Adaptive Query Relaxation and Processing over Heterogeneous XML Data Sources

Jianxin Li

B.E. (NEU, China)

M.E. (NEU, China)

A dissertation

submitted to the Faculty of Information and Communication Technologies

Swinburne University of Technology

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

August 2009

Abstract

Traditionally, searching XML data with a structured query often identifies the exact matches for the query and returns them as the qualified results. However, the structural heterogeneity of the large number of XML data sources will make it difficult to answer the structured queries exactly. As such, query relaxation is necessary when the exact results do not exist or the number of the exact results is not enough. Previous work on XML query relaxation poses the problem of unnecessary computation of a big number of unqualified relaxed queries. This thesis addresses several fundamental issues in ranking a set of heterogeneous XML data sources, designing the adaptive relaxation rules and efficiently processing the relaxed queries over the data sources. Relaxing the specified queries may loose the constraints of the users' original preferences, which can increase the number of relevant results significantly. Therefore, in this work the users' queries would be answered with a ranked list of the best matched results, e.g., top-k problem means that only the top k relevant results are interesting to users. In addition, at each time we prefer to evaluate the relaxed queries over the data source that is the most relevant to the specified queries, and incrementally output the retrieved results that are guaranteed to have the higher relevance to the queries than the candidates in the other data sources, thus minimizing query processing time.

However, query relaxation would become time-consuming and ineffective in some cases, such as the data structure is too complex for users to write structured queries,

or the users do not know structured query languages. To address this problem, we allow users to issue keyword queries. Different from previous keyword search methods, we first construct structured query templates based on the given keyword query and the underlying source schemas and then evaluate the query templates to answer the original keyword query based on our proposed ranking model.

Acknowledgements

First, I would like to thank my supervisor Professor Chengfei Liu for his patience and guidance. He taught me a great deal about research and writing, and was always available for discussion. His thoughtful and painstaking comments on every aspect of my writing style and research methodology have tremendously helped me improve my work. I am grateful I had the chance to work with him.

I learned how exciting and fun research could be from professor Guoren Wang at Northeastern University, China. He encouraged me to pursue a Ph.D. in Australia, and I am forever grateful for that great advice. During the study of my Ph.D., Professor Jeffrey Xu Yu at Chinese University of Hong Kong and Dr. Jixue Liu at University of South Australia, helped me tremendously how to present a research paper in an interesting way. In addition to my thesis work, the members in Swinburne Web & Data Engineering group were always available to discuss research and non-research issues. Especially, Rui Zhou gave me invaluable comments and help on my research over the three years. The other members Hai Huang, Jiajie Xu and Shangfeng Hu have been wonderful people with whom to share ideas. Professor Yun Yang was kind enough to serve on my PhD penal committee and to provide useful comments on my work. Ms. Mandish Webb and Ms. Gillian Foster patiently answered my never-ending questions on all possible aspects of academic life and administrative details. I will miss seeing them all every day.

My parents have been a great inspiration in my life. Finally, and more importantly,

my wife Ms. Man Li has encouraged me all throughout my PhD, and was there to support me when I was discouraged or just plain tired. I could not have done it without her.

To my wife.

To my family.

"Nothing is too small to know, and nothing too big to attemp."

— William Van Horne (1843-1915)

"Knowledge is treasure, but practice is the key to it."

— Thomas Fuller (1608-1661)

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Name: Jianxin Li

Signature: _____

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to its flexibility, XML has emerged as the standard for exchanging and querying documents on the Web required for the next generation Web applications including electronic commerce and intelligent Web searching. XML documents comprise hierarchically nested collections of elements, where each element can be either atomic (i.e., raw character data) or composite (i.e., a sequence of nested sub-elements). Tags stored with elements in an XML document describe the semantics of the data. Thus, XML data, like semi-structured data, is hierarchically structured and self-describing. The widespread adoption of XML has raised many challenging issues, such as processing XML queries across large and heterogeneous XML document collections.

Traditionally, query processing techniques over XML data need to identify the exact matches for a given XML query. To find the exact matches, it is necessary to consider not only the contents but also the structures in the query. Let's take an example to illustrate the procedure of identifying the exact matches. An XML document consists of nested elements enclosed by user-defined tags that indicate the meaning of the contained content. Figure 1.1 shows an example of an XML document named "pub.xml", which contains some publication information. Since the hierarchical structure of an XML document can be usually modeled as a tree,

```
<?xml version="1.0" ?>
<publication>
      <journal title="DBMS">
            <editor> Jack</editor>
            <article>
                  <title>
                        Index Construction
                  </title>
                  <authors>
                        <author> Smith</author>
                        <author> John</author>
                  </authors>
            </article>
      </journal>
      <journal title="Algorithm">
      </journal>
</publication>
```

Figure 1.1: An example of an XML document pub.xml

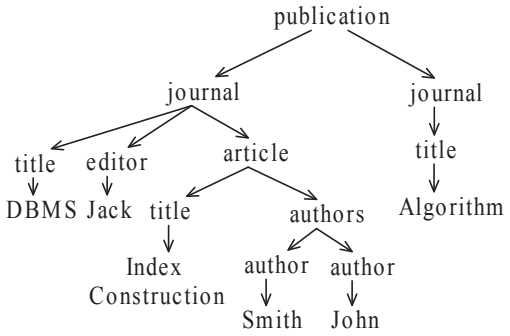


Figure 1.2: The tree representation of the XML document example

the XML document in Figure 1.1 can be represented with the tree in Figure 1.2. Therefore, we can consider XML documents on Internet as an forest of XML trees.

Figure 1.3 shows an XML query expressed in XQuery [16] over the document in Figure 1.1 where "//" indicates the ancestor-descendant relationship, and "/" indicates the parent-child relationship. This query is used to retrieve the titles of the articles that are written by "Smith" and published in a journal. It contains both structure and content information. We can use a tree to depict the query as shown

```
FOR $a IN document("http://.../pub.xml")//journal/article
      $b IN $a/title
WHERE $a/authors/author="Smith"
RETURN <article>$b</article>
```

Figure 1.3: An example of XQuery

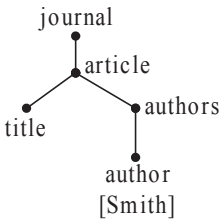


Figure 1.4: The tree representation of the example query

in Figure 1.4. In other words, this query will find all the matches of the tree pattern

in the XML database. In this example, the article's title "Index Construction" will be returned as the exactly matched result.

To find all the exactly matched results, there are lots of previous methods [27, 7, 63, 95] designed for tree pattern queries in XML, which rely on the following three similar strategy - (1) *Decomposition*: decompose the tree pattern into linear patterns which might be binary (parent-child or ancestor-descendant) relationships between pairs of nodes or root-to-leaf paths; (2) *Matching*: find all matches of each linear pattern; and (3) *Merging*: merge the intermediated results to produce the final results.

However, the structures of such XML data are often too complex for users to fully grasp. In addition, due to the heterogeneous nature of large number of XML data sources, exact matching of queries is often inadequate. It has become increasingly important that the system can smartly modify users' queries to get satisfied answers from multiple XML data sources. To address this issue, there are four types of categories by modifying the users' original queries: (1) *Query Rewriting*: modified queries produce the same set of answers as original queries; (2) *Query Restriction*: less number of or smaller-scoped answers will be returned; (3) *Query Shift*: modified queries generate partially overlapping, but different set of answers; (4) *Query Relaxation*: more number of or bigger-scoped answers will be retrieved.

Among the four categories, *Query Rewriting* and *Query Relaxation* have drawn most of the attentions from research community, because both of them can guarantee the answers are lossless. But *Query Rewriting* requires users to offer their own schemas and set up mappings between their own schemas and those utilized in multiple sources, which is the foundation of query rewriting techniques. Unfortunately, it is often infeasible for a casual user to give out an entire schema, since his or her aim is only to issue a query. However, *Query Relaxation* can relax the users' queries by analysing the given queries and the corresponding data sources and evaluate the

relaxed queries to return the relevant results to the users, which does not ask for any other inputs from the users. Although the other two categories are also required in some cases, there are few work to discuss them due to the incomplete result set of *Query Restriction* and the varied result set of *Query Shift*. Compared with the other three categories, we think *Query Relaxation* is a more worthwhile and competent research topic in most cases.

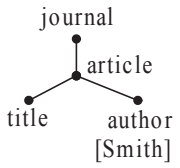


Figure 1.5: A different tree pattern query

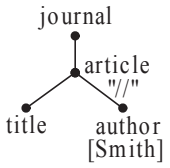


Figure 1.6: The example of relaxed query

For example, there is another user who intends to express the same query request over the same XML document in Figure 1.1. However, he or she issues a different tree pattern query as shown in Figure 1.5 because he or she does not know whether the node "*authors*" exists in the XML document. According to the decomposition-matching-merging process, no exact result can be returned from the document because the parent-child edge "*article*/*author*" does not appear in the document. To address this problem, query relaxation is required to generalize the parent-child edge "*article*/*author*" into ancestor-descendant edge "*article*//*author*" during query evaluation. The relaxed query is shown in Figure 1.6. Now, the same result - the article's title "*Index Construction*" would be returned as the matched result for the relaxed query. Most of the time, the wealth of XML data makes it difficult for users to identify relevant fragments for their query requests. Particularly, the loosely-coupled nature of the data sources also makes it inapplicable for deploying the traditional federated database approach for integrating the XML data sources by defining a global schema.

To address these issues, this thesis focuses on adaptive query processing techniques

to efficiently identify the fragments that are most relevant to user queries, preventing users from having to access all the XML data sources.

1.1 Motivation of This Thesis

Most existing work in [78, 58, 8] are proposed to relax a user's query depending on the intermediate feedback of query evaluation over XML documents, i.e., they do relaxation operations at running time. For instance, a query may be partitioned into a set of binary relationships at first step *decomposition*, based on the decomposition-matching-merging process strategy, and each binary relationship will be evaluated. If one binary relationship does not produce any result, the binary relationship would be required to be relaxed into a loosed relationship that has to be tested again. In addition, Amer-Yahia et al. in [10] propose a framework, called FleXPath, for relaxing XML tree pattern queries. Given a tree pattern query q, the closure of the structural and value-based predicates in q is first inferred and then is used to generate relaxed queries. The set of generated relaxed queries, including only the root of q, is complete. Based on the discussions, we can know that the above relaxation process is basically blind and wild, which may increase the number of relaxed queries greatly. For a large number of heterogeneous XML data sources, many of the relaxed queries could be unqualified, which will result in unnecessary cost of either computing or testing them.

