceConfig)$) has a higher benefit and its size does not exceed the disk space constraint, we make it the *recommended* configuration.

In Algorithm 26, we use the following helper functions:

- *isGeneral*(*cand1*, *cand2*): returns *true* if *cand1* is of the same type as *cand2* and is a general form of *cand2*.
- *addCandIfSpaceAvl*(*cand*,*config*): adds the candidate physical structure *cand* to the configuration of physical structures *config* if the size of the new configuration is not larger than the disk space constraint input to the advisor.
- *replaceCandIfSpaceAvl*(*cand*,*replaceConfig*,*config*): it replaces the set of structures in *replaceConfig* from the configuration *config* with the candidate structure *cand* if the new configuration's size does not exceed the disk space constraint. The new configuration is: $cand \cup (config - replaceConfig)$.

5.5 Implementation in DB2

The Integrated Index-View Advisor does not require any new server-side infrastructure. Instead, we rely on the implementation of the XML Index Advisor and the XMLTable View Advisor described in Chapters 3 and 4, respectively. We implemented the search algorithm described in the previous section in a Java client-side application.

5.6 Experimental Evaluation

5.6.1 Experimental Setup

We use the same experimental setup as the previous chapters. We have conducted our experiments on a Dell PowerEdge 2850 server with two Intel Xeon 2.8GHz CPUs (with hyperthreading) and 4GB of memory running SuSE Linux 10. The database is stored on a 146GB 10K RPM SCSI drive. We use IBM DB2 9.7 as our database system.

We again use the TPoX [70] benchmark in our experiments, and we generate the data using a scale factor of 1GB. To evaluate the results from the Integrated Index-View Advisor and compare it to the results from both the XML Index Advisor and XMLTable View Advisor, we used the same workload of the 10 TPoX queries that we used in evaluating the XMLTable View Advisor (Section 4.9.1).

Our metrics for evaluating the recommendations of the Integrated Index-View Advisor and for comparing its results with the recommendations of the XML Index Advisor and the XMLTable View Advisor are the estimated query execution time in optimizer units (timerons) and actual query execution time in seconds. In contrast to our approach in Chapters 3 and 4 in relying on workload speedups for evaluating the benefit of our advisors, we present only query execution times in this chapter because they give a better presentation of the differences between the advisor recommendations.

5.6.2 Effectiveness of the Integrated Index-View Advisor Recommendations

In our first experiment, we show that the Integrated Index-View advisor recommends configurations of XML indexes and XMLTable views that are useful to workload queries. Figures 5.6 and 5.7 show the estimated and actual execution time per query for a configuration with no physical structures and configurations of XML indexes and XMLTable views that are recommended by the Integrated Index-View Advisor for various disk space constraints (10MB, 50MB, 100MB, and 200MB). The larger the disk size used to build physical structures, the better the workload execution time that can be achieved. For the best execution time that can be reached, a configuration of XML indexes and XMLTable views of size 158MB is recommended by the advisor. The corresponding workload estimated speedup is 30174.

In our second experiment, we compare the recommendations of the XML Index Advisor, the XMLTable View Advisor, and the Integrated Index-View Advisor.

Figure 5.6: Estimated execution time per query for the Integrated Index-View Advisor recommendations.

Figures 5.8 and 5.9 are an updated version of Figures 5.3 and 5.4 discussed earlier in this chapter, respectively showing estimated and actual query execution times, after adding one more column to represent the estimated and actual execution times of the queries in the workload when the recommendations of the Integrated Index-View Advisors are materialized in the database. We observe that the Integrated Index-View Advisor always chooses XML indexes for all queries in this experiment. For some queries in this workload, XML indexes have higher benefit than XMLTable views so choosing indexes is simple to justify. However, there are queries for which the advisor chooses an index even though it has lower benefit than the candidate view for this query because the index has a much smaller size and hence its benefit/size ratio is higher. Another reason that the advisor chooses an index for some queries in this workload when the candidate materialized view is more beneficial for the query is that the index can be useful to other queries in the workload while the materialized view is only useful to one query. Hence, choosing the index is justified because its benefit to the entire workload is higher than the benefit of selecting the candidate materialized views for each of the queries helped by this index. For example, for query Q1 in the TPoX workload, the Inte-

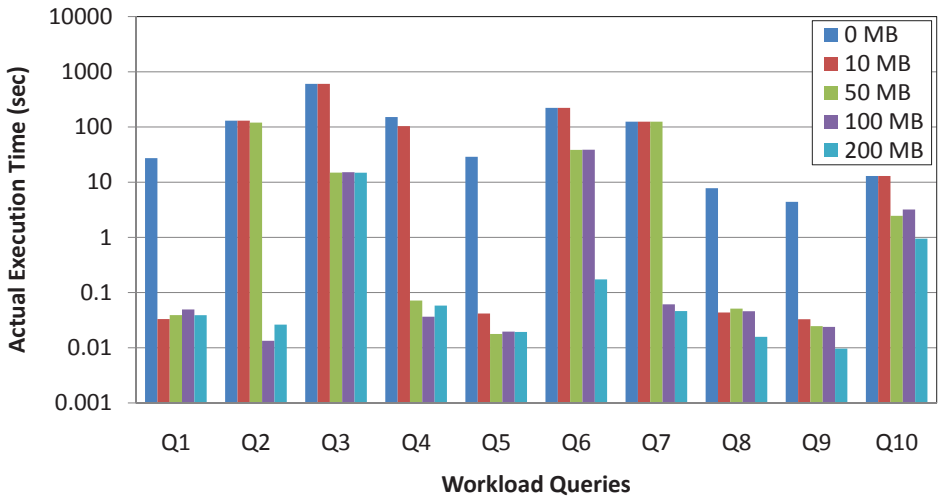


Figure 5.7: Actual execution time per query for the Integrated Index-View Advisor recommendations.

grated Index-View Advisor recommends an XML index for it although selecting an XMLTable view is expected to result in a lower execution time. The explanation of this behavior is as follows: Query Q1 benefits from building an XML index I1, which is also useful for queries Q4 and Q5. Query Q1 also benefits from building an XMLTable view V1. The estimated benefit of the XML index I1 when calculated for the entire workload is higher than the estimated benefit of the XMLTable view V1. Hence, the candidate XML index is chosen by the search algorithm.

To eliminate the effect of this type of interaction, we compare the recommendations of the three advisors when the input workload is composed of queries Q1, Q7–Q10. The results are shown in Figure 5.10. The figure shows that the Integrated Index-View Advisor selects the candidate structures that lower the execution time of individual queries when these structures also lower the execution time of the entire workload. In this figure we can see that the advisor sometimes recommends indexes and sometimes recommends views.

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

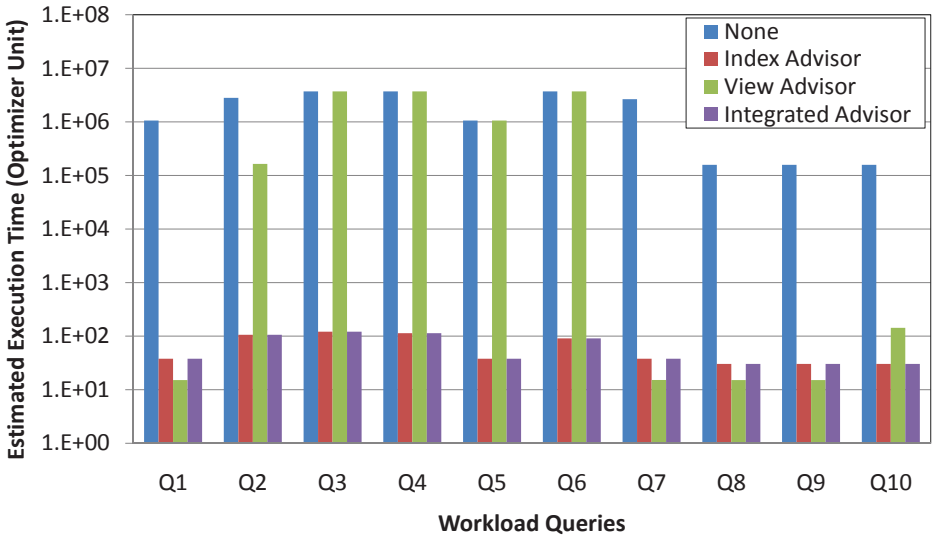


Figure 5.8: Estimated execution time per query for advisor recommendations. Disk budget 400MB.

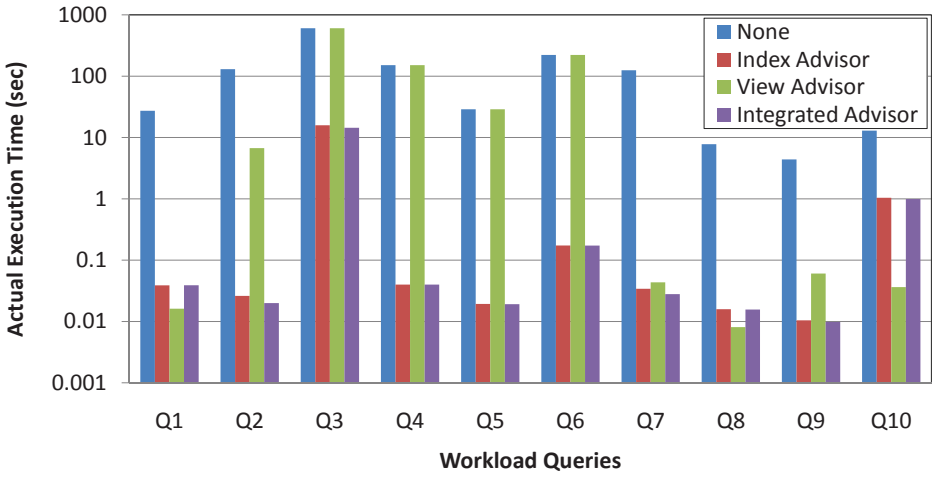


Figure 5.9: Actual execution time per query for advisor recommendations. Disk budget 400MB.

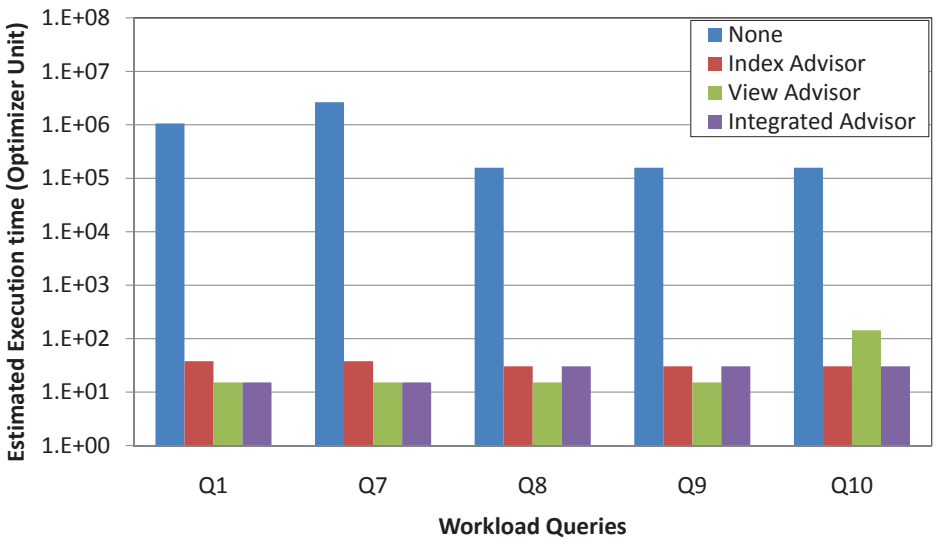


Figure 5.10: Estimated execution time per query for advisor recommendations. Disk budget 400MB.