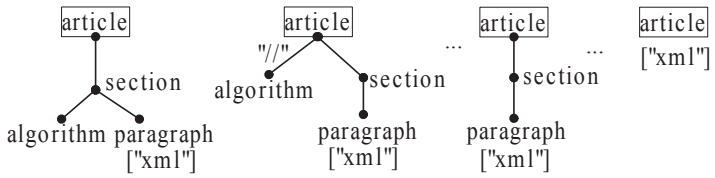
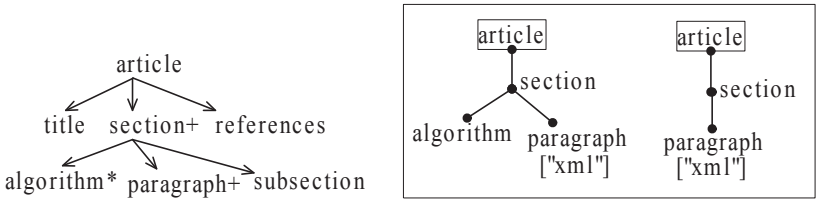


Figure 1.7: Relaxed queries of FleXPath

Suppose a user wishes to find *articles* that are relevant to *algorithms* on XML



(a) XML Schema (b) Relaxed Queries

Figure 1.8: Relaxed queries of our approach

data. At the beginning, the user may issue a precise structured query (the first one) as shown in Figure 1.7 where the node in a box is a distinguished node, indicating that matches of the node will be the required answers. FleXPath [10] may generate a great number of relaxed queries as shown in Figure 1.7, based on the original query. Here, we assume we do not consider term relaxation at this moment, i.e., we do not discuss whether the term "*xml*" is promoted to the nodes "*section*" or "*article*". However, if the XML document conforms to an XML schema as shown in Figure 1.8(a), only two of the large number of relaxed queries need to be generated and evaluated. The two qualified relaxed queries have been presented in Figure 1.8(b). The rest of the set of relaxed queries in Figure 1.7 are unqualified in structure with regards to the document.

To resolve the problem of unnecessary computation of a big number of unqualified relaxed queries, there are four challenges:

- **Challenge 1:** To enhance the quality of searching and ranking big volume of source documents, we need to study the problem of measuring structural similarities of a large number of source schemas against a single domain schema, which is different from previous methods that compute the similarity in a symmetric way.

- **Challenge 2:** Since previous query relaxation methods relax users' query

blindly, most of the time many of the relaxed queries generated are unqualified for a given XML document. Can we relax a user's query issued to a set of XML data sources adaptively?

- **Challenge 3:** To answer a top-k query over a set of heterogeneous XML data sources, it is time-consuming to evaluate all the relaxed queries over each data source and then select the top k results. Can we decide which data source is preferable to be processed as early as possible? Can we return the searched results incrementally during evaluation for a top-k query?

- **Challenge 4:** When issuing a structured query, users must be familiar with one of structured query languages, such as XPath or XQuery. Meanwhile, they need also know the structure of the XML data source to be retrieved. Otherwise, they either do not know how to express a structured query or issue a structured query with a lot of irrelevant information. For both of the cases, if we continue to deploy a query relaxation strategy, the relaxed queries may be very far away from the users' original intention because only a small part of information is preserved. Therefore, the practical usability of query relaxation will be affected greatly. Can we allow users issue and evaluate their requests in a loose-structured query, such as a set of label-term pairs?

We are motivated by the above challenges.

To address Challenge 1, an asymmetric similarity model is proposed to compare the structural features between a domain schema and the source schemas where we leverage the structural features of XML schema, such as ancestor-descendant ("//") relationship, parent-child ("/") relationship and cardinality ("*") information. The similarity model can be used to decide which data source may be more relevant to users' query request.

To address Challenge 2, we propose an adaptive relaxation approach that adaptively relaxes a query against different data sources based on their conformed schemas where all the inconsistent relationships are generalized or relaxed before query evaluation. For a data source, only the qualified relaxed queries need to be generated.

To address Challenge 3, we design a scheduling strategy to efficiently process top-k queries across heterogeneous XML data sources where the irrelevant candidates and data sources are able to be pruned as early as possible. Specifically, the results to be determined can be output incrementally. Therefore, users would see the most relevant results as soon as possible.

To address Challenge 4, it is desirable to allow users issue a set of label-term pairs in a keyword query, rather than a pure keyword query containing only terms. This is because if we permit users issue a pure keyword query, the possibility returning lots of uninteresting results to users may become very higher. Therefore, we allow a set of label-term pairs to be specified in a keyword query, which is helpful to express more semantics while limit the range of the terms under the nodes matching to the corresponding labels. In addition, to efficiently process a keyword query, we first construct structured queries w.r.t. XML data sources to be searched and then evaluate them in an effective strategy, which can obtain more relevant results and better performance than directly evaluating the keyword query.

1.2 Contributions of This Thesis

The main contributions of this thesis are as follows.

Firstly, we focus on the study of XML schema, which is helpful for enhancing the quality of searching and ranking a big volume of XML documents. The flexible structures of XML documents make query evaluation quite complex, and sometimes impose relaxed conditions and return approximate results. Due to different perception

of an application domain, different providers of a service may define different schemas which we call source schemas. Meanwhile users may issue their queries based on the common understanding of the domain which we call a domain schema. If one source schema is much more similar to the domain schema than the others, the probability of those XML documents that conform to the source schema will be much higher because the XML documents contain the most relevant results than the other XML documents. Therefore, to be able to get the most relevant results, it is very important to work out the similarity of the source schemas w.r.t. the domain schema. In Chapter 3, we design an asymmetric similarity model to compute the similarity of source XML schemas against domain XML schema. Particularly, we design similarity preserving rules and a trimming algorithm to filter out uninteresting elements or relationships in source schemas for the purpose of optimizing the similarity computation. In addition, an extension of Dietz's numbering schema is used to reduce the cost of computation.

Next, we concentrate on the study of XML query relaxation. As we know, there are some existing methods that can be applied to relax users' queries. But their relaxation is processed in a blind and wide way. That is to say, the previous approaches are required to generate all the relaxed queries in advance or adjust the inconsistent relationships based on the intermediate feedbacks during evaluation. Obviously, the former may generate many unqualified relaxed queries for a certain data source while the latter may make the query evaluation time-consuming. To address this issue, in Chapter 4 we propose an adaptive query relaxation approach, which adaptively relaxes a query to each XML data source according to its conformed schema. Hence each relaxed query will be guaranteed to agree with the structural constraints imposed by the conformed schema. And as a positive result, the adaptive relaxation approach will have a high probability of generating an answer from the XML data source. Particularly, a set of schema-aware relaxation rules and corresponding algorithms

are designed and presented. In addition, since the relaxed queries are derived from various relaxation operations subject to different source schemas, a penalty model is established for measuring to what extent the original query is modified. In this model, we allow users to convey their preference to the relationships between adjacent nodes in the query tree by assigning weights to edges. The computed penalty of a relaxed query can reflect the relevance of its matched results and the original query, which can be used to rank the retrieved answers.

Thirdly, we study top-k query evaluation when the given query cannot be answered exactly from the XML data sources or the exactly matched results are not enough for the users (i.e., the number of exactly matched results is less than the specified value k). Different from the traditional top-k approaches holding the similar assumption that all attributes are independent for each other, the top-k search issue over XML databases has to consider the relationships among the relevant elements (or attributes). Therefore, many traditional techniques cannot be directly applied to XML query evaluation. In Chapter 5 we propose a scheduling strategy to efficiently evaluate relaxed queries across heterogeneous XML data sources for answering top-k queries. By using the scheduling strategy, we can not only skip those XML data sources that will not produce the desired results, but also prune the intermediate results with lower relevance as early as possible. Most importantly, we can output the top-k results incrementally, rather than waiting for the end of all top-k results to be determined. Therefore, users would see the most relevant results as soon as possible. But we have to say that the above query relaxation techniques are the foundation of this scheduling strategy. This is because the weight of the relaxed query with the minimal penalty w.r.t. a data source can be taken as the upper bound of the relevance of the returned results from the data source. That is to say, the score of the most relevant results from the data source is lower than or equal to the upper bound value.

Finally, in Chapter 6 we present a novel keyword search approach that can construct a set of answer templates by analyzing the given keyword queries and the XML schemas conformed by XML data sources. Different from previous keyword search methods, our approach can prune plenty of irrelevant intermediate results as early as possible. In addition, answer templates can be used to design efficient execution plans. Specifically, we propose a ranking function to measure the relevance between the results and a keyword query by considering the context and the weights of the keywords. For efficiently finding the top-k relevant results, we also develop a corresponding algorithm for adapting to our proposed ranking function.

1.3 Structure of This Thesis

The structure of this thesis is organized as follows.

- **Chapter** 2 gives the literature review in which we discuss the significant models or algorithms related to our work.
- **Chapter** 3 introduces a model to compute structural similarity of source XML scheams against domain XML schema, which can rank the data sources for a given domain schema. The ranking results can be used to guide the query evaluation across a heterogeneous XML data sources.
- **Chapter** 4 presents an adaptive relaxation approach for query heterogeneous XML data sources, which can avoid the blind and wild relaxations in previous query relaxation methods.
- **Chapter** 5 discusses two scheduling strategies: brute-force scheduling strategy and BT-based scheduling strategy to process top-k queries, which can efficiently find the top k relevant answers by pruning the data sources and irrelevant intermediate results as early as possible.

- **Chapter** 6 introduces a precise keyword search model, which can improve the efficiency and accuracy of XML keyword search.
- **Chapter** 7 concludes the thesis with a discussion of the future work.

Chapter 2

Related Work

There has been a lot of interesting work in the field of query relaxation and processing over XML data recently. In this chapter, we first summarize the work about structural similarity algorithms for ranking XML data sources to be retrieved in Section 2.1. And then, we discuss the related research about XML structured query relaxation and XML keyword query processing in Section 2.2 and Section 2.3, respectively. Finally, we introdce the related work for processing top-k queries in different applications in Section 2.4.

2.1 Similarity Computation Measures

Measuring the structural similarity among XML documents has been an active area of research in the past a few years and it is fundamental to many applications, such as integrating XML data sources, XML dissemination and repositories. In this section, we discuss previous work of identifying similarity between XML documents, between XML documents and schemas and between XML schemas in Section 2.1.1, Section 2.1.2 and Section 2.1.3, respectively.

2.1.1 Similarity Computation between XML Documents

There is considerable previous work on finding edit distances between XML trees [23, 24, 25, 82, 84, 88, 96]. Most algorithms in this category are direct descendants of the dynamic programming techniques for finding the edit distance between strings. The basic idea in all of these tree edit distance algorithms is to find the cheapest sequence of edit operations that can transform one tree into another.

A key differentiator between the various tree-distance algorithms is the set of edit operations allowed. An early work in this area is by Selkow [82], which allows inserting and deleting of single nodes at the leaves, and relabeling of nodes anywhere in the tree. The work by Chawathe in [23] utilizes these same edit operations and restrictions, but is targeted for situations when external memory is needed to calculate the edit distance. There are several other approaches that allow insertion and deletion of single nodes anywhere within a tree [84, 88, 96]. Expanding upon these more basic operators, Chawathe et. al. [25] and Cobena et al. [29] also considered a move operation on sub-trees, which is essential in the context of XML. Differently Cobena et al. used signatures to match sub-trees that are left unchanged between the old and new versions.

General Tree-Edit Distance: Assume that we are given a cost function defined on each edit operation. An edit script S between two trees T and T' is a sequence of edit operations turning T into T'. The cost of S is the sum of the costs of the operations in S. An optimal edit script between T and T' is an edit script between T and T' of minimum cost. This cost is called the *tree edit distance*, denoted by $\delta(T, T')$. For example, the mapping in Figure 2.1 shows a way to transform T to T'. The transformation includes deleting node labeled b in T and inserting node labeled f in T'.

An edit distance mapping (or just a mapping) between T and T' is a representation of the edit operations, which is used in many of the algorithms for the tree edit

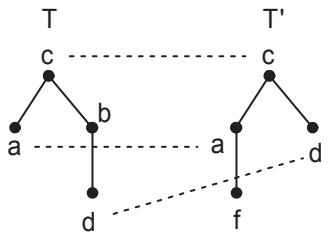


Figure 2.1: A mapping from T to T'.

distance problem. In [96], Zhang et. al. used a postorder numbering to encode the nodes in the trees. Let $T[i]$ represent a node of T whose position in the postorder for the nodes of T is i. When there is no confusion, they also used $T[i]$ to represent the label of node $T[i]$. Formally, a mapping from T to T' is a triple (M, T, T') (or simply M if the context is clear), where M is any set of pairs of integers (i, j) satisfying: (Map Conditions)

1. $1 \leq i \leq |T|$, $1 \leq j \leq |T'|$;

2. For any pair of (i_1, j_1) and (i_2, j_2) in M,

   (a) $i_1 = i_2$ if and only if $j_1 = j_2$ (one-to-one);

   (b) $T[i_1]$ is to the left of $T[i_2]$ if and only if $T'[j_1]$ is to the left of $T'[j_2]$ (sibling order preserved);

   (c) $T[i_1]$ is an ancestor of $T[i_2]$ if and only if $T'[j_1]$ is an ancestor of $T'[j_2]$ (ancestor order preserved).

Let (M, T, T') be a mapping. It says that a node v in T or T' is touched by a line in M if v occurs in some pairs in M. Let N_1 and N_2 be the set of nodes in T and T' respectively not touched by any line in M. The cost of M is given by:

$$\gamma(M) = \sum_{(\upsilon,\omega)\in M} \gamma(\upsilon \to \omega) + \sum_{\upsilon in N_1} \gamma(\upsilon \to \lambda) + \sum_{\omega \in N2} \gamma(\lambda \to \omega)$$

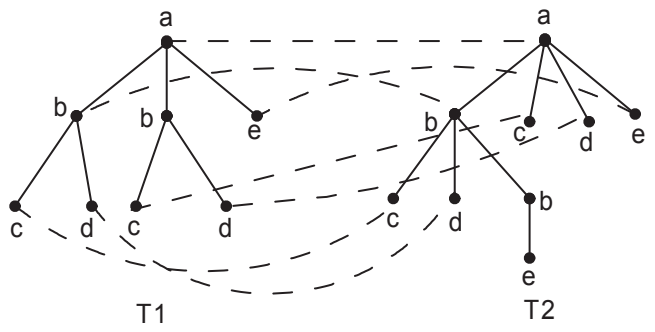


Figure 2.2: Tree Examples

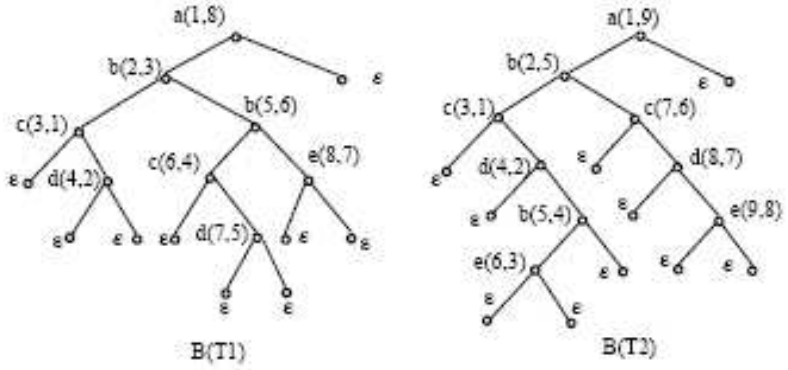


Figure 2.3: Normalized Binary Tree Representation

Since γ is a metric, it is not hard to show that a minimum cost mapping is equivalent to the edit distance: $\delta(T, T') = min\{\gamma(M) \mid (M,T,T')$ is an edit distance mapping$\}$.

Binary Tree-based Edit Distance: Yang et al. in [91] transformed tree-structured data into corresponding binary trees, and then encoded the binary trees and generated the corresponding approximate numerical multidimensional vectors to compute similarity, denoted as *BDist*.

There is a natural correspondence between forests and binary trees. The standard algorithm to transform a forest (or a tree) to its corresponding binary tree is through the left-child, right-sibling representation of the forest (tree): (1) Link all the siblings in the tree with edges. (2) Delete all the edges between each node and its children

in the tree except those edges which connect it with its first child. Note that the transformation does not change the labels of vertices in the tree. For example, the trees T_1 and T_2 of Figure 2.2 can be transformed into $B(T_1)$ and $B(T_2)$ shown in Figure 2.3, respectively. To generate the vectors, they first build an inverted file for all binary branches, as shown in Figure 2.4(a). An inverted file has two main parts: a vocabulary which stores all distinct values being indexed, and an inverted list for each distinct value which stores the identifiers of the records containing the value. The vocabulary here consists of all existing binary branches in the datasets. The inverted list of each component records the number of occurrences of it in the corresponding trees. The resulting vectors of the transformation for the trees in Figure 2.2 and the normalized binary trees in Figure 2.3 are shown in Figure 2.4(b). Based on the vector representation, the binary branch distance can be computed by the equation $\mathrm{BDist}(T_1, T_2) = \sum_{i=1}^{|T|} |b_i - b'_i| = 9$, which can be taken as the lower bound edit distance of the two trees.

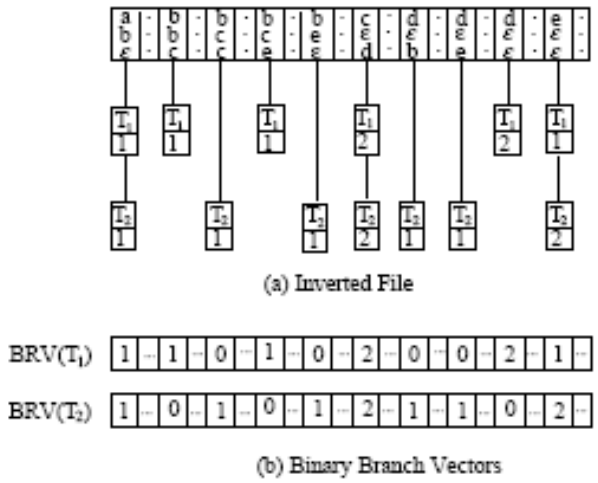


Figure 2.4: Binary Branch Vector Representation

However, when $BDist(T_1, T_2) = 0$, it can not imply that T_1 is identical to T_2. This

is illustrated in Figure 2.5, where both trees T_1 and T_2 have the same binary branch tree.

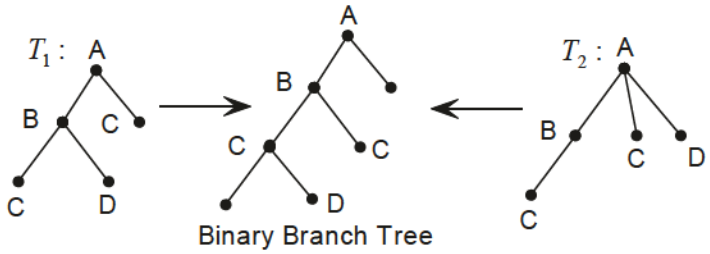


Figure 2.5: Trees with 0 Binary Branch Distance

Although the binary branch distance is not a metric, it approximates and lower bounds the tree-edit distance metric. If the lower bound distance is greater than the query range, it is safe to filter out the data since its real edit distance cannot be less than that range, i.e., objects that cannot qualify are filtered out while verification of the original complex similarity measure is necessary only for the candidates filtered through. Therefore, they can efficiently process similarity search on the tree-structured data by exploiting the lower bounds.

Time Series-based Edit Distance: In order to compute the similarity between XML documents, Flesca et al. in [45] described the structure of an XML document into a time series where each occurrence of a tag corresponds to a given impulse. By analysing the frequencies of the corresponding Fourier transform [74], they can state that the degree of similarity between the documents. As a matter of fact, the exploitation of the Fourier transform to check similarities among time series is not completely new (see, e.g., [5, 75]) and has been proved successfully. The main contribution of this work is the systematic development of an effective encoding scheme for XML documents, in a way that makes the use of the Fourier Transform extremely profitable.

2.1.2 Similarity Computation between Documents and Schemas

Bertino et al. [12] exploited a graph-matching algorithm to associate elements in the XML document with element definitions in the DTD. An algorithm, named $Match$, is proposed to evaluate the similarity between any kind of XML documents and DTDs. Given a document D, and a DTD T, algorithm $Match$ first checks whether the root labels of the two trees are equal. If not, then the two structures do not have common parts. If the root labels are equal, the maximal level l between the levels of the two structures is determined, and the recursive function M is called on:

1. the root of the document,
2. the first (and only) child of the DTD,
3. the level weight taking into account that function M is called on the second level of the DTD structure, and
4. a flag indicating that the current element (the root element) is not repeatable.

Function M recursively visits the document and the DTD, at the same time, from the root to the leaves, to match common elements.

2.1.3 Similarity Computation Between Schemas

At the schema level, different methods computing XML schema similarity have been studied for the purpose of generating qualified schema matching.

Cupid [68] addressed the problem of schema matching by computing similarity coefficients between elements of the two schemas and then deducing a mapping from those coefficients. The coefficients, in the [0, 1] range, are calculated in two phases.

1. linguistic matching: It matches individual schema elements based on their names, data types, domains, etc. They use a thesaurus to help match names

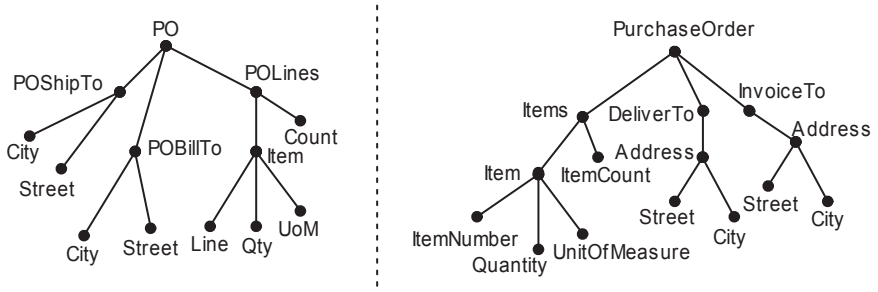


Figure 2.6: Purchase Order Schemas

by identifying short-forms (*Qty* for *Quantity*), acronyms (*UoM* for *UnitOfMeasure*) and synonyms (*Bill* and *Invoice*). The result is a linguistic similarity coefficient, *lsim*, between each pair of elements.