Chapter 6

Conclusions

This thesis focuses on developing automatic physical design tools for XML databases. In this chapter, we present our conclusions and suggestions for future work.

6.1 Summary of Contributions

The main contributions presented in this thesis are as follows: (1) Chapter 3 presented techniques and algorithms to automatically recommend XML indexes, (2) We described the XMLTable materialized views and their automatic recommendation in Chapter 4, and (3) Chapter 5 presented an integrated advisor that recommends XML indexes and XMLTable materialized views.

The XML Index Advisor presented in Chapter 3 has the following key features:

1. The advisor is tightly coupled with the query optimizer, using it for both enumerating and evaluating indexes. To leverage the query optimizer for enumerating candidate indexes and evaluating their benefit to queries, we use the notion of virtual indexes in two query optimizer extensions. The virtual universal index, which is a new notion introduced in this thesis, hypothetically indexes all elements in the document and hence can be matched with any XPath pattern that appears in a query and can be answered using an index. This allows us to enumerate these matched patterns as candidate index patterns. To estimate the benefit of an index, we create this index as

a virtual index in the database in a way that enables the query optimizer to use it in its query plans.

2. 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.

3. We employ techniques to minimize the number of optimizer calls that the XML Index Advisor makes.

4. We have developed a method for estimating the penalty imposed by the indexes on update, delete, and insert statements. We take this penalty into account when estimating the benefit of an index for a workload that has maintenance statements in addition to queries.

In Chapter 4, we presented an XMLTable View Advisor. This is a new approach for building relational materialized views for XQuery workloads. The recommended relational views are in the form of XMLTable materialized views. These views can help the query execution performance by pre-navigating to queried values that appear in the data. In addition, XMLTable view matching is based on relational view matching and XPath matching, and hence we can leverage the existing infrastructure of many database system query optimizers.

In Chapter 5, we analyzed the different benefits that XML indexes and XMLTable views can provide to various types of XQuery queries. We concluded that both XML indexes and XMLTable views are useful and that they benefit different query types with different degrees. However, we cannot rewrite one query to use both XML indexes and XMLTable views, and we need to choose between them for any query in the workload. We presented an Integrated Index-View Advisor that searches for the best physical design for a workload in a pool of candidate physical structures that contains XML indexes and XMLTable views.

We have implemented our proposed advisors in a prototype version of IBM DB2. Our experiments with this implementation show that our advisors can effectively recommend physical designs that result in significant speedups for workload queries.

6.2 Future Work

The work presented in this thesis can be extended in the following directions:

- **Studying other optimization constraints.** In the work that studies building relational design advisors [4, 23, 87, 92], the main constraint in the optimization problem when searching for the best configuration of physical structures is usually the disk space budget. In this thesis, we used the same constraint when recommending the best configuration of physical structures. The disk size allocated to build physical structures indirectly affects other constraints such as maintenance overhead because large physical structures will necessarily impose a high penalty to maintain them. Maintenance can also be reflected directly as a constraint in the optimization problem by constraining the number of physical structures that can be recommended by the advisor [23]. Alternatively, we can include a threshold for the overhead imposed by these structures because of maintenance statements. If the database is rarely maintained, then we do not need to limit the physical structures recommended by the advisor. Studying other constraints for the search problem of finding the best physical design is a possible direction for future research.

- **Penalty of maintaining XMLTable views**. We have discussed our approach for estimating the penalty incurred by maintenance statements when indexes exist in the database in Section 3.5.2. We left for future work estimating the penalty incurred by maintenance statements because of materialized XMLTable views and taking these estimates into account when estimating the benefit of views.

- **Removing the limitations on XMLTable view recommendation.** There are several limitations in our current approach to recommending XMLTable views, such as the lack of support for compensating queries, or the limitations on the XQuery format that can be handled. Removing these limitations is another interesting direction for future work.

- **Automatically partitioning XML data.** We focused in this thesis on automatically recommending physical design structures for XML databases. Query execution can also be improved by rearranging the way that the XML data is stored (partitioning). Two notable characteristics of XML databases are: (1) the schema is heterogeneous, and (2) XML query processing is based on path navigation and result construction. Commercial database systems allow XML documents with different schemas to be stored in the same XML column, which leads to the heterogeneity. The XML query languages supported by these systems rely on navigation. Therefore, partitioning is a good candidate for increasing database manageability and improving performance [5]. All previous research has focused on partitioning XML data trees stored in one XML document [9, 54, 55, 83]. A novel research direction is partitioning a table with columns of XML type. This involves finding partitioning keys from the XML data to cluster the documents that are stored separately. These concepts can be applied to create multiple partitions on one machine, or on several machines in a shared nothing environment.

- **Recommending XQuery views.** In Chapter 4, we focused on recommending XMLTable materialized views for an XML database and an XQuery workload. XQuery query containment and rewriting is studied in [7, 29, 72]. This opens the door to recommending XQuery materialized views for XQuery workloads. To recommend XQuery views, we need to study enumerating candidate views for an XQuery workload and evaluating their benefit to queries. We also need to ensure that the views that we recommend can be matched by the optimizer at query execution time.

- **Online Physical Design.** This thesis focused on selecting the best physical structures for a database and input workload in an offline manner. Changing the physical design decisions online to adapt to the changes in the workload by modifying the physical design of the database is studied in [20, 77, 79] for relational databases. Studying online physical design for XML databases is an open problem and an interesting direction for future work.

Bibliography

[1] XMLTABLE overview, 2006. Available at: `http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.apdv.embed.doc/doc/c0023903.htm`.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

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

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2000.

[5] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2004.

[6] A. Arion, V. Benzaken, and I. Manolescu. XML access modules: Towards physical data independence in XML databases. In *Proc. Int. Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 2005.

[7] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2007.

[8] A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: choosing the right storage for your XML application. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2005.

[9] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. In *Proc. Int. Conf. on World Wide Web (WWW)*, 2006.

[10] A. Balmin, K. S. Beyer, F. Özcan, and M. Nicola. On the path to efficient XML queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2006.

[11] 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.

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

[13] A. Berglund, S. B. D. Chamberlin, M. F. Fernndez, M. Kay, J. Robie, and J. Simon. XML path language (XPath) 2.0. W3C Candidate Recommendation, 2007. Available at: `http://www.w3.org/TR/xpath20/`.

[14] K. Beyer, R. Cochrane, M. Hvizdos, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, R. Lyle, M. Nicola, F. Özcan, 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.

[15] 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 *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2005.

[16] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C Candidate Recommendation, 2006. Available at: `http://www.w3.org/TR/xquery`.

[17] P. Bohannon, J. Freire, J. R. Haritsa, and M. Ramanath. LegoDB: Customizing relational storage for XML documents. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2002.

[18] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2002.

[19] P. Bohannon, S. Ganguly, H. F. Korth, P. P. S. Narayan, and P. Shenoy. Optimizing view queries in ROLEX to support navigable result trees. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2002.

[20] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2007.

[21] 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), 2005.

[22] S. Chaudhuri and V. Narasayya. AutoAdmin 'what-if' index analysis utility. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 1998.

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

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

[25] J. Clark and S. DeRose. XML path language (XPath). W3C Candidate Recommendation, 1999. Available at: `http://www.w3.org/TR/xpath`.

[26] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2003.

[27] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 1999.

[28] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2003.

[29] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of nested XML queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2004.

[30] I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. Beyer, and C. Zuzarte. XML index recommendation with tight optimizer coupling. Technical Report CS-2007-22, University of Waterloo, 2007.

[31] I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. Beyer, and C. Zuzarte. An XML index advisor for DB2. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2008.

[32] I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. Beyer, and C. Zuzarte. XML index recommendation with tight optimizer coupling. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2008.

[33] I. Elghandour, A. Aboulnaga, D. C. Zilio, F. Chiang, A. Balmin, K. Beyer, C. Zuzarte, W. Krause, and P. Mierzejewski. IBM XML Index Advisor for DB2 for Linux, UNIX, and Windows. `http://www.alphaworks.ibm.com/tech/xmlindexadvisor`.

[34] I. Elghandour, A. Aboulnaga, D. C. Zilio, and C. Zuzarte. Recommending XMLTable views for XQuery workloads. In *Proc. Int. Symp. on Database and XML Technologies (XSym)*, 2009.

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

[36] M. F. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: trading between relations and XML. *Computer Networks*, 33(1-6), 2000.

[37] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *The VLDB Journal*, 11(4), 2002.

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

[39] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[40] P. Godfrey, J. Gryz, A. Hoppe, W. Ma, and C. Zuzarte. Query rewrites with views for XML in DB2. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2009.

[41] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1997.

[42] 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.

[43] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.

[44] A. Halverson, V. Josifovski, G. M. Lohman, H. Pirahesh, and M. Mörschel. ROX: Relational over XML. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2004.

[45] 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.

[46] 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.

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

[48] IBM Corp. Comparing XML and relational storage: A best practices guide, 2005. Available at: `ftp://ftp.software.ibm.com/software/data/pubs/papers/GC34-2497.pdf`.

[49] IBM Corp. *IBM DB2 Database for Linux, UNIX, and Windows Information Center*, 2006. Available at: `http://publib.boulder.ibm.com/infocenter/db2luw/v9/`.

[50] V. Josifovski, S. Massmann, and F. Naumann. Super-Fast XML wrapper generation in DB2: A demonstration. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2003.

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

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

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

[54] J. Kim and H.-J. Kim. A partition index for XML and semi-structured data. *Data & Knowledge Engineering*, 51(3), 2004.

[55] P. Kling, M. T. Özsu, and K. Daudjee. Generating efficient execution plans for vertically partitioned XML databases. *PVLDB*, 4(1), 2010.

[56] G. Lapis. XML and relational storage – Are they mutually exclusive? In *Proc. Conf. on XML, the Web and beyond (XTech)*, 2005.

[57] W. Lehner, B. Cochrane, H. Pirahesh, and M. Zaharioudakis. fAST refresh using mass query optimization. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2001.

[58] Z. H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery processing in Oracle XMLDB. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2005.

[59] Z. H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery processing in Oracle XMLDB. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2005.

[60] Z. H. Liu, M. Krishnaprasad, H. J. Chang, and V. Arora. XMLTable index an efficient way of indexing and querying XML property data. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, 2007.

[61] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2005.

[62] M. Mani and D. Lee. XML to relational conversion using theory of regular tree grammars. In *Proc. Int. Workshop on Efficiency and Effectiveness of XML Tools and Techniques*, 2002.

[63] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2001.

[64] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. *SIGMOD Rec.*, 26(3), 1997.

[65] W. Meier. eXist: An open source native XML database. In *Web, Web-Services, and Database Systems*, 2002.