2. structural matching: It matches the schema elements based on the similarity of their contexts or vicinities. Let's take the *purchase order schemas* in Figure 2.6 as an example. *Line* is mapped to *ItemNumber* because their parents, *Item*, match and the other two children of *Item* already match. The structural match depends in part on linguistic matches calculated in phase one. For example, *City* and *Street* under *POBillTo* match *City* and *Street* under *InvoiceTo*, rather than under *DeliverTo*, because *Bill* is a synonym of *Invoice* but not of *Deliver*. The result is a structural similarity coefficient, *ssim*, for each pair of elements.

The weighted similarity (*wsim*) is a mean of *lsim* and *ssim*: $wsim = w_{struct} \times ssim + (1\text{-}w_{struct}) \times lsim$, where the constant w_{struct} is in the range 0 to1. A mapping is created by choosing pairs of schema elements with maximal weighted similarity. In order to improve the performance of computation, its structural algorithm only considers the elements with the same number of leaves and immediate descendants.

The similarity comparison in XClust [60] and Similarity Flooding [72] considers not only the linguistic and structural information of schema elements but also the context of a schema element (defined by its ancestors and descendants in a schema tree).

Differently, XClust also considered the cardinality of elements. Similarity Flooding employed fixpoint computation approach to detect the similar schema elements. In addition, Similarity Flooding is usable across different scenarios. Yi and Weng et. al. [93] represented XML schemas by constructing a universal semantic model and then compared the generated models to compute the similarity between XML schemas. The semantic model consisted of three components: ontological categories, properties and contextual constraints. In addition, the operation of relaxation labeling was devised to improve the quality of schema matching. Formica [46] computed the similarity of XML schema elements by combining two parts: the maximum information content that is measured by the minimal common type to be shared and their own type declarations. COMA [38] is a generic schema match system which provided an extensible library of simple and hybrid match algorithms and supported a framework for combining the match results obtained from different algorithms. In the system, the similarity between the elements was recursively computed from the similarity between their respective children with a leaf-level matcher.

To support efficient computation of schema similarity, Duta [39] proposed seven reduction rules to transform XML schemas into minimum structures capable of storing the same information and preserved the cardinality constraints of leaf nodes. The first three rules called *conversion rules* convert the attribute and text node types of the source structure into an element node, and the last four rules called *elimination rules* eliminate intermediate tree levels. Rahm and Bernstein [76] did a comprehensive survey of some early work in schema matching.

2.2 Structured Query Relaxation

Query relaxations on structure have been studied recently. In this section, we first describe the strategy that embeds the relaxation operations in query evaluation plan

in Section 2.2.1. And then we illustrate the procedure of relaxation by deriving the correlations from malleable schemas in Section 2.2.2. Finally, Section 2.2.3 briefly introduce the rest methods related to query relaxation.

2.2.1 Embedding Relaxation in Query Evaluation Plan

Amer-Yahia et al. [8] proposed the relaxations of weighted tree pattern queries: generalizing nodes or edges, deleting leaf nodes, promoting subtrees. Weights are added to either edges and nodes for determining penalties in a straightforward manner. Let us consider the example in Figure 2.7.

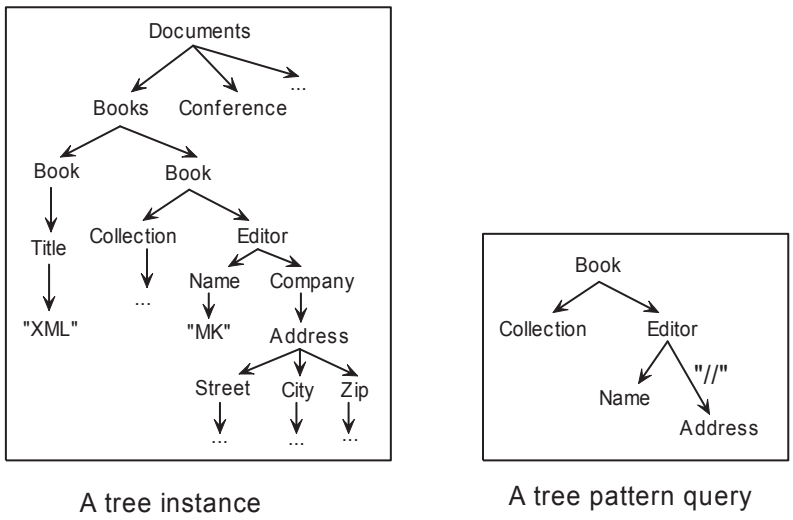


Figure 2.7: An Instance and a Query Example

In order to evaluate structured query, several query evaluation strategies have been proposed for XML (e.g. [71, 95]). They typically rely on a combination of index retrieval and join algorithms using specific structural predicates. In the work [8], Amer-Yahia et al. used the algorithm of [95] and two specific predicates $c(n_1, n_2)$ to check for the parent-child relationship and $d(n_1, n_2)$ to check for the ancestor-descendant one. Figure 2.8 shows a translation of the tree pattern query of Figure 2.7

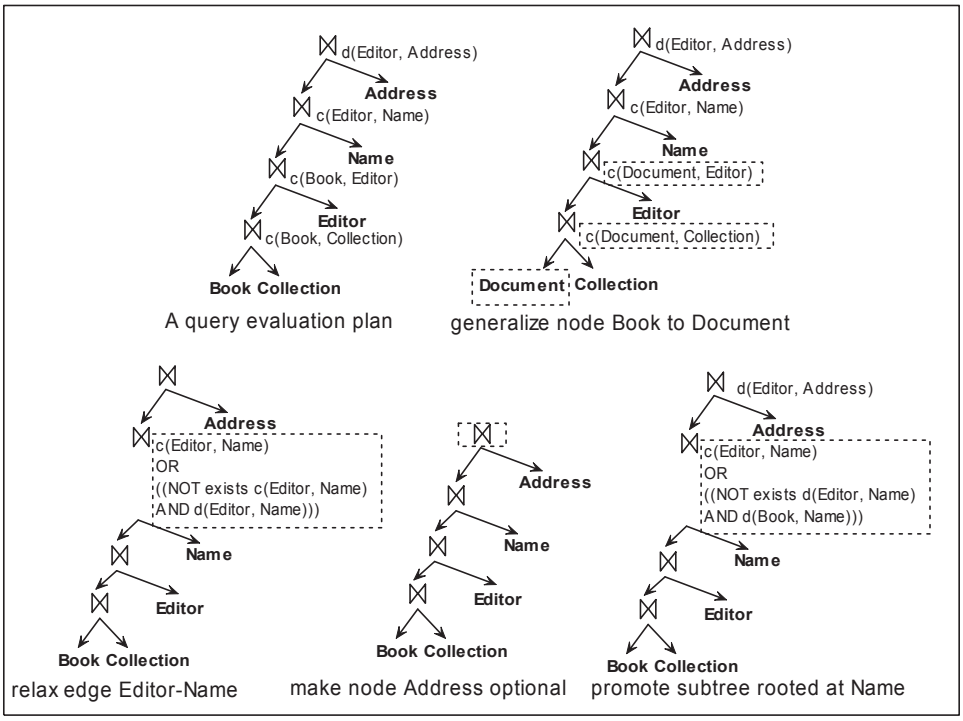


Figure 2.8: An example of relaxation embedded in evaluation plan

into a left-deep join evaluation plan with the appropriate predicates. And some relaxations of the query are embedded in the query evaluation plan.

Generalizing nodes. In order to encode a node type generalization in a query plan, the predicate on the node type is replaced by a predicate on its super-type. We show how *Book* can be replaced by *Document* in Figure 2.8.

Relaxing the structural predicate between nodes. In order to capture the generalization of a parent-child edge to an ancestor-descendant edge in an evaluation plan, we transform the join predicate c(t_1, t_2) into the predicate: c(t_1, t_2) OR ((not exists c(t_1, t_2)) AND d(t_1, t_2)).

This new join predicate can be checked by first determining if a parent-child relationship exists between the two nodes, and then, if this relationship does not exist, determining if an ancestor-descendant relationship exists between them. For example,

Figure 2.8 depicts how the parent-child edge (Editor, Name) can be generalized to an ancestor-descendant edge in the evaluation plan.

Making a leaf node optional. To allow for the possibility that a given query leaf node may or may not be matched, the join that relates the leaf node to its parent node in the query evaluation plan becomes an outer join. More specifically, it becomes a left outer join for left-deep evaluation plans. For example, Figure 2.8 illustrates how the evaluation plan is affected by allowing the *Address* node to be deleted. The left outer join guarantees that even the books whose editor does not have an address will be returned as an approximate answer.

Promoting a subtree. This relaxation causes a query subtree to be promoted to become a descendant of its current grandparent. In the query evaluation plan, the join predicate between the parent of the subtree and the root of the subtree, say jp(t_1, t_2) needs to be modified to: jp(t_1, t_2) OR ((not exists jp(t_1, t_2)) AND d(t_3, t_2)) where t_3 is the type of the grandparent. For example, Figure 2.8 illustrates the evaluation plan is affected by promoting the subtree rooted at *Name*.

Combining relaxations. It consists of the above four relaxation operations. But the root node and the original descendant edges do not need to be relaxed.

2.2.2 Deploying Malleable Schemas

Recently, Zhou et. al. in [97] proposed query relaxation by using malleable schemas where they first utilize the duplicates in different data sets to discover the correlations within a malleable schema and then relax users' queries based on the derived correlations.

Since each data instance uses only a subset of the attributes or relationships defined in a malleable schema, the predicates in a query have to be properly relaxed to retrieve all relevant results. In this work, such relaxation is achieved by extending the types of attributes or relationships. For example, Given a query q_1 based on one

schema (please refer to Figure 1 in [97]).

$q_1 = \{E1|E1.title \ni$ 'XML' $\wedge E1.title \ni$ 'Query' $\wedge E1.ISA - paper \ni$ 'True' $\wedge E1.author \ni E2 \wedge E2.name \ni$ 'Daniel'$\}$

In order to retrieve more relevant results, q_1 can be relaxed by extending $E2.name \ni$ 'Daniel' to $E2.firstname \ni$ 'Daniel'. By applying query relaxation, a query will be turned into a set of relaxed queries. However, users would like to see the results with the higher relevance. Therefore, the system in this work returns query results based on their probabilities of relevance. That means, given a query q_0 that could be relaxed to $q_1 \vee q_2 \vee \ldots \vee q_n$, the system returns the results of the relaxed queries according to the probabilities $P(q_0|q_1)$, $P(q_0|q_2)$, ..., $P(q_0|q_n)$, where $P(q_i|q_j)$ represents the probability that a result of q_j is also a relevant result of q_i. As an example, $q_0 = \{E|A \ni a \wedge B \ni b\}$ is relaxed to $q_1 \vee q_2$, where $q_1 = \{E|A_1 \ni a \wedge B_1 \ni b\}$ and $q_2 = \{E|A_2 \ni a \wedge B_2 \ni b\}$. If the system can know that $P(A \ni a \wedge B \ni b|A_1 \ni a \wedge B_1 \ni b) < P(A \ni a \wedge B \ni b|A_2 \ni a \wedge B_2 \ni b)$, then the system will return the results of q_2 prior to the results of q_1 because q_2 will retrieve more relevant results than q_1.

The authors assume that the entity-relationship data are stored in relational database. In contrast to ordinary queries in relational database, a query using malleable schema will be relaxed to multiple queries that are executed on different columns or tables. The major performance consideration is to find a plan that executes as less queries as possible to retrieve sufficient relevant results. The optimal plan is to execute relaxed queries in a sequence based on the expected precisions of their result sets. However, sometimes it is infeasible to evaluate all relaxed queries. In practice, it is more desirable to evaluate the relaxed queries in the order of their precisions until the users can obtain more than k results or the users are satisfied and stop the processing. To achieve this, the authors exploit the relationship between the relaxed queries. And they find that a relaxed query always yields better precision

than its child queries, so that it should always be evaluated prior to its child queries.

2.2.3 Others

In addition, there are some approaches that relax queries when no results can be returned. Delobel and Rousset [35] defined three kinds of relaxations: unfolding a node (replicating a node by creating a separate path to one of its children), deleting a condition at a node, and propagating a condition to its parent node. Schlieder[78] considered relaxations on an XQL query: deleting nodes for making the context loose, inserting a node between inner nodes for specifying more specific context and renaming nodes for changing the search context. Koudas et al.[58] presented a framework for relaxing join and selection conditions in relational schema. In information retrieval, some work have been proposed to search the approximate results. For example, VCAS [66] is an approach for vague content and structure retrieval, which partitions a user's structure query into a SCAS (strict content and structure) sub query and a CO (content only) sub query and produces results by combining the results of two sub queries.

2.3 XML Keyword Search Processing

Keyword search is a user-friendly way of querying HTML documents in the World Wide Web. Keyword search is well-suited to XML documents as well, which are often modeled as labeled trees or graphs. It allows users to find the information they are interested without having to learn a complex query language or needing prior knowledge of the structure of the underlying data. In this section, we first introduce some approaches that take the lowest common ancestors (LCA) or smallest LCA (SLCA) of the keyword-matched nodes as the returned results. And then we describe the more effective methods that pay more attention to the semantic relationships

among the keyword-matched nodes, which can improve the accuracy of the searched results. At last, we illustrate some specific keyword search methods over data graph, which will return a set of subgraphs or particular representatives as the results.

2.3.1 LCA/SLCA based Approaches

Definition 1 ***LCA in XML*** *Given a list of m keywords k_1, k_2, $\cdots$, k_m and an input XML document tree T, an answer subtree of keywords k_1, k_2, $\cdots$, k_m is a subtree of T such that it contains at least one instance of each keyword and at least one of the instances cannot be covered by other answer subtrees. The results $LCA(k_1, k_2, \cdots, k_m, T)$ of the list of keywords k_1, k_2, $\cdots$, k_m on the input XML document tree T is the set of roots of all answer subtrees of the list of keywords.*

Definition 2 ***SLCA in XML*** *A smallest answer subtree of keywords k_1, k_2, $\cdots$, k_m is answer subtree (of keywords k_1, k_2, $\cdots$, k_m) such that none of its subtrees is an answer subtree (of keywords k_1, k_2, $\cdots$, k_m). The results $SLCA(k_1, k_2, \cdots, k_m, T)$ of the list of keywords k_1, k_2, $\cdots$, k_m on the input XML document tree T is the set of roots of all smallest answer subtrees of the list of keywords.*

Guo et. al. in [49] presented the XRANK system to process ranked keyword search over XML documents. The keyword query results are defined as LCA. In addition, to make the returned results more meaningful, XRANK also supported both user navigation for context information and the ability to pre-define answers nodes where the former is to allow the user to navigate up to the ancestors of the query result to get more context information when desired and the latter is to predefine a set of "answer nodes" (that is originally proposed in the context of keyword searching graph database [15, 33]). But there is an assumption that pre-defining answer nodes for XML documents may require knowledge of the domain and underlying XML schema, if such knowledge is not available, all XML elements can be treated as answer nodes.

To sort the retrieved results, three properties are considered in their ranking function:

- **Result specificity**: The ranking function should rank more specific results higher than less specific results.
- **Keyword proximity**: The ranking function should take the proximity of the query keywords into account. This is the other dimension of result proximity. Note that a result can have high keyword proximity and low specificity, and vice-versa.
- **Hyperlink Awareness**: The ranking function should use the hyperlinked structure of XML documents.

To efficiently evaluate XML keyword search queries, they proposed three algorithms: Dewey Inverted List (DIL) algorithm, Ranked Dewey Inverted List (RDIL) and Hybrid Dewey Inverted List (HDIL).

The key idea of DIL is to merge the query keyword inverted lists, and simultaneously compute the longest common prefix of the Dewey IDs in the different lists. Since each prefix of a Dewey ID is the ID of an ancestor, computing the longest common prefix will automatically compute the ID of the deepest ancestor that contains the query keywords. Since the inverted lists are sorted on the Dewey ID, all the common ancestors are clustered together, and this computation can be done in a single pass over the inverted lists.

Although DIL evaluates queries in a single pass over the query inverted lists, it suffers from a potential disadvantage. If inverted lists are long (due to common keywords or large document collections), even the cost of a single scan of the inverted lists can be expensive, especially if users want only the top few results. One solution is to order the inverted lists by the ElemRank (element ranking score) instead of

by the Dewey ID. In this way, higher ranked results are likely to appear first in the inverted lists, and query processing can usually be terminated without scanning all of the inverted lists. As a simple example, if a query contains just one keyword, only the first m inverted list entries have to be scanned to find the top m results.

Even though RDIL is likely to perform well in many cases, there are certain cases where it is likely to perform much worse than DIL. For example, consider a query where the keywords are not very correlated, i.e., the individual query keywords occur relatively frequently in the document collection but rarely occur together in the same document. Since the number of results is small, RDIL has to scan most (or all) of the inverted lists to produce the output, incurring the cost of random index lookups along the way. In contrast, DIL sequentially scans the inverted lists, and is likely to be faster. In general, the overhead of performing random index lookups in RDIL can sometimes outweigh the benefit of processing the inverted lists in rank order. To address this problem, they considered an adaptive strategy. They first started evaluating the query using RDIL, and periodically monitor its performance to calculate (a) the time spent so far - t, and (b) the number of results above the threshold so far - r. Based on this, they estimate the remaining time for RDIL as $(m - r) * t/r$, where m is the desired number of query results. If this estimated time is more than the expected time for DIL, we switch to DIL. Note that the expected time for DIL is relatively easy to compute a priori for a given machine configuration because it mainly depends on the number of query keywords, and the size of each query keyword inverted list (since DIL scans inverted lists fully in all cases). If there are very few results above the threshold (corresponding to low keyword correlation), it switches to DIL; else it sticks with RDIL.

Xu and Papakonstantinou in [90] presented the XKSearch system to process keyword search in XML databases. The proposed keyword search returned a set of SLCA of the keyword matches as the interesting results. But in their work, how to rank the

returned SLCA nodes is not discussed.

To efficiently compute the SLCAs of keyword matches, they proposed two algorithms: the Indexed Lookup Eager algorithm that exploits key properties of smallest trees in order to outperform prior algorithms by orders of magnitude when the query contains keywords with significantly different frequencies and the Scan Eager variant algorithm that is tuned for the case where the keywords have similar frequencies.

The Indexed Lookup Eager Algorithm improves the algorithm by adding "eagerness" - it returns the first part of the answers without having to completely go through any of the keyword lists and it pipelines the delivery of SLCAs. Assume there is a memory buffer size of P nodes. The Indexed Lookup Eager algorithm first computes $X_2 = slca(X_1, S_2)$ where X_1 is the first P nodes of S_1. Then it computes $X_3 = slca(X_2, S_3)$ and so on, until it computes $X_k = slca(\ldots slca(X_1, S_1) \ldots S_k)$. All nodes in X_k except the last node are guaranteed to be SLCAs and are returned. The last node of X_k is carried on to the next operation to be determined whether it is a SLCA or not. The above operation is repeated for the next P nodes of S_1 until all nodes in S_1 have been processed. The smaller P is, the faster the algorithm produces the first SLCA. If $P = 1$, again only three nodes are needed to be kept in memory in the whole process. However, a smaller P may delay the computation of all SLCAs when considering disk accesses.

When the occurrences of keywords do not differ significantly, the total cost of finding matches by lookups may exceed the total cost of finding matches by scanning the keyword lists. Therefore, they implement a variant of the Indexed Lookup Eager Algorithm, named Scan Eager Algorithm, to take advantage of the fact that the accesses to any keyword list are strictly in increasing order in the Indexed Lookup Eager algorithm. The Scan Eager algorithm is exactly the same as the Indexed Lookup Eager algorithm except that its lm and rm implementations scan keyword lists to find matches by maintaining a cursor for each keyword list. In order to find the

left and right match of a given node with id p in a list S_j, the Scan Eager algorithm advances the cursor of S_j until it finds the node that is closest to p from the left or the right side. Notice that nodes from different lists may not be accessed in order, though nodes from the same list are accessed in order.

There are lots of other related work [83, 65, 67] to evaluate XML keyword query by computing LCA or SLCA of the keyword matches. Sun et. al. in [83] proposed a multiway-SLCA approach to process SLCA-based keyword search queries. The multiway-SLCA approach computes each potential SLCA by taking one data node from each keyword list S_i in a single step instead of breaking the SLCA computation into a series of intermediate binary SLCA computations. On the other hand, the approach picks an "anchor" node from the k keyword data lists to drive the multiway SLCA computation. By doing so, it is able to optimize the selection of the anchor node to maximize the skipping of redundant computations. Li et. al. in [65] integrated keywords into standard XML Query to process keyword search in XML where searching the keyword matches can be constrained by associating more specific context with keywords. Liu and Chen in [67] focused on the procedure of identifying meaningful return information for XML keyword search where it is possible to return the whole entities (SLCA) or attributes related to the keyword matches as the interesting results.

2.3.2 Semantics based Approaches

LCA or SLCA based approaches determine the results by considering the structural relationships among the keyword-matched nodes. However, to retrieve the more meaningful results, there are some other work that investigate under what conditions different elements of an XML document are semantically related.

Li et. al. in [61] introduced the notion of valuable lowest common ancestor to accurately and effectively answer keyword queries over XML documents, which not

only improves the accuracy of LCAs by eliminating redundant LCAs that should not contribute to the answers, but also retrieves the false negatives filtered out wrongly by SLCAs.