[66] J. Melton and S. Muralidhar. XML syntax for XQuery 1.0 (XQueryX). W3C Recommendation, 2007. Available at: `http://www.w3.org/TR/xqueryx`.

[67] J. Melton (ed.). SQL standard - part 14: XML-related specifications (SQL/XML). ISO Standard, 2006.

[68] T. Milo, , and D. Suciu. Index structures for path expressions. In *Proc. Int. Conf. on Database Theory (ICDT)*, 1999.

[69] M. M. Moro, L. Lim, and Y.-C. Chang. Schema advisor for hybrid relational-XML DBMS. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2007.

[70] M. Nicola, I. Kogan, and B. Schiefer. An XML transaction processing benchmark. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2007.

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

[72] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2006.

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

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

[75] 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.

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

[77] K.-U. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous query-driven index tuning. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2003.

[78] 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.

[79] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous on-line tuning. In *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2006.

[80] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2000.

[81] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[82] D. M. Srinivasa and R. Rajendra. XQuery normalizer and static analyzer. IBM alphaWorks, 2004. Available at: `http://www.alphaworks.ibm.com/tech/xqnsta`.

[83] N. Tang, G. Wang, J. X. Yu, K.-F. Wong, and G. Yu. WIN: an efficient data placement strategy for parallel XML databases. *IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, 2005.

[84] 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 *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD)*, 2002.

[85] TPC-D: TPC Benchmark D (Decision Support). Available at: `http://www.tpc.org/tpcd/`.

[86] Transaction processing over XML (TPoX), XML database benchmark, 2007. Available at: `https://sourceforge.net/projects/tpox/`.

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

[88] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2005.

[89] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transaction on Internet Technology (TOIT)*, 1(1), 2001.

[90] F. Zemke, M. Rys, K. Kulkarni, J.-E. Michels, B. Reinwald, F. Özcan, Z. Liu, I. Davis, and K. Hare. XMLTABLE. Available at: `http://www.wiscorp.com/H2-2004-039-xmltable.pdf`, 2004.

[91] H. Zhang and F. W. Tompa. Querying XML documents by dynamic shredding. In *Proc. ACM Symp. on Document Engineering*, 2004.

[92] 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 *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 2004.

Appendices

Appendix A

Update Penalty Estimation Verification

In Section 3.5.2, we described the method we use to estimate the penalty of the workloads' update statements incurred because of indexes created in the database. In this appendix, we describe an experiment that we have designed to measure how accurately our maintenance (insert, delete, or update) penalty estimation method reflects the actual maintenance cost of the indexes. Our procedure is based on confirming that the estimated index maintenance cost corresponds to the actual time needed to perform that task.

Our experimental procedure computes an estimated penalty for one database maintenance operation based not on the estimation formulas of Section 3.5.2, but rather on fitting a linear equation to the measured penalty. We call this estimated penalty that we compute P. We use the following variables to calculate P:

- E_{QN}: The estimated query execution time when no indexes are used in the query execution plan.
- E_{QI}: The estimated query execution time when indexes are used in the query execution plan.
- A_{QN}: The measured query execution time when no indexes are used in the query execution plan.

- A_{QI}: The measured query execution time when indexes are used in the query execution plan.

- P_E: The estimated penalty for one database maintenance computed using the formula in Section 3.5.2. Recall that this penalty P_E computed for maintenance statement s is defined as:

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

- P_A: The measured penalty for one database maintenance (insert, delete, or update).

- A_{UN}: The measured maintenance statement execution time when no indexes are created in the database.

- A_{UI}: The measured maintenance statement execution time when indexes are created in the database.

- B_{EST}: The estimated benefit of an index or a set of indexes.

- B_{ACT}: The measured benefit of an index or a set of indexes.

In this experiment, we measure the discrepancy between P and P_E. Based on the average P and P_E, we compute a multiplicative factor representing an approximation of P/P_E. The index advisor multiplies P_E by this factor as a way of calibrating its estimates. For maximum accuracy, this process is repeated for every database and workload. However, it is possible to obtain reasonably accurate advisor recommendations based on one value of this calibration factor for all databases and workloads.

The experiment we performed is as follows. First, we run the XML Index Advisor with the TPoX workload and one maintenance statement. For this experiment our maintenance statements are always insert statements in the `ORDER` table. We learn from this advisor invocation the recommended indexes, the estimated cost of one maintenance statement, and the estimated costs of the queries and the maintenance statement before and after creating the indexes. We want to map the

estimated cost of the maintenance statement to the actual cost. Therefore, we want to find a reference point where both the actual and estimated penalty are equal. In principle, it should be possible to use any estimated or actual cost as this reference point, since the estimated and actual penalties should always be equal. However, for this calibration experiment we use the point when the cost of updating the index is equal to the reduction in queries execution time due to using indexes. Hence, we increase the number of updates (N) until the measured benefit of using indexes reaches zero. This point is used to compute P by equating the measured benefit to the estimated benefit for the same N.

We calculate the estimated benefit of indexes for a query workload (B_{EST}) as the difference between the estimated query execution time of queries in the workload when no indexes are created and the estimated query execution time of queries in the workload when indexes are created, reduced by the estimated penalty of update statements in the workload. Similarly, the measured benefit of indexes for a query workload (B_{ACT}) is the difference between the measured query execution time of queries in the workload when no indexes are created and the measured query execution time of queries in the workload when indexes are created, reduced by the measured penalty of maintenance statements in the workload. These two definitions are formulated as follows:

$$\begin{aligned} B_{EST} &= (E_{QN} - E_{QI}) - N * P \\ B_{ACT} &= (A_{QN} - A_{QI}) - N * (A_{UI} - A_{UN}) \end{aligned}$$

Our goal is to compute P so that the estimated index benefit when considering any maintenance statements in the workload approaches the actual benefit of the index. This can be presented as follows:

$$\begin{aligned} B_{EST} &= B_{ACT} \\ (E_{QN} - E_{QI}) - N * P &= (A_{QN} - A_{QI}) - N * (A_{UI} - A_{UN}) \end{aligned}$$

For our calibration, we use the point where the estimated and actual benefits are both zero. We call it the zero benefit point.

$$\begin{aligned} B_{EST} &= B_{ACT} = 0 \\ (E_{QN} - E_{QI}) - N * P &= (A_{QN} - A_{QI}) - N * (A_{UI} - A_{UN}) = 0 \end{aligned}$$

Therefore, we can calculate P as follows:

$$P = \frac{(E_{QN} - E_{QI})}{(A_{QN} - A_{QI})} * (A_{UI} - A_{UN})$$

We define these new variables:

$$\begin{aligned} K_E &= (E_{QN} - E_{QI}) \\ K_A &= (A_{QN} - A_{QI}) \\ P_A &= (A_{UI} - A_{UN}) \end{aligned}$$

Using these variables, we can rewrite P as:

$$P = \frac{K_E}{K_A} * P_A$$

To find the calibration value for our estimated index insertion penalty, we designed an experiment to compare the value of P as calculated using the formula above and P_E which is the value of the index penalty as estimated by the advisor using the calculation equations in Section 3.5.2. To calculate P we select a database, a workload, and a disk space budget upon which we will base this calculation. We calculate K_E, K_A, and P_A using the following procedure for every index I_i identified by the advisor as being useful for the workload:

1. We identify the queries that can use index I_i in their plan, we call this set of queries SQ_{I_i}.

2. We measure the execution times of queries in SQ_{I_i}. The total execution time of queries in SQ_{I_i} is A_{QN}.

3. We measure the time needed to perform an insertion in the data, which will affect index I_i. We repeat this experiment for values of N that are equivalent to 1 insertion, 10%, 100%, and 200% of the size of the data. We calculate A_{UN} from the measured values for different N values.

4. We create index I_i and calculate the values A_{QI} and A_{UI} for different N values. The average time to do one insertion is the penalty incurred because of the index P_A.

5. We run the XML Index Advisor to find the estimated values E_{QN}, E_{QI}.

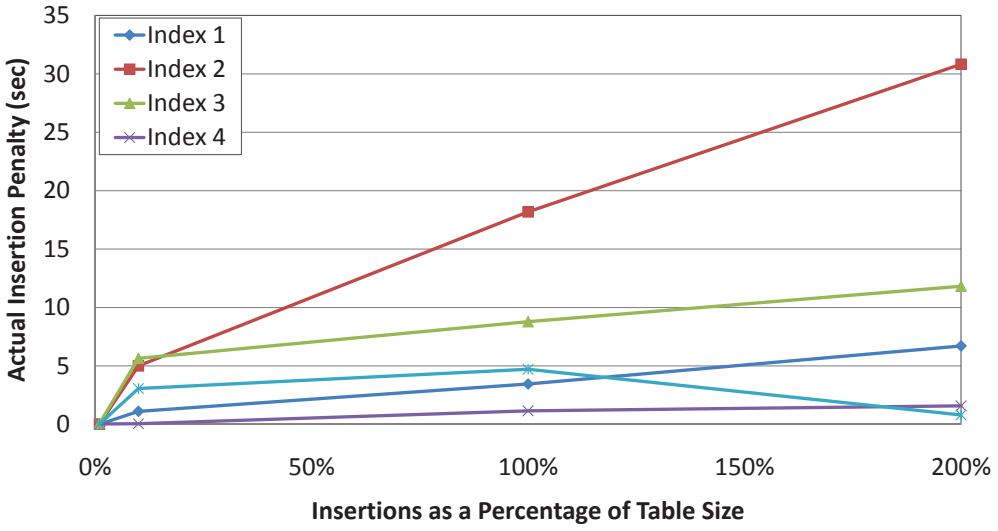


Figure A.1: Measured insertion penalty due to different indexes in created in the database.

For estimating P in this thesis, we use the queries of the TPoX workload as our calibration queries, and we give the advisor a disk space budget of 2GB. The advisor recommends 5 indexes for these queries that are built on the same table that we are inserting data in. We plotted the penalty incurred due to inserting XML data into the database when these indexes are created in the database for different N values and we confirmed that the penalty increases linearly with N

(Figure A.1). Hence, we can take the average penalty per maintenance operation performed to the database from all these runs to be P_A.

Table A.1 summarizes the results found from running the penalty evaluation experiment. We also compare P to the penalty calculated by the XML Index Advisor for each index P_E. On average, we calculate P as 1189498 and P_E as 130. This means that to calibrate the penalty estimated by the advisor we need to multiply the calculated value P_E by $\frac{P}{P_E} = \frac{1189498}{130} \approx 9150$. This is the calibration factor we use in this thesis.

Index	Queries	E_{QN}	E_{QI}	K_E	A_{QN}	A_{QI}	K_A	P_A	P	P_E
I1	3	4.3E6	2.2E6	2E6	1.5E3	1.4E3	39.2	6.4E-4	34.5	5.4
I2	2,4,6,7	5.6E10	3.1E8	6E10	5.2E2	1.8E2	334	5E-3	8E5	599
I3	2,7	5.5E10	2.9E8	6E10	2E2	24.87	180	1.7E-2	5E6	17.2
I4	4,6	2.6E7	2.3E8	3E6	3.3E2	158.99	171	7E-3	107	19.2
I5	6	3.7E6	1.1E6	3E6	2.2E2	171.92	45	4.5E-3	265	8.8

Table A.1: Calibration experiment results for multiple indexes.