Cohen et. al. in [30] presented a semantic search engine for XML by studying the semantic relationships between any two nodes of an XML data tree. They formalize the idea as follows. Let n and n' be nodes in an XML data tree T. It can say that n and n' are interconnected if one of the following conditions holds:

- Any two distinct nodes with the same label do not exist on the "path" between n and n', or
- The only two distinct nodes on the "path" with the same label are n and n'.

Therefore, only the interconnection relationship between nodes can be considered in their work. Let $Q(t_1, \cdots, t_m)$ be a query. It says that a sequence $N = n_1, \cdots, n_m$ of nodes is an answer for the query Q if the nodes in N satisfy the interconnection relationship and for all $1 \leq i \leq m$:

1. n_i is not the null value if t_i is a required term;
2. n_i satisfies t_i if it is not the null value.

In order to return the answers in order of relevance, they use the vector space model, common in information retrieval, when determining how well an answer satisfies a query. Intuitively, the measure of similarity between a query Q and an answer A, denoted $sim(Q, A)$, is the sum of the cosine distances between the vectors associated with the nodes in A and the vectors associated with the terms that they match in Q. In addition, the relationships between nodes in A are also required to be considered.

During query evaluation, path index is designed to efficiently match the labels in the queries. Moreover, dynamic offline/online interconnection indices are proposed

to improve the performance of detecting the relationships between any two nodes in the answers.

2.3.3 Keyword Search over Graph

He et. al. in [50] proposed a system BLINKS that explored a single- or bi-level indexing and query processing scheme for top-k keyword search on graphs.

A Single-Level Index A common approach to enhance online performance is to perform some offline computation. They pre-compute, for each keyword, the shortest distances from every node to the keyword (or, more precisely, to any node containing this keyword) in the data graph. The result is a collection of *keyword-node lists*. For a keyword w, $L_{KN}(w)$ denotes the list of nodes that can reach keyword w, and these nodes are ordered by their distances to w. They also pre-compute, for each node u, the shortest graph distance from u to every keyword, and organize this information in a hash table called *node-keyword map*, denoted M_{NK}. Given a node u and a keyword w, $M_{NK}(u, w)$ returns the shortest distance from u to w, or ∞ if u cannot reach any node that contains w.

They call the duo of keyword-node lists and node-keyword map a *single-level index* because the index is defined over the entire data graph. Therefore, we can find the answers faster by augment backward search with efficient forward expansion.

Bi-Level Index For large data graphs, the single-level index is impractical because the index is too large to store and too expensive to construct. To address this problem, the BLINKS uses a divide-and-conquer approach to create a bi-level index. BLINKS partitions a data graph into multiple subgraphs, or blocks. A bi-level index consists of a top-level *block index*, which stores the mapping between keywords and nodes to blocks, and an *intrablock index* for each block, which stores more detailed information within a block. The total size of the bi-level index is a fraction of that of a single-level index.

The block index is a simple data structure consisting of:

- For each keyword w, $L_{KB}(w)$ denotes the list of blocks containing keyword w, i.e., at least one node in the block is labeled with w.
- For each portal p, $L_{PB}(p)$ denotes the list of blocks with p as an out-portal.

The keyword-block lists are used by the search algorithm to start backward expansion in relevant blocks. The portal-block lists are used by the search algorithm to guide backward expansion across blocks. Note that with the portal-block lists, it is not necessary for each node to remember which block it belongs to; during backward expansion it should always be clear what the current block is.

For each block b, the intra-block index consists of the following data structures:

- Intra-block keyword-node lists: For each keyword w, $L_{KN}(b, w)$ denotes the list of nodes in b that can reach w without leaving b, sorted according to their shortest distances (within b) to w (or more precisely, any node in b containing w).
- Intra-block node-keyword map: Looking up a node $u \in b$ together with a keyword w in this hash map returns $M_{NK}(b, u, w)$, the shortest distance (within b) from u to w (∞ if u cannot reach w in b).
- Intra-block portal-node lists: For each out-portal p of b, $L_{PN}(b, p)$ denotes the list of nodes in b that can reach p without leaving b, sorted according to shortest distances (within b) to p.
- Intra-block node-portal distance map: Looking up a node $u \in b$ in this hash map returns $D_{NP}(b, u)$, the shortest distance (in b) from a node u to the closest out-portal of b (∞ if u cannot reach any out-portal of b).

Another different but related research topic is keyword search in relational databases that is also viewed as a graph of objects with edges representing relationships between the objects. DBXplorer [6], DISCOVER [53], Hristidis et al. [51] and BANKS [15] are systems that support free-form keyword search on relational databases. They return tuple trees as answers for a given keyword query. One focus of the above works is to generate tuple trees efficiently. DBXplorer, DISCOVER and Hristidis et al. construct a set of join expressions (i.e. answer graph) for a given query, and then evaluate these join expressions to produce tuple trees. BANKS finds all tuple trees from the data graph directly using a Steiner tree algorithm. In the data graph, they use PageRank style methods to assign weights to tuples and assign weights to edges between tuples. XKeyword [54] stores the XML data in a relational database and delivers much higher efficiency than the above systems, which perform keyword search on arbitrary graphs, by being tuned for SLCA keyword search on trees. More recently, Sayyadian et al [77] introduced schema mapping into keyword search and proposed a method to answer keyword search across heterogenous relational databases. [56, 50] studied the problem of keyword search over graphs by employing the techniques of bidirectional expansion and graph partition respectively.

2.4 Top-k Queries Processing

In this section, we first present the traditional top-k approaches in relational database in Section 2.4.1 and then show an adaptive top-k approach over XML data in Section 2.4.2.

2.4.1 Traditional Top-k Algorithms

Assume that there are m attributes, and that the aggregation function is the m-ary function t. If $x_1, \cdots, x_m$ are the grades of object R under each of the m attributes,

then $t(x_1, \cdots, x_m)$ is the (overall) grade of object R. They use $t(R)$ for the grade $t(x_1, \cdots, x_m)$ of R. We say that an aggregation function t is monotone if $t(x_1, \cdots, x_m) \leq t(x'_1, \cdots, x'_m)$ whenever $x_i \leq x'_i$ for every i. Certainly monotonicity is a reasonable property to demand of an aggregation function: if for every attribute, the grade of object R' is at least as high as that of object R, then we would expect the overall grade of R' to be at least as high as that of R. They restrict their attention to monotone aggregation functions. Their goal is to find the top k objects for some fixed choice of k, that is, the k objects R with the highest overall grades $t(R)$.

Fagin's algorithm (FA) [40] works as follows:

- Do sorted access in parallel to each of the m sorted lists L_i. (By "in parallel", it means that they access the top member of each of the lists under sorted access, then they access the second member of each of the lists, and so on.) Wait until there are at least k "matches", that is, wait until there is a set H of at least k objects such that each of these objects has been seen in each of the m lists.

- For each object R that has been seen, do random access as needed to each of the lists L_i to find the ith field x_i of R.

- Compute the grade $t(R) = t(x_1, \cdots, x_m)$ for each object R that has been seen. Let Y be a set containing the k objects that have been seen with the highest grades. The output is then the graded set $\{(R, t(R)) | R \in Y\}$.

FA is correct for monotone aggregation functions t (that is, the algorithm can successfully find the top k results).

The Threshold Algorithm (TA) [43, 48, 73] works as follows:

- Do sorted access in parallel to each of the m sorted lists L_i. As an object R is seen under sorted access in some list, do random access to the other lists to find the grade x_i of object R in every list L_i. Then compute the grade $t(R) =$

$t(x_1, \cdots, x_m)$ of object R. If this grade is one of the k highest we have seen, then remember object R and its grade $t(R)$ (so that only k objects and their grades need to be remembered at any time).

- For each list L_i, let x_i be the grade of the last object seen under sorted access. Define the threshold value τ to be $t(x_1, \cdots, x_m)$. As soon as at least k objects have been seen whose grade is at least equal to τ, then halt.

- Let Y be a set containing the k objects that have been seen with the highest grades. The output is then the graded set $\{(R, t(R))|R \in Y\}$.

TA is correct for each monotone aggregation function t. The stopping rule for TA always occurs at least as early as the stopping rule for FA (that is, with no more sorted accesses than FA). In FA, if R is an object that has appeared under sorted access in every list, then by monotonicity, the grade of R is at least equal to the threshold value. Therefore, when there are at least k objects, each of which has appeared under sorted access in every list (the stopping rule for FA), there are at least k objects whose grade is at least equal to the threshold value (the stopping rule for TA).

In relational databases, there are lots of other work that extend the evaluation of SQL queries for top-k processing. Carey and Kossmann [21] optimized top-k queries when the scoring is done through a traditional SQL order by clause, by limiting the cardinality of intermediate results. Other works [17, 26] deployed statistical information to map top-k queries into selection predicates which may require restarting query evaluation when the number of answers is less than k. Over multiple repositories in a mediator setting, Fagin et al. proposed a family of algorithms [41, 44, 42], which can evaluate top-k queries that involve several independent subsystems, each producing scores that are combined using arbitrary monotonic aggregation functions. These algorithms are sequential in that they completely process one tuple before moving to the next tuple. The Upper [70], MPro [22] and TPUT [20] algorithms show that

interleaving probes on tuples results in substantial savings in execution time. In addition, Upper [70] used an adaptive per-tuple probe scheduling strategy, which results in additional savings in execution time when probing time dominates query execution time.

2.4.2 Adaptive Top-k Approach in XML

Marian et. al. in [69] explored an adaptive top-k query processing strategy in XML, which permits different query plans for different partial matches and maximizes the best scores. Based on the intermediate results, the irrelevant answers for the top-k query may be pruned as early as possible. In their work, they present a conservative extension of the $tf * idf$ function to XPath queries against XML documents. The first point to note is that, unlike traditional IR, an answer to an XPath query need not be an entire document, but can be any node in a document. The second point is that an XPath query consists of several predicates, instead of simply "keyword containment in the document" (as in IR). Thus, the XML analogs of idf and tf would need to take these two points into consideration.

Definition 3 ***XPath Component Predicates*** *Consider an XPath query Q, with q_0 denoting the query answer node, and q_i, $1 \leq i \leq l$, denoting the other query nodes. Let $p(q_0, q_i)$ denote the XPath axis between query nodes q_0 and q_i, ($i \geq 1$). Then, the component predicates of Q, denoted P_Q, is the set of predicates $\{p(q_0, q_i)\}$, $1 \leq i \leq l$.*

For example, the component predicates of the XPath query /a[./b and ./c[.//d and following-sibling::e]] is the set {a[parent::doc-root], a[./b], a[./c], a[.//d], a[./e]}. The component predicates provide a unique decomposition of the query into a set of "atomic predicates".

Definition 4 ***XML*** *idf Given an XPath query component predicate $p(q_0, q_i)$, and*

an XML database D, p's idf against D, $idf(p(q_0, q_i), D)$, is given by:

$$\log(\frac{|\{n \in D : tag(n) = q_0\}|}{|\{n \in D : tag(n) = q_0 \wedge (\exists n' \in D : tag(n') = q_i \wedge p(n, n'))\}|})$$

Intuitively, the *idf* of an XPath component predicate quantifies the extent to which q_0 nodes in the database D additionally satisfy $p(q_0, q_i)$. The fewer q_0 nodes that satisfy predicate $p(q_0, q_i)$, the larger is the *idf* of $p(q_0, q_i)$.