Appendix B

Data and Workloads

In our experiments we have used two benchmarks: XMark [78] and TPoX [70]. TPoX, which stands for Transaction Processing over XML, is an application-level XML database benchmark, which simulates a financial multi-user workload. The main focus of TPoX is to evaluate the performance of XML database systems. In the rest of this appendix we describe the TPoX data, TPoX main workload, which is part of the benchmark, and the other workloads we used in the experiments.

B.1 TPoX Data

TPoX has five logical data entities: (1) Customer, (2) Account, (3) Holding, (4) Order, and (5) Security. These five entities are represented by three XML schemas:

1. `CUSTACC`: every XML document includes a customer's information and all of her accounts and holdings.

2. `ORDER`: every document includes an order information.

3. `SECURITY`: a fixed number of 20833 security documents representing the vast majority of US-traded stocks.

TPoX has its own data generator that efficiently generates millions of XML documents with well-defined value distributions and referential consistency across documents. The TPoX data generation process is described in [70] and [86].

B.2 TPoX Workload

TPoX has a set of transactions that represent a financial scenario to be run on the generated data. These transactions include the following:

1. Queries in XQuery and SQL/XML formats.

2. Maintenance statements in the form of insert, update, and delete operations.

In the rest of this section, we first list the 11 TPoX queries in the XQuery language. We then describe the method we used to generate a synthetic workload to use in our evaluation experiments in Chapter 3. Finally, we list the queries that we have modified to use in our evaluation experiments in Chapters 4 and 5.

B.2.1 Main TPoX Workload Queries

The following are the 11 TPoX queries that are defined as part of the benchmark. We have used this query workload in the experiments in Sections 3.8.2 and 3.8.5.

Q1: Retrieve an order with the specified `ID`.

```
for $ord in ("ORDER.ODOC")/FIXML
where $ord/Order/@ID = "103415"
return $ord/Order
```

Q2: Return a security having the specified `Symbol`.

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

Q3: Return a customer profile for the specified customer.

```
for $cust in ("CUSTACC.CADOC")/Customer
where $cust/@id = 2009
return
    <Customer_Profile CUSTOMERID="{$cust/@id}">
        {$cust/Name}
        {$cust/DateOfBirth}
        {$cust/Gender}
        {$cust/Nationality}
        {$cust/CountryOfResidence}
        {$cust/Languages}
        {$cust/Addresses}
        {$cust/EmailAddresses}
    </Customer_Profile>
```

Q4: Return a list of securities in a particular sector, with a given `PE` **and** `Yield` **ranges. The query uses** `'*'` **in** `/Security/SecurityInformation/*/Sector` **to filter and retrieve both funds and stocks of the** `Sector`**.**

```
for $sec in ("SECURITY.SDOC")/Security
where $sec/SecurityInformation/*/Sector = "Energy" and $sec/PE >= 30
      and $sec/PE < 35 and $sec/Yield > 4.5
return
    <Security>
        {$sec/Symbol}
        {$sec/Name}
        {$sec/SecurityType}
        {$sec/SecurityInformation//Sector}
        {$sec/PE}
        {$sec/Yield}
    </Security>
```

Q5: Return the securities in each account of the specified customer.

```
for $cust in db2-fn:xmlcolumn("CUSTACC.CADOC")/Customer[@id=1011]
return
    <Customer>{$cust/@id}
        {$cust/Name}
        <Customer_Securities>
            {
             for $account in $cust/Accounts/Account
             return
                <Account BALANCE="{$account/Balance/OnlineActualBal}"
                         ACCOUNT_ID="{$account/@id}">
                    <Securities>
                        {$account/Holdings/Position/Name}
                    </Securities>
                </Account>
            }
        </Customer_Securities>
    </Customer>
```

Q6: Print the open price of a particular security.

```
for $s in ("SECURITY.SDOC")/Security
where $s/Symbol = "SFDBX"
return
    <print>The open price of the security "{$s/Name/text()}"
           is {$s/Price/PriceToday/Open/text()} dollars
    </print>
```

Q7: Return the most expensive order of the customer with the specified id.

```
let $orderprice :=
    for  $cust in ("CUSTACC.CADOC")/Customer[@id=1011]
    for $ord in ("ORDER.ODOC")/FIXML/Order
                [@Acct=$cust/Accounts/Account/@id/fn:string(.)]
    return $ord/OrdQty/@Cash
return max($orderprice)
```

Q8: Return the maximum order value for the stocks in a certain industry bought by customers living in the specified State.

```
let $order  :=
    for $ss in ("SECURITY.SDOC")/Security
                [SecurityInformation/StockInformation/Industry =
                      "Software&amp;Programming"]
    for $ord in ("ORDER.ODOC")/FIXML/Order[Instrmt/@Sym=$ss/Symbol/fn:string(.
    for $cs in ("CUSTACC.CADOC")/Customer[Addresses/Address/State="West Virgin
                /Accounts/Account[@id=$ord/@Acct/fn:string(.)]
    return $ord/OrdQty/@Cash
return string(max($order))
```

Q9: Retrieve the names of the customers in the specified country who have orders higher than a given value.

```
for $ord in ("ORDER.ODOC")/FIXML/Order
for $cust in ("CUSTACC.CADOC")/Customer
              [Accounts/Account/@id=$ord/@Acct/fn:string(.)]
where $ord/OrdQty/@Cash>3000 and $cust/CountryOfResidence="Taiwan"
return $cust/ShortNames/ShortName
```

Q10: Get the phone numbers of customers in a specific postal code who have sold any security. Sort by customer last name.

```
for $cust in ("CUSTACC.CADOC")/Customer[Addresses/Address/PostalCode=26652]
for $ord in ("ORDER.ODOC")/FIXML/Order
             [@Acct=$cust/Accounts/Account/@id/fn:string(.) and @Side="2"]
order by $cust/Name/LastName/text()
return
    <Customer>
        {$cust/Name/LastName/text()} -
        {$cust/Addresses/Address[@primary="Yes"]/Phones/Phone[@primary="Yes"]}
    </Customer>
```

Q11: For a given order, get the current open price of the corresponding security.

```
for $ord in ("ORDER.ODOC")/FIXML/Order[@ID="109505"]
for $sec in ("SECURITY.SDOC")/Security[Symbol=$ord/Instrmt/@Sym/fn:string(.)]
return
<Today_Order_Price ORDER_ID="{$ord/@ID}">
   {string($ord/OrdQty/@Qty*$sec/Price/PriceToday/Open)}
</Today_Order_Price>
```

B.2.2 Synthetic Workload

In Section 3.8.3, we generated a Synthetic TPoX workload to show the benefit of recommending generalized indexes. The Synthetic workload is composed of the main TPoX queries listed in the previous section as well as other generated queries. We used Q1, Q4, and Q5 as templates for generating more queries. The query templates that we have chosen represent different return value complexities as well as the three XML tables in the TPoX database. All of the three query templates have a predicate in the FOR or WHERE clause and a return value. The method we followed to generate queries is to insert a new predicate in the FOR or WHERE

clause while keeping the RETURN clause the same as it is in the original query. For example, Table B.1 summaries some of the XPath path expressions and their values that we used to replace the predicate in the FOR clause of Q5 to generate new queries.

XPath expression	Value
`/Customer/BankingInfo/CustomerStatus`	"Inactive"
`/Customer/Name/LastName`	"Tedrick"
`/Customer/Nationality`	"Georgia"
`/Customer/Addresses/Address/Phones/Phone/AreaCode`	519
`/Customer/Addresses/Address/Phones/Phone/Number`	8849780
`/Customer/BankingInfo/PremiumCustomer`	"Yes"

Table B.1: XPath path expressions and the values inserted as new predicates in the query Q5 template.

B.2.3 Updated Workload Queries

To cope with the limitations of the XMLTable View Advisor described in Chapter 4, we made changes to the TPoX queries listed in Section B.2.1. The main changes that we made to queries are of two types:

1. Adding data type casting information to all XPath path expressions that appear in queries.

2. Simplifying the query by removing the complex constructs or the expressions that select multiple values.

In the revisited query workload, we have added type casting to all the queries. We have dropped query Q8 because it performs a three table join that is not currently supported by our XMLTable view advisor. We have also simplified the complex return structure in Q5. The new version of Q5 is as follows:

Q5': Return the name of the specified customer.

```
for $cust in db2-fn:xmlcolumn("CUSTACC.CADOC")/Customer[@id/xs:double(.) = 1011]
return
    <Customer>{$cust/@id/xs:double(.)}
        {$cust/Name/string(.)}
    </Customer>
```

Appendix C

An Index Advisor Example Run

In this appendix, we describe the results of various XML Index Advisor runs to highlight the decisions made by various phases of the advisor and the differences between the proposed search algorithms. First, we show the enumerated candidate index patterns for the TPoX workload in Section C.1. Next, we show the generalized XML patterns generated by the index advisor in Section C.2. Finally, we compare the set of XML indexes recommended by the different search algorithms studied in this thesis in Section C.3.

C.1 Basic Candidate Set

The first phase of the index advisor is to enumerate candidate XML patterns that are useful to the input workload. Table C.1 shows the basic candidate XML index patterns enumerated by the XML Index Advisor for the TPoX workload. For each candidate, we mention the data type of the values included in it as well as the workload queries that would benefit from it.

C.2 Candidate Generalization

The XML Index Advisor uses the algorithm described in Section 3.4.1 for generalizing the set of basic candidate index patterns to generate new index patterns that

Index	Index XPath expression	Type	Queries
I0	`/Customer/@id`	double	Q0, Q3, Q4
I1	`/Security/Symbol`	string	Q1, Q7, Q8
I2	`/FIXML/Order/@ID`	string	Q1, Q6
I3	`/Customer/CountryOfResidence`	string	Q2
I4	`/Customer/Accounts/Account/@id`	string	Q2, Q9
I5	`/FIXML/Order/OrdQty/@Cash`	double	Q2
I6	`/FIXML/Order/@Acct`	string	Q3, Q5
I7	`/FIXML/Order/@Side`	string	Q5
I8	`/Customer/Addresses/Address/PostalCode`	double	Q5
I9	`/Customer/Addresses/Address/State`	string	Q9
I10	`/FIXML/Order/Instrmt/@Sym`	string	Q9
I11	`/Security/SecurityInformation/` StockInformation/Industry	string	Q9
I12	`/Security/Yield`	double	Q10
I13	`/Security/PE`	double	Q10
I14	`/Security/SecurityInformation/*/Sector`	string	Q10

Table C.1: Basic set of candidates enumerated by the XML Index Advisor for the TPoX workload.

are useful beyond the input workload. Table C.2 lists the generalized candidate XML patterns that would be generated when running the generalization algorithm on the XML patterns in Table C.1. Figure C.1 illustrates the relation between generated candidate XML patterns, basic candidate XML patterns, and the workload queries that produced them.

C.3 Searching for the Optimal Configuration

Having enumerated a set of candidate index patterns, the XML Index Advisor employs a combinatorial search algorithm to find the best set of candidate index

Index	Index XPath expression	Type	Queries
I15	`/FIXML/Order/@*`	string	Q1, Q3, Q5, Q6
I16	`/Customer//*`	string	Q2, Q9
I17	`/Security/SecurityInformation//*`	string	Q9, Q10
I18	`/Security/*`	double	Q10
I19	`/FIXML/Order//@*`	string	Q1, Q3, Q5, Q6, Q9
I20	`/Security//*`	string	Q1, Q7, Q8, Q9, Q10

Table C.2: Expanded set of candidates generated by the XML Index Advisor for the TPoX workload.

patterns to recommend that fits within a disk space budget. We have proposed two search algorithms in Section 3.6, namely the greedy search with heuristics and the top down search. We have implemented two versions of the top down search: (1) top down full, which evaluates the benefits of candidate XML indexes in every iteration of the algorithm to account for the interaction between these indexes and the indexes already selected in previous iterations, and (2) top down lite, which evaluates the benefits of candidate XML indexes once. We have also implemented a naive greedy search algorithm and a dynamic programming algorithm as described in Section 3.7.3. In this section, we compare the set of XML indexes recommended by the five search algorithms for different disk size constraints.

The configuration that includes all the basic candidate index patterns is the smallest configuration that is overfitted to the input workload. The size of the basic candidate XML patterns enumerated for the TPoX workload is 96.4MB. In this section, we compare the five candidate search algorithms for a disk constraint that is smaller than the basic candidate configuration size (50MB) and two disk constraints that are larger than the basic candidate configuration size (100MB and 1000MB). Figure C.2 shows the DAG of all the candidate XML index patterns for the TPoX workload after annotating it with information about the estimated benefits and sizes of the XML index patterns.

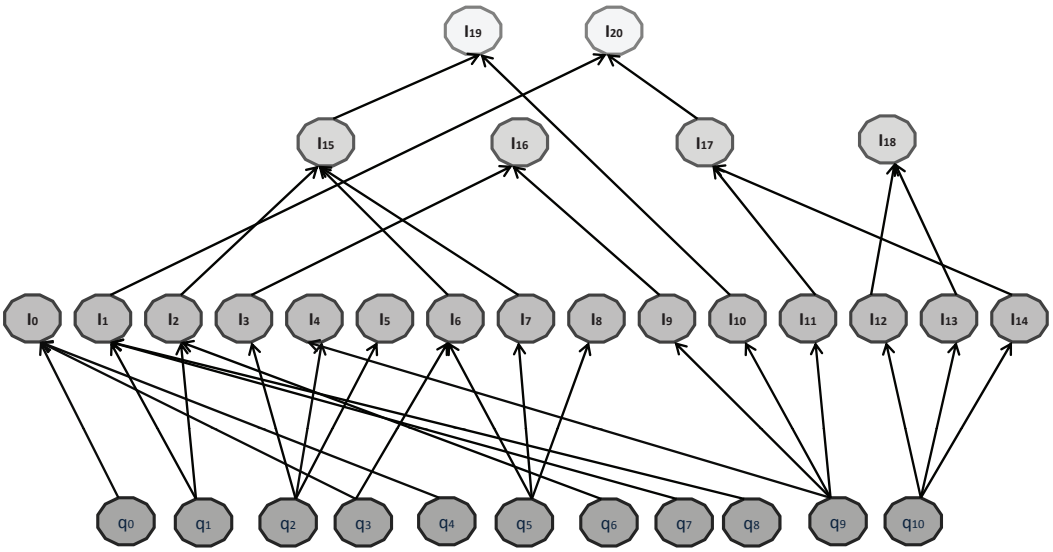


Figure C.1: The DAG showing the relationship between the TPoX workload queries and candidate XML patterns enumerated by the XML Index Advisor.

C.3.1 Disk Constraint Smaller than Basic Candidate Set (50MB)

In this section, we compare the XML index recommendations of the five candidate search algorithms that we have implemented in the XML Index Advisor when the disk space constraint is 50MB.

Greedy Search

The initial step of the greedy search algorithm is to sort all the candidate indexes according to their benefit size ratio. Table C.3 shows the list of candidates after sorting them. Table C.4 shows the iterations of the algorithm until it chooses a configuration of size 47.95MB. Tables C.5 and C.6 show the XML indexes recommended by the greedy search algorithm.

Greedy Search with Heuristics

For low disk size constraint, the greedy search with heuristics behaves the same as greedy search because the smaller candidates are selected first in both cases. The

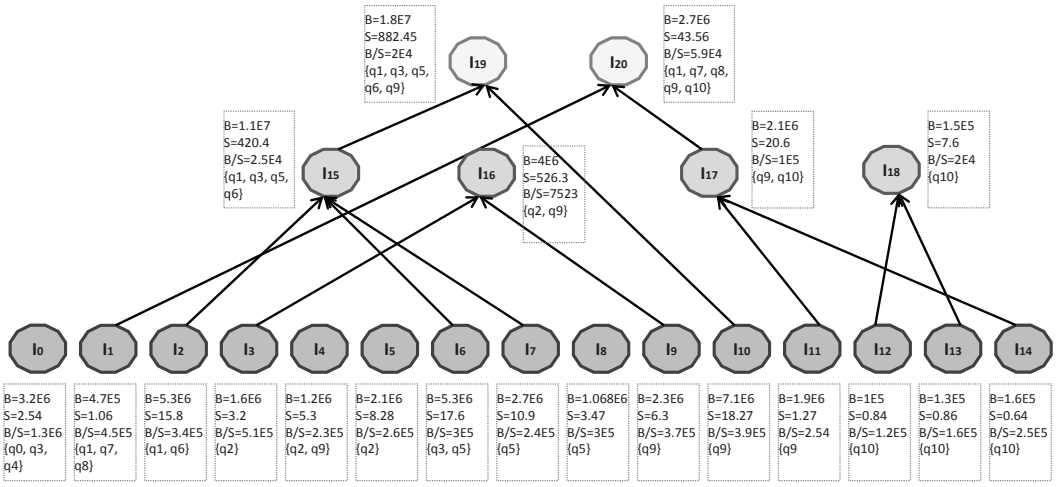


Figure C.2: The DAG constructed for the TPoX workload after annotating it with the benefit and size information for all XML patterns.

iterations of the greedy search algorithm with heuristics are the same as those listed for greedy search in Table C.4. Hence, the recommended indexes are also the same as the ones listed in Tables C.5 and C.6.

Top Down Search

In the top down search algorithm, we traverse the DAG nodes by replacing nodes in the recommended configuration of indexes with their children until the configuration size fits within the disk constraint. For disk size constraint that is smaller than the total size of basic candidates, the recommended configuration will be composed of basic candidates (leaves of the DAG) and yet does not fit into the disk constraint. In this case, we perform greedy search to find the best subset of basic candidate indexes to recommend.

Top Down Lite: The greedy search performed by the top down lite algorithm is similar to the naive greedy search algorithm except that we do not have generalized indexes in the pool of candidates we are searching. When running the top down lite search for a disk size constraint of 50MB, the set of recommended indexes are the same as the one recommended by the greedy search and greedy search with heuristics listed in Tables C.5 and C.6.

Top Down Full: In the top down full algorithm, we evaluate the benefit of index every iteration to account for index interaction with the indexes selected by the algorithm so far. We note that the recommendations of the top down lite algorithm and the top down full algorithm might differ in this case. For example, when running the top down lite search, the indexes I12 (`/Security/Yield`) and I13 (`/Security/PE`) are not selected, which is different from the recommendations of the naive greedy search, greedy search with heuristics, and top down lite algorithms. The main reason is that the benefits of indexes I12 and I13 are negligible when index I14 is used in the query plan of query Q10 in the TPoX workload. The total size of the index configuration recommended by the top down full algorithm is 46.25MB. Table C.7 shows the indexes recommended by the top down full algorithm for a disk size constraint of 50MB.

Dynamic Programming Search

The dynamic programming search algorithm tries to construct the set of optimal sub-configurations in every iteration. The indexes recommended by the algorithm are shown in Tables C.8 and C.9.

Index	Benefit/Size	Benefit	Size	Subsumed Indexes	Queries
I0	1248180	3178959	2.55	I0	Q0, Q3, Q4
I3	508942	1622254	3.19	I3	Q2
I1	445314	473146	1.06	I1	Q1, Q7, Q8
I10	386509	7059823	18.27	I10	Q9
I2	335224	5300725	15.81	I2	Q1, Q6
I8	305522	1059780	3.47	I8	Q5
I6	301321	5301365	17.59	I6	Q3, Q5
I11	280796	355383	1.27	I11	Q9
I5	255931	2119432	8.28	I5	Q2
I14	246177	157707	0.64	I14	Q10
I7	243364	2650389	10.89	I7	Q5
I4	231657	1227056	5.3	I4	Q2, Q9
I9	192114	1218723	6.34	I9	Q9
I13	156717	134679	0.86	I13	Q10
I12	119090	100483	0.84	I12	Q10
I15	25218	10602090	420.42	I2, I6, I7, I15	Q1, Q3, Q5, Q6
I17	24914	513075	20.59	I12	Q9, Q10
I20	22639	986191	43.56	I1, I11, I14, I17, I20	Q1, Q7, Q8, Q9, Q10
I19	20015	17661913	882.45	I2, I6, I7, I10, I15, I19	Q1, Q3, Q5, Q6, Q9
I18	19527	148895	7.63	I12, I13, I18	Q10
I16	5386	2834331	526.28	I3, I9, I16	Q2, Q9

Table C.3: A sorted list of candidate index patterns according to their Benefit/Size ratio.

Itr	Cnd	Recommended Indexes	Size	Query Coverage
0		{}	0	{}
1	I0	{I0}	2.55	{Q0, Q3, Q4}
2	I3	{I0, I3}	5.73	{Q0, Q2–Q4}
3	I1	{I0, I1, I3}	6.8	{Q0-Q4, Q7, Q8}
4	I10	{I0, I1, I3, I10}	25.06	{Q0–Q4, Q7–Q9}
5	I2	{I0, I1, I2, I3, I10}	40.88	{Q0–Q4, Q6–Q9}
6	I8	{I0, I1, I2, I3, I8, I10}	44.34	{Q0–Q9}
7	I6	{I0, I1, I2, I3, I8, I10}	44.34	{Q0–Q9}
8	I11	{I0, I1, I2, I3, I8, I10, I11}	45.61	{Q0–Q9}
9	I5	{I0, I1, I2, I3, I8, I10, I11}	45.61	{Q0–Q9}
10	I14	{I0, I1, I2, I3, I8, I10, I11, I14}	46.25	{Q0–Q10}
11	I7	{I0, I1, I2, I3, I8, I10, I11, I14}	46.25	{Q0–Q10}
12	I4	{I0, I1, I2, I3, I8, I10, I11, I14}	46.25	{Q0–Q10}
13	I9	{I0, I1, I2, I3, I8, I10, I11, I14}	46.25	{Q0–Q10}
14	I13	{I0, I1, I2, I3, I8, I10, I11, I13, I14}	47.11	{Q0–Q10}
15	I12	{I0, I1, I2, I3, I8, I10, I11, I12, I13, I14}	47.95	{Q0–Q10}
16	I15	{I0, I1, I2, I3, I8, I10, I11, I12, I13, I14}	47.95	{Q0–Q10}
17	I17	{I0, I1, I2, I3, I8, I10, I11, I12, I13, I14}	47.95	{Q0–Q10}
18	I20	{I0, I1, I2, I3, I8, I10, I11, I12, I13, I14}	47.95	{Q0–Q10}
19	I19	{I0, I1, I2, I3, I8, I10, I11, I12, I13, I14}	47.95	{Q0–Q10}
20	I18	{I0, I1, I2, I3, I8, I10, I11, I12, I13, I14}	47.95	{Q0–Q10}
21	I16	{I0, I1, I2, I3, I8, I10, I11, I12, I13, I14}	47.95	{Q0–Q10}

Table C.4: Iterations of the greedy search algorithm for the TPoX workload, Budget = 50MB

Index	Size	Query Coverage
I0	2.55	Q0, Q3, Q4
I1	1.06	Q1, Q7, Q8
I2	15.81	Q1, Q6
I3	3.19	Q2
I8	3.47	Q5
I10	18.27	Q9
I11	1.27	Q9
I12	0.84	Q10
I13	0.86	Q10
I14	0.64	Q10

Table C.5: Recommended indexes for the TPoX workload using greedy search, Budget = 50MB.