Definition 5 ***XML*** tf *Given an XPath query component predicate $p(q_0, q_i)$, and a node $n \in D$ with tag q_0, p's tf against node n, $tf(p(q_0, q_i), n)$, is given by:*

$$|\{n' \in D : tag(n') = q_i \wedge p(n, n')\}|$$

Intuitively, the tf of an XPath component predicate p against a candidate answer $n \in D$ quantifies the number of distinct ways in which n satisfies predicate p.

Definition 6 ***XML*** $tf * idf$ ***score*** *Given an XPath query Q, let P_Q denote Q's set of component predicates. Given an XML database D, Let N denote the set of nodes in D that are answers to Q. The the score of answer $n \in N$ is given by:*

$$\sum_{p_i \in P_Q} (idf(p_i, D) * tf(p_i, n))$$

In their work, the Whirlpool architecture is proposed to compute top k answers. It consists of the router, router queue, servers, server queues and the top-k set where the router will select the partial matches from the router queue and make the determination of the next server that needs to process the partial matches and send the partial matches to the queue of that server; the server to be reached will extend the partial matches with the nodes in the server queue if the nodes are consistent with the structure of the queries and then compute scores for each of the extended matches;

The newly computed partial matches will be sent to a Top-k set where a candidate set of top-k matches is maintained.

However, in their work, the query component predicates are assumed to be independent. The top-k results can not be determined until all candidates are evaluated. In addition, they do not consider how to efficiently compute top-k results over a larger number of different data sources.

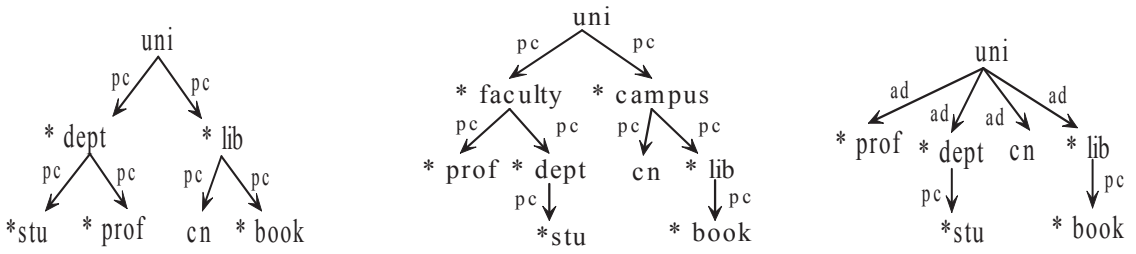
Chapter 3

Similarity Computation between Schemas

In this chapter, we study the problem of measuring structural similarities of large number of source schemas against a single domain schema, which is useful for enhancing the quality of searching and ranking big volume of source documents on the Web with the help of structural information. After analyzing the improperness of adopting existing edit-distance based methods, we propose a new similarity measure model that caters for the requirements of the problem. Given the asymmetric nature of the similarity comparisons of source schemas with a domain schema, similarity preserving rules and algorithm are designed to filter out uninteresting elements in source schemas for the purpose of optimizing the similarity computation. Based on the model, a basic algorithm and an improved algorithm are developed for structural similarity computation. The improved algorithm makes full use of a new coding scheme that is devised to reduce the number of comparisons. Complexities of both algorithms are analyzed and extensive experiments are conducted showing the significant performance gain achieved by the improved algorithm.

3.1 Introduction

Since XML has become the standard for representing, exchanging and integrating data on the Web, more and more information or application data stored or exchanged on the Web is adhering to this format. Searching the Web for finding interesting services or information now becomes part of people's lives. The flexible structures of XML documents make this kind of search quite complex, and sometimes may impose relaxed conditions and return approximate results. Due to different perceptions of an application domain, different providers of a service may define different schemas which we call source schemas for their source data. Meanwhile, clients who require the service may issue queries based on the common understanding of the domain which we call a domain schema. For example, Figure 3.1(a) shows a domain schema T_0 for universities. The schema defines university, department, student, professor, library, campus name, and book as its interesting elements and their relationships. Figure 3.1(b) shows a source schema T for a particular university. Apart from all interesting elements in Figure 3.1(a), this schema has elements for faculty and campus which may not be interesting. There are also some structural differences between T_0 and T, such as the relationship between departments and professors. Therefore, to be able to get results or approximate results directly from a source XML document, it is very important to work out the similarity of the source schema with regards to the domain schema. Recently, research in information retrieval on XML documents over the Web attracts a lot of attention. Instead of using a pure CO (content only) query, we may now use a so called CAS (content-and-structure) query [47] to express the topic statement more precisely by adding explicit references to XML structure, by restricting either the context of interest or the context of certain search concepts. In this kind of search, the structural similarity of the source schemas compared with the domain schema against which a CAS query is issued is again a key point for ranking



(a) Domain Schema Tree T_0 (b) Source Schema Tree T (c) Trimmed Tree of T

Figure 3.1: Extracting Interesting Structure from Source Schema based on Domain Schema

the set of source XML documents. The problem is how to efficiently compute the structural similarities of the potentially huge number of candidate source schemas against a domain schema.

In recent years, a great deal of attention has been put on computing the structural similarity between XML documents [29, 45, 12]. Work has also been presented on matching XML schemas for schema mapping and integration [68, 38]. To the best of our knowledge, none of them is proposed to tackle the problem that we propose in this chapter. The structural similarity problem of our interest is to compare source schemas with a domain schema for the purpose of searching and ranking those source documents that conform to their corresponding source schemas. Queries are issued against the domain schema and the returned source documents are ranked based on the similarities between their corresponding source schemas and the domain schema. Therefore, the problem is somehow asymmetric for the schemas to be compared. It takes as input two schemas, T and T_0 representing a candidate source XML schema and the domain XML schema, respectively. We are able to take T_0 as a base to trim T first, and then compare their structures. For example, the source schema T in Figure 3.1(b) can be trimmed based on the domain schema T_0 in Figure 3.1(a), yielding a trimmed schema as shown in Figure 3.1(c). In this chapter, we present a

framework and algorithms to measure the structural similarity of T with regards to T_0.

The contributions of this chapter are as follows:

- We propose a new similarity measure model to compute the structural similarity degree of any candidate source XML schema against a given domain XML schema after analyzing the requirements of the problem and the improperness of adopting existing edit-distance based methods to tackle the problem (Section 3.2).
- Given the asymmetric nature of the similarity comparisons of schemas to be compared, we design several similarity preserving rules and an algorithm to filter out uninteresting elements from source schemas for the purpose of optimizing the similarity computation (Section 3.3).
- We develop a basic algorithm and an improved algorithm to compute the structural similarity between a source schema and a domain schema with complexity analysis. An efficient coding scheme is devised to speed up the similarity computation and is fully used in the improved algorithm (Section 3.4). Experimental results show the significant performance gain of the improved algorithm (Section 3.5).

The conclusions of this chapter are provided in Section 3.6.

3.2 Measuring Structural Similarity

In this section, we present a framework for measuring the structural similarity of a candidate source schema against a domain schema. First we use a motivating example to illustrate what are required for the similarity problem we are targeting and why an existing edit-distance method is not suitable to the problem. Following

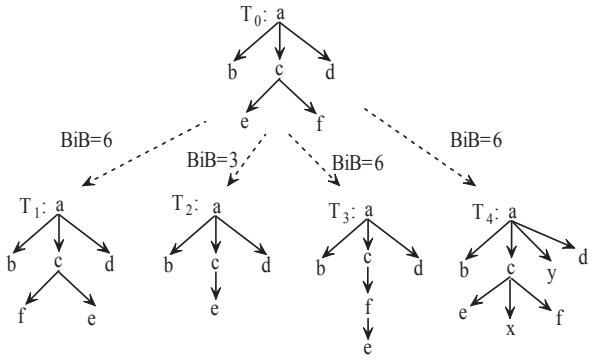


Figure 3.2: A Motivating Example

the discussions, we propose a new similarity measure model. Finally we justify that the proposed model is effective for solving the similarity problem by reviewing the motivating example and comparing with the edit-distance method.

3.2.1 A Motivating Example

To motivate our work, let us look at an example shown in Figure 3.2. In the example, a domain schema tree T_0 and four source schema trees T_1, T_2, T_3 and T_4 are given. If we use an edit-distance based method such as *BiBranch* proposed in [91] to compute the structural similarity, the similarity values of T_1, T_2, T_3 and T_4 against T_0 are 6, 3, 6, and 6, respectively. They are shown in Figure 3.2 denoted as *BiB*. In *BiBranch*, the smaller the *BiB* value is, the more similar its corresponding pair of trees are.

There are several problems in the results calculated using *BiBranch*.

- $BiB(T_0, T_1) = 6$ is not correct. Instead, we would expect $BiB(T_0, T_1) = 0$. In other words, when we query a source document conforming to T_1 based on T_0, we do not care about the order of the sibling elements e and f and the results retrieved from those source documents conforming to T_1 should be ranked among the highest.

- $BiB(T_0, T_2) = 3$ and $BiB(T_0, T_3) = 6$ are not expected. Instead, we would expect $BiB(T_0,T_2) > BiB\ (T_0,T_3)$. In other words, T_3 is more similar to T_0 than T_2. When searching documents, it is most important that all interesting elements are not missed out. Missing an element will impact more on the similarity than placing the element in an inconsistent position. In T_2 element f is missing while in T_3 only the relationship between e and f is inconsistent with that in T_0. The relationship between c and e is slightly different where a parent-child (pc) relationship ("/") holds in T_0 while an ancestor-descendant (ad) relationship ("//") holds in T_3. When a query containing element f is issued against T_0, no result will be returned from those source documents conforming to T_2. However, results will be returned from those source documents conforming to T_3 as long as the relationship between c and e is not strictly a pc relationship in the query.

- $BiB(T_0, T_4) = 6$ is not correct either. Instead, we would also expect $BiB(T_0, T_4) = 0$. In other words, when we query a source document conforming to T_4 based on T_0, we do not care about whether the source document contains some "noise" elements that do not impact on the query results. Again, the results retrieved from those source documents conforming to T_4 should be ranked among the highest.

From the above analysis, it is obvious that edit-distance based methods are not suitable for solving our schema similarity problem and a new similarity measure model is needed. To compute the similarity of a source schema against a domain schema, the new model is required to first prioritize the coverage in the source schema of every interesting element appearing in the domain schema and then to consider the consistency of the relationships between any pair of interesting elements in both schemas. From the example, we have pc, ad, and sibling relationships. We also know

that the order of sibling elements do not matter and that "noise" elements in a source schema can be ignored if they do not affect the relationships of any pair of elements in the schema.