Query	Indexes Used	Estimated Speedup
Q0	I0	20000.37
Q1	I1, I2	25525.76
Q2	I3	1.61
Q3	I0	1.4
Q4	I0	27998.03
Q5	I8	1.4
Q6	I2	70032.31
Q7	I1	5209.66
Q8	I1	5209.16
Q9	I10, I11	7.88
Q10	I12, I13, I14	5202.88

Table C.6: Index recommendations per query for the TPoX workload using greedy search, Budget = 50MB.

Index	Size	Query Coverage
I0	2.55	Q0, Q3, Q4
I1	1.06	Q1, Q7, Q8
I2	15.81	Q1, Q6
I3	3.19	Q2
I8	3.47	Q5
I10	18.27	Q9
I11	1.27	Q9
I14	0.64	Q10

Table C.7: Recommended indexes for the TPoX workload using top down full search, Budget = 50MB.

Index	Size	Query Coverage
I0	2.55	Q0, Q3, Q4
I1	1.06	Q1, Q7, Q8
I2	15.81	Q1, Q6
I3	3.19	Q2
I4	5.3	Q2, Q9
I8	3.47	Q5
I10	18.27	Q9

Table C.8: Recommended indexes for the TPoX workload using the dynamic programming search algorithm, Budget = 50MB.

Query	Indexes Used	Estimated Speedup
Q0	I0	520000.37
Q1	I1, I2	25525.76
Q2	I3, I4	1.61
Q3	I0	1.4
Q4	I0	27998.03
Q5	I8	1.4
Q6	I2	70032.31
Q7	I1	5209.66
Q8	I1	5209.16
Q9	I4, I10	41.31
Q10		1

Table C.9: Index recommendations per query for the TPoX workload using the dynamic programming search algorithm, Budget = 50MB.

C.3.2 Disk Constraint Slightly Larger than Basic Candidate Set (100MB)

In this section, we study the different search algorithms when the disk space constraint is 100MB, which is very close to the size of the configuration required by the basic candidate indexes.

Greedy Search

The iterations of the greedy search algorithm proceed as shown in Table C.4 except that different decisions are taken at different iterations. At iterations 7, 9, 11, 12, and 13, index patterns I6, I5, I7, I4, and I9 are added to the configuration that the algorithm recommends. The recommended configuration size is 96.4MB.

Greedy Search with Heuristics

The search proceeds the same as the greedy search. The resulting configuration is the same as the one recommended by the greedy search.

Top Down Search

For a disk space budget of 100MB, the top down search algorithm navigates the DAG of candidate indexes and finally recommends the basic candidate indexes. We next show how the DAG is navigated in top down lite and top down full search algorithms.

Top Down Lite: Figure C.3 shows the initial state of the DAG with indexes I0, I4, I5, I8, I16, I18, I19, and I20 selected as the configuration to recommend. The nodes of the DAG are annotated with the calculated values for ΔB and ΔS, which are described in Section 3.6.3. Figures C.4, C.5, C.6, C.7, C.8, and C.9, show the iterations of the algorithm to traverse the DAG based on the values of ΔB and ΔS calculated at every iteration. At each iteration, the indexes currently selected by the algorithm are shown with dotted circles.

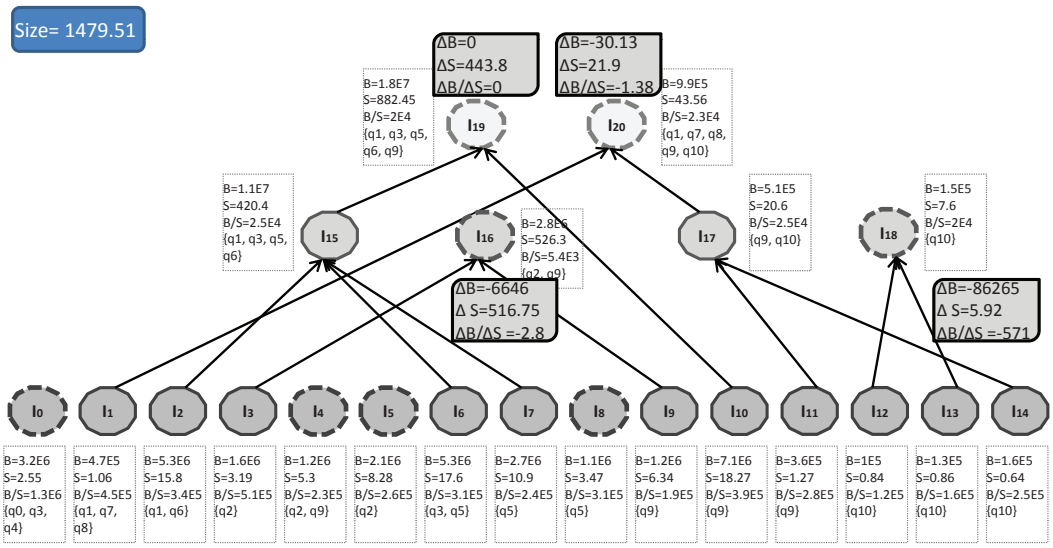


Figure C.3: Initial state of the DAG showing the XML patterns initially selected by the top down lite search algorithm.

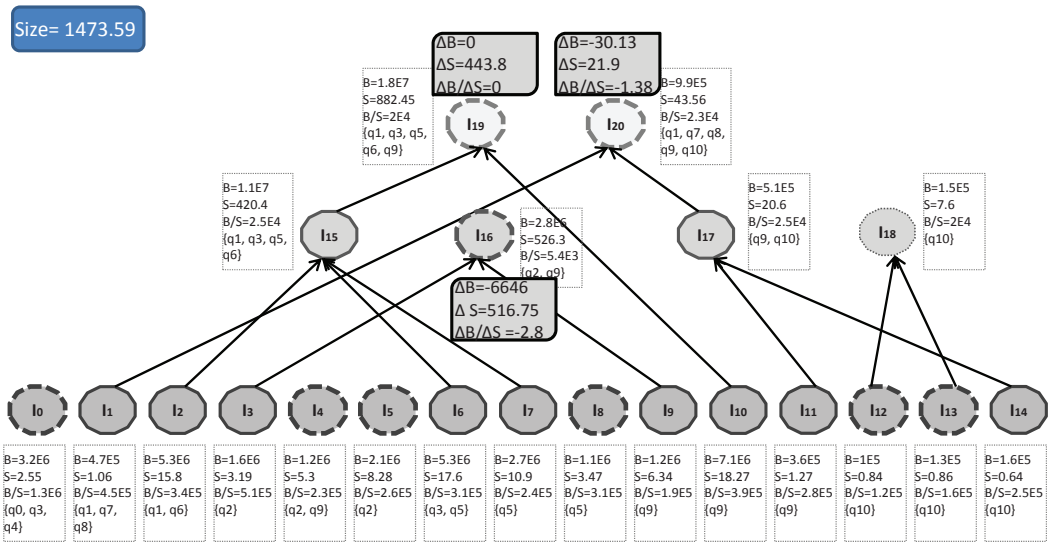


Figure C.4: State of the DAG after the first iteration of the top down lite search algorithm, Budget = 100MB.

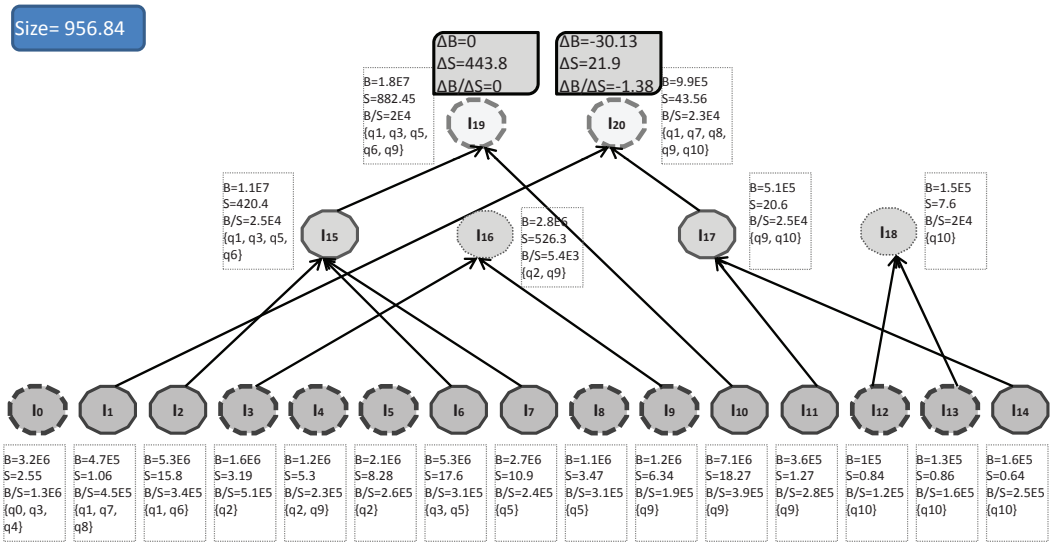


Figure C.5: State of the DAG after the second iteration of the top down lite search algorithm, Budget = 100MB.

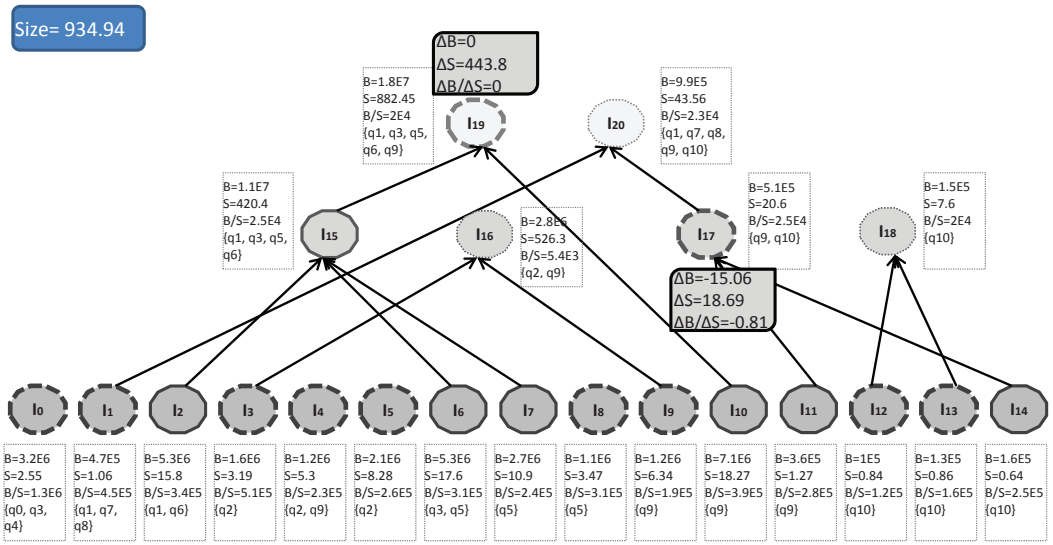


Figure C.6: State of the DAG after the third iteration of the top down lite search algorithm, Budget = 100MB.

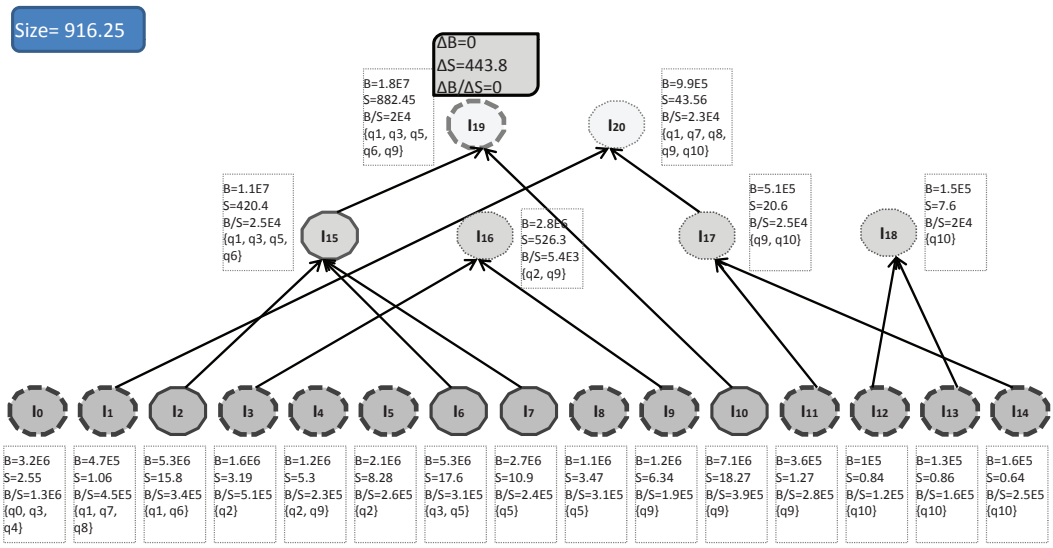


Figure C.7: State of the DAG after the fourth iteration of the top down lite search algorithm, Budget = 100MB.

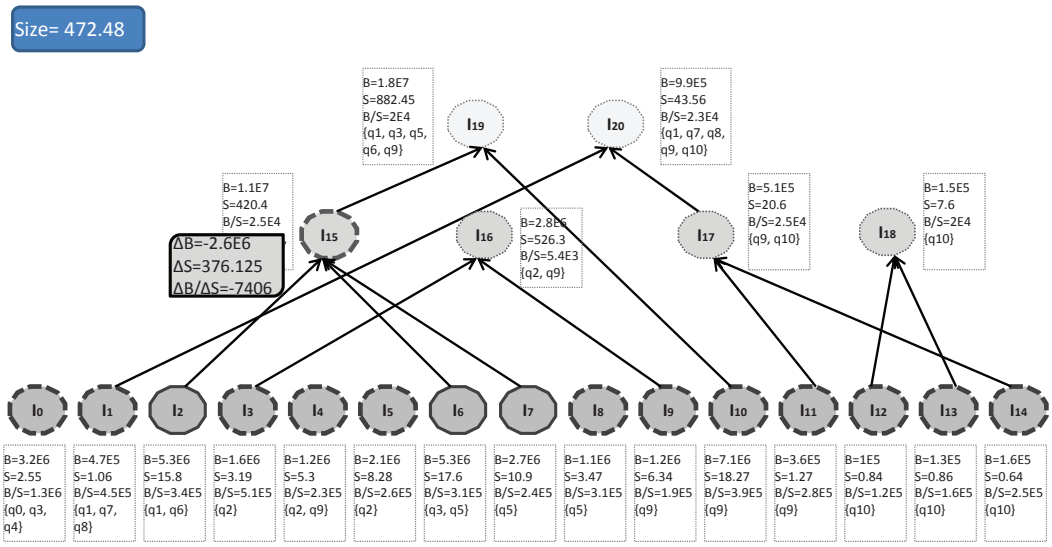


Figure C.8: State of the DAG after the fifth iteration of the top down lite search algorithm, Budget = 100MB.

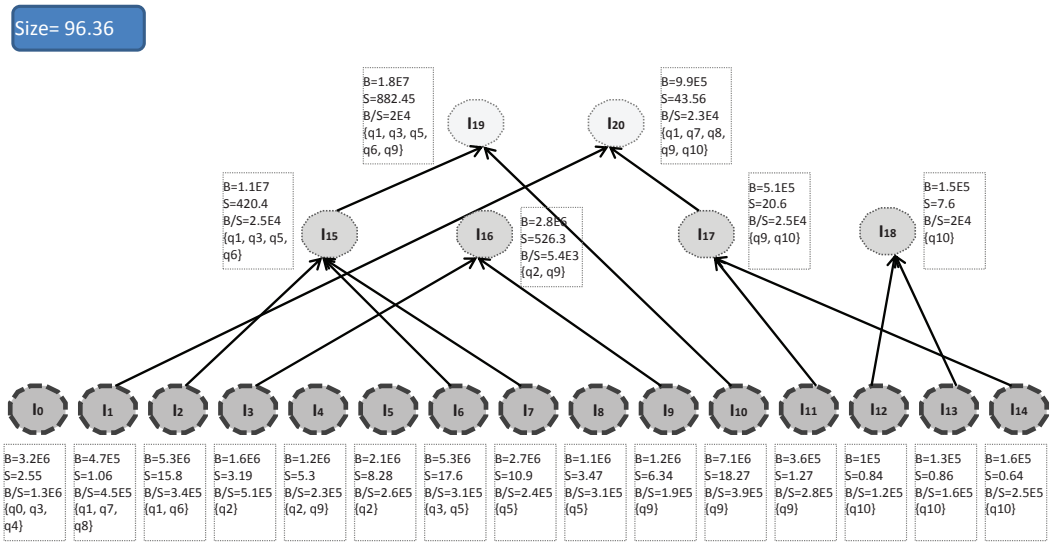


Figure C.9: State of the DAG after the sixth iteration of the top down lite search algorithm, Budget = 100MB.

Top Down Full: Figure C.10 shows the initial state of the DAG with indexes I0, I4, I5, I8, I16, I18, I19, and I20 selected as the configuration to recommend. The nodes of the DAG are annotated with the calculated values for ΔB and ΔS. We note that the values of ΔB and ΔS in Figure C.10 are different from those in Figure C.3, which will affect the decisions made at every iteration. Figures C.11, C.12, C.13, C.14, C.15, and C.16 show the iterations of the algorithm to traverse the DAG based on the values of ΔB and ΔS calculated at every iteration. As before, the indexes currently selected by the algorithm at each iteration are shown with dotted circles.

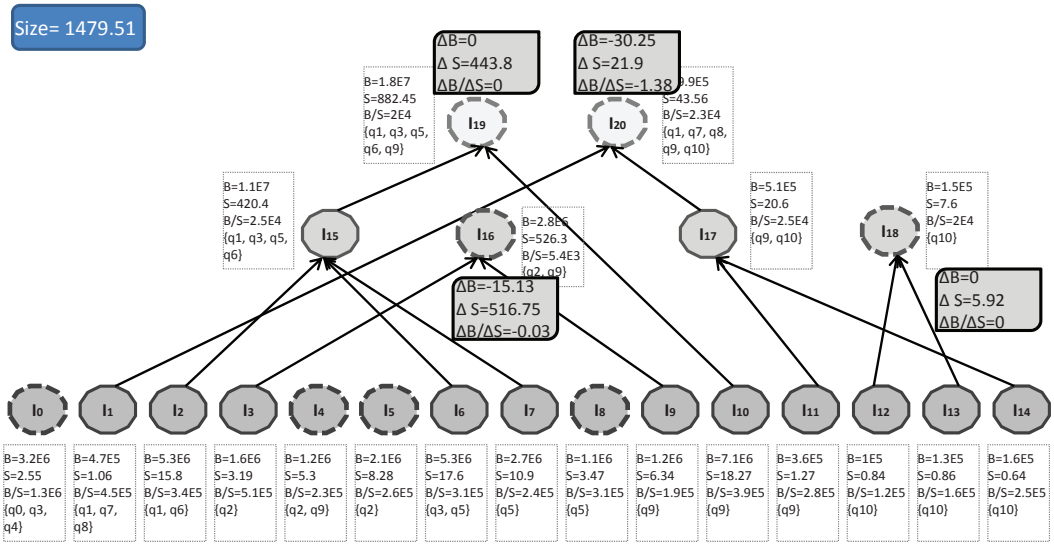


Figure C.10: Initial state of the DAG showing the XML patterns initially selected by the top down full search algorithm.

C.3.3 Disk Constraint Much Larger than Basic Candidate Set (1000MB)

Greedy Search

The algorithm behaves as described in Section C.3.2 and more indexes can now be included in the configuration that it recommends. One more index is now added

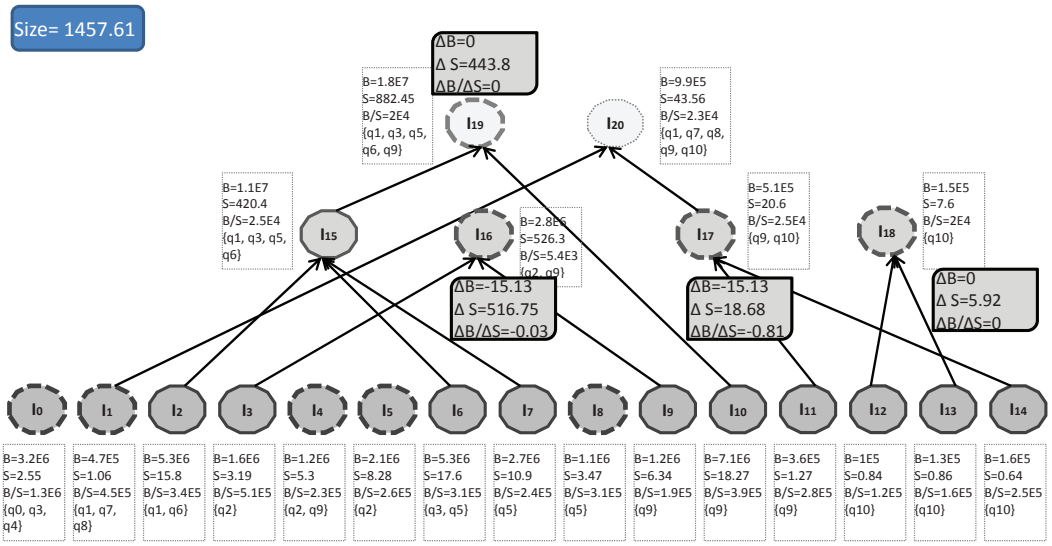


Figure C.11: State of the DAG after the first iteration of the top down full search algorithm, Budget = 100MB.

to the recommended set of indexes which is I15 from iteration 16. The size of the recommended configuration is 524.4MB. We note that I15 is redundant in this case as it is a generalized form of indexes I2, I6, and I7, which are already selected in the configuration.