3.2.2 Structural Similarity Model SSD

In this section, we present a new similarity model for measuring **S**imilarity of **S**ource schemas against a **D**omain schema (*SSD*). The *SSD* model provides accurate similarity measures by taking into account two main factors that contribute to the structural similarity or difference: element coverage and consistency of relationships of element pairs. In addition, we also consider the difference of element cardinality in the model. We choose the similarity degree value in $[0, 1]$. Unlike *BiBranch*, the bigger the *SSD* value is, the more similar a source schema is with regards to the domain schema.

Both the source schemas and the domain schema are represented as schema trees. We first define a schema tree.

Definition 7 ***XML Schema Tree:*** *An XML schema tree is defined as T = (V, E, v_r, $Card$) where*

- *V is a finite set of nodes, representing elements and attributes of the schema.*
- *E is a set of directed edges. Each edge $e(v_1, v_2)$ represents the parent-child relationship between two nodes v_1, $v_2 \in V$, denoted by $P(v_2)$=v_1 or $v_2 \in Ch(v_1)$ where $P : V \rightarrow V$, for any $v \in V$ and $v \neq v_r$, $P(v)$ is the parent node of v; $Ch : V \rightarrow 2^V$, for any $v \in V$, $Ch(v)$ is a set of child nodes of v.*
- *$v_r \in V$ is the root node of tree T.*
- *$Card : V \rightarrow \{$"1", "*"$\}$ where $Card(v)$ = "1" represents that there is only one occurrence of v under $P(v)$ in a document conforming to T; $Card(v)$ = "*" represents that there are more than one occurrences of v under $P(v)$.*

Given a domain schema tree $T_0 = (V_0, E_0, v_{r0}, Card)$ and a source schema tree $T = (V, E, v_r, Card)$, we need to compute $SSD(T_0, T)$. The first contributing factor is the element coverage. This can be calculated by the ratio of interesting objects (RIO) showing the proportion of interesting elements of T_0 in T, which is calculated by the following formula.

Ratio of Interesting Object $(RIO(V_0, V))$

$$RIO = \frac{|V'|}{|V_0|} \tag{3.1}$$

where $V' = V \cap V_0$ is the set of interesting nodes in V.

Compared with the domain schema tree T_0 in Figure 3.2, the source schema tree T_4 contains all the interesting elements while T_2 includes 5 interesting nodes. So we have $RIO(V_0, V_4) = 6/6 = 1$ and $RIO(V_0, V_2) = 5/6 = 0.833$, respectively.

The second contributing factor is the consistency degree of all node pairs in T compared with the corresponding node pairs in T_0. Before we discuss the similarity of node pairs, we first discuss the cardinality that may affect the similarity of node pairs. In a schema tree, the cardinality of each element is recorded. Based on this cardinality, we can derive the relative cardinality of a pair of nodes in the schema tree.

Definition 8 ***Relative Cardinality:*** *Given a schema tree $T = (V, E, v_r, Card)$ and any two nodes v_1, $v_2 \in V$ such that there exists a path from v_1 to v_2, we define the relative cardinality between v_1 and v_2 as RCard(v_1, v_2) where RCard(v_1, v_2) is set to "1" if every node v on the path from v_1 to v_2 satisfies $Card(v)$ = "1"; otherwise, RCard(v_1, v_2) is set to "*".*

Given a pair of nodes (v_1, v_2) in V' and its counterpart (v_{01}, v_{02}) in V_0, we can first define the cardinality similarity of node pairs (CSNP) indicating the cardinality difference between node pairs that satisfy the ad or pc relationships.

Cardinality similarity of node pairs ($CSNP(v_1, v_2, v_{01}, v_{02})$)

$$CSNP = \begin{cases} \omega, & RCard(v_1, v_2) \neq RCard(v_{01}, v_{02}) \\ 1, & RCard(v_1, v_2) = RCard(v_{01}, v_{02}) \end{cases} \tag{3.2}$$

CSNP is set to 1 when the two node pairs have consistent relative cardinality; otherwise it is set to ω. Here, $\omega \in [0, 1]$ represents the degree that clients can tolerate with the cardinality difference. We permit clients to evaluate the effect of the cardinality on the similarity by adjusting the value of ω. Normally this value is relatively high. We assume its default value is 0.8.

For any pair of nodes, they must have one of the three relationships: a pc relationship, an ad relationship, or an extended sibling relationship defined as follows.

Definition 9 ***eSibling:*** *Given a pair of nodes* (v_1, v_2)*, if neither* $pc(v_1, v_2)$ *or* $ad(v_1, v_2)$*, nor* $pc(v_2, v_1)$ *or* $ad(v_2, v_1)$ *hold,* v_1 *and* v_2 *are said to satisfy an extended sibling relationship and this relationship is denoted as* $eSibling(v_1, v_2)$*.*

Now, we define the similarity of node pairs (SNP) that specifies the structural relationship between node pairs.

Similarity of Node Pairs ($SNP(v_1, v_2, v_{01}, v_{02})$)

$$SNP = \begin{cases} CSNP(v_1, v_2, v_{01}, v_{02}), & \text{case 1} \\ \lambda \times CSNP(v_1, v_2, v_{01}, v_{02}), & \text{case 2} \\ 1, & \text{case 3} \\ 0, & \text{case 4} \end{cases} \tag{3.3}$$

where

- Case 1: $((v_{01}/v_{02}) \wedge (v_1/v_2)) \vee ((v_{01}//v_{02}) \wedge (v_1//v_2))$ means if the node pairs satisfy the pc or ad relationships at the same time, we can directly compute the value of SNP according to the CSNP value of the node pairs.

- Case 2: $((v_{01}/v_{02}) \wedge (v_1//v_2)) \vee ((v_{01}//v_{02}) \wedge (v_1/v_2))$ means that one node pair satisfies the pc relationship and the other satisfies the ad relationship, in this case, clients may choose to adjust the parameter $\lambda \in [0, 1]$ which represents the degree that clients can tolerate the difference between "/" and "//". We assume its default value is also 0.8.

- Case 3: $eSibling(v_1, v_2) \wedge eSibling(v_{01}, v_{02})$ means the node pairs (v_1, v_2) and (v_{01}, v_{02}) are structurally consistent in that both of the node pairs have an extended sibling relationship and the value of SNP is set to 1.

- Case 4: if the above three cases do not hold, the value of SNP will be set to 0 representing that the node pairs are not matched, e.g., one pair satisfies pc/ad relationship while the other satisfies *eSibling* relationship.

Given a node pair (c, e) in T_0 and its counterpart in T_3 shown in Figure 3.2, it is easy to see (c, e) satisfies a pc relationship in T_0 while its counterpart satisfies an ad relationship in T_3. And they have the same relative cardinality due to $RCard(c, e) = 1$ in both sides. So we have $SNP = \lambda \times 1 = 0.8$ where we use the default value of λ. If one of them changed its relative cardinality, i.e. their relative cardinality were not same, the value of SNP would be $\lambda \times \omega = 0.8 \times 0.8 = 0.64$.

Now we provide the SSD similarity value between T_0 and T, which combines both contributing factors.

Similarity of source schema w.r.t. domain schema (SSD(T_0, T))

$$SSD = RIO(V_0, V) \times (\frac{1}{C^2_{|V'|}} \sum SNP(v_1, v_2, v_{01}, v_{02})) \tag{3.4}$$

In the above equation, the second contributing factor is calculated by taking into account similarities of all corresponding node pairs in V' and V_0. After we substitute Equation 3.1 and $C^2_{|V'|} = \frac{|V'| \times (|V'|-1)}{2}$ into Equation 3.4, we can get the final similarity

model:

$$SSD = \frac{2}{|V_0| \times (|V \cap V_0| - 1)} \sum SNP(v_1, v_2, v_{01}, v_{02}) \tag{3.5}$$

3.2.3 Effectiveness Analysis

Come back to the motivating example in Section 3.2.1, now we can compute the similarity values of T_1, T_2, T_3 and T_4 against T_0 using Equation 3.5. For example, given the source schema tree T_3 and the domain schema tree T_0, we have $|V_0| = |V_3 \cap V_0| = 6$, so $RIO = 1$ because all the elements in T_0 can be found in T_3. Then we check if the relationship between every two elements in T_0 is consistent with its counterpart in T_3. From Figure 3.2, differences can be found for two node pairs (c, e) and (f, e). For (c, e), a pc relationship holds in T_0 while an ad relationship holds in T_3, so it contributes to 0.8 (the default value of λ. For (f, e), an *eSibling* relationship holds in T_0 while a pc relationship holds in T_3, so it contributes to 0 because the inconsistency of the node pairs. It is easy to calculate $\sum SNP(v_1, v_2, v_{01}, v_{02}) = 13 + 0.8 + 0 = 13.8$ out of the total of 15 node pairs. Now, we can get the final result $SSD(T_0, T_3) = \frac{2}{6\times(6-1)} \times 13.8 = 0.92$. According to the same procedure, we have $SSD\ (T_0,\ T_1) = 1$, $SSD\ (T_0,\ T_2) = 0.556$ and $SSD\ (T_0,\ T_4) = 1$, respectively.

Based on the above SSD results, we can see that the SSD similarity values of T_1 and T_4 against T_0 are all 1. These two similarities are exactly what we expected. We can also see that the SSD similarity value of T_3 against T_0 (0.92) is much higher than that of T_3 against T_0 (0.556). This reflects that a missing element will affect more on the similarity than misplacing an element, which is what we expected. Compared with *BiBranch*, SSD is much more effective when computing the similarity of a source schema against its domain schema.

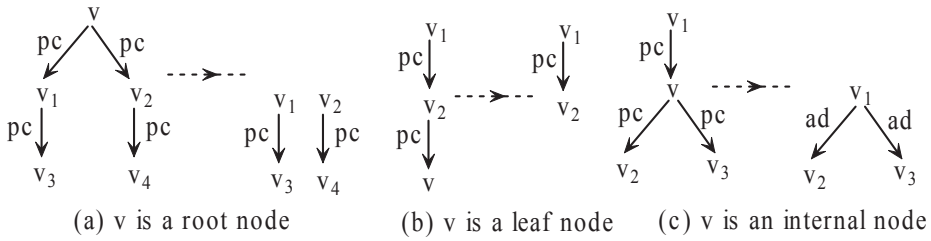


Figure 3.3: Trimming Rules

3.3 Similarity Preserving Trimming

In the SSD model, the "noise" elements of a source schema are not counted directly for the similarity computation because of the asymmetric nature of the similarity problem. To reduce the number of comparisons, it is desirable to filter out those "noise" elements while preserving the similarity of the source schema against the domain schema. In this section, we propose a set of trimming rules and an algorithm based on these rules to filter out all "noise" elements in a source schema based on the domain schema. We also prove that the order of applying rules is insignificant and the algorithm as well as each rule preserve the similarity property.

3.3.1 Basic Trimming Rules

Let $T(V, E, v_r, Card)$ and $T_0(V_0, E_0, v_{r0}, Card)$ be the XML schema trees for a source schema and a domain schema, respectively. As