Greedy Search with Heuristics

The configuration recommended by the algorithm is the same as the one recommended for a disk constraint of 100MB (Section C.3.2). This shows that the algorithm does not usually recommend generalized indexes even if enough disk space is available.

Top Down Search

When the disk size constraint is increased, the traversal of the DAG stops when the largest configuration that fits into the disk constraint is found. The top down lite search stops at the second iteration, which is shown in Figure C.5. On the other hand, the top down full stops at the third iteration, which is shown in Figure C.13.

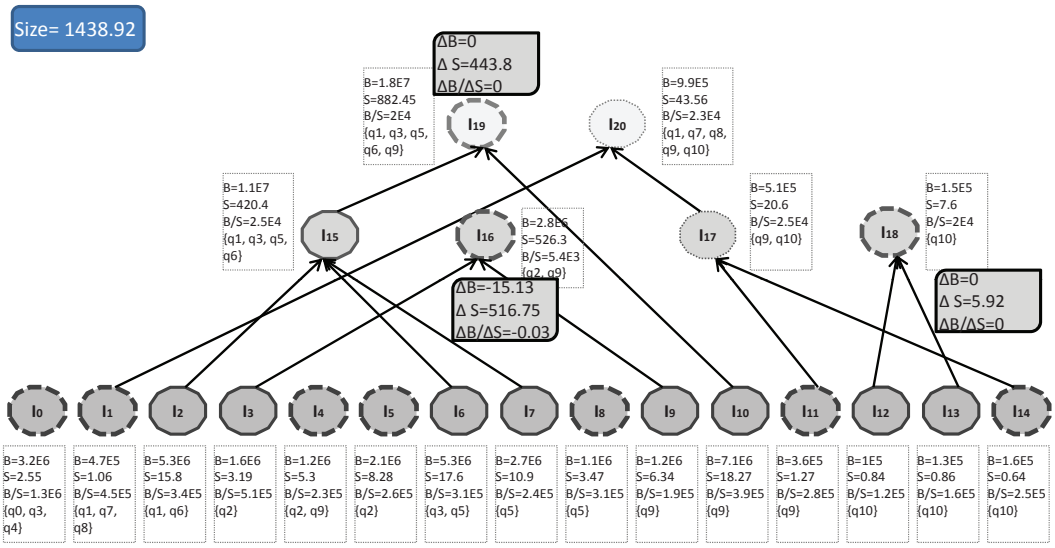


Figure C.12: State of the DAG after the second iteration of the top down full search algorithm, Budget = 100MB.

We note that in the indexes configurations recommended by both versions of the top down search algorithm recommended general indexes with no redundancy, which reflects the wise usage of the allocated disk space.

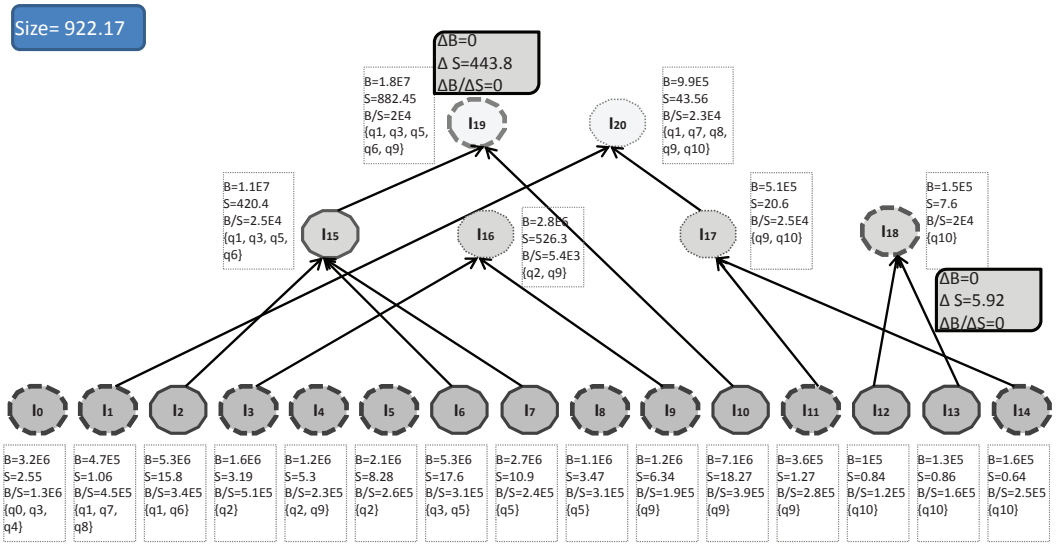


Figure C.13: State of the DAG after the third iteration of the top down full search algorithm, Budget = 100MB.

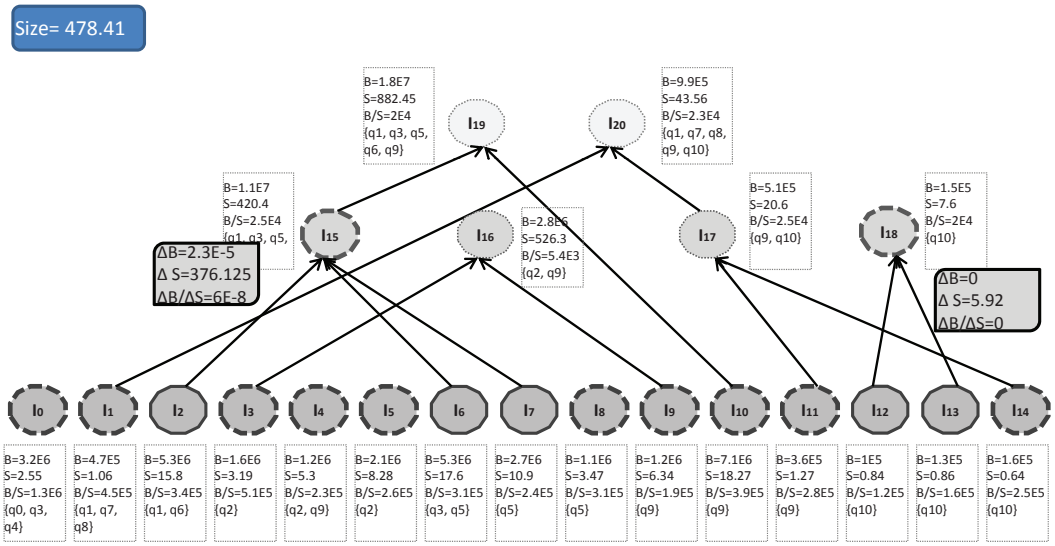


Figure C.14: State of the DAG after the fourth iteration of the top down full search algorithm, Budget = 100MB.

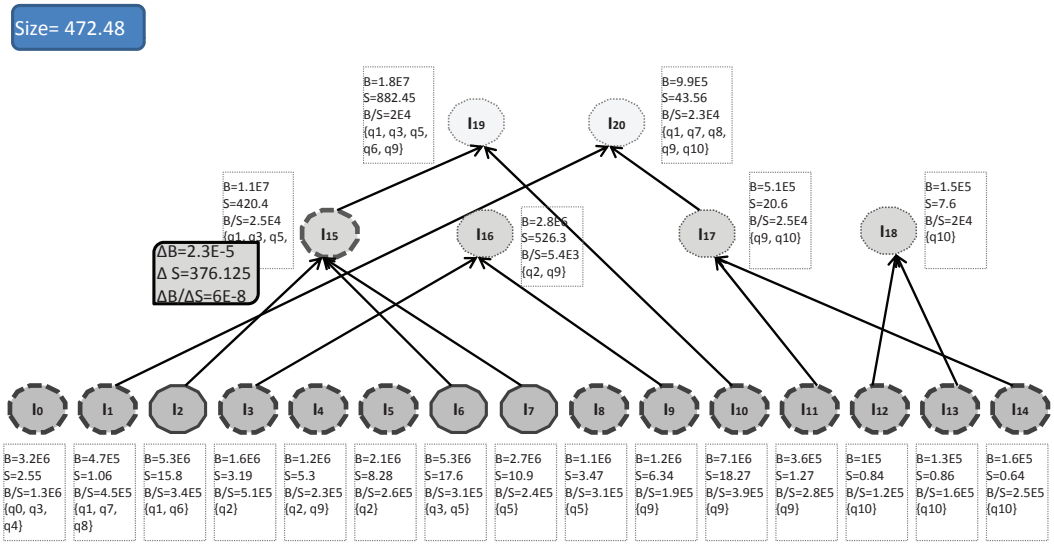


Figure C.15: State of the DAG after the fifth iteration of the top down full search algorithm, Budget = 100MB.

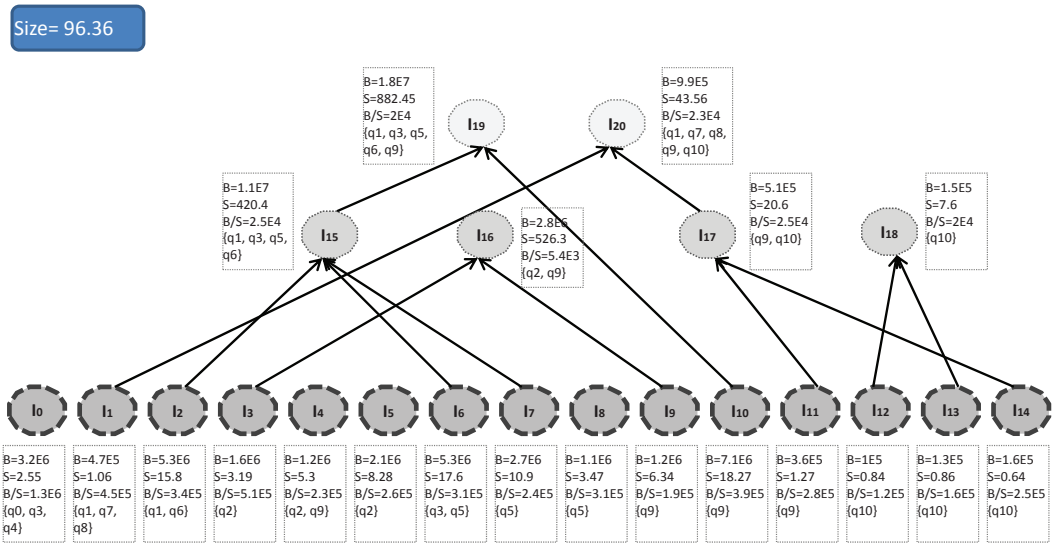


Figure C.16: State of the DAG after the sixth iteration of the top down full search algorithm, Budget = 100MB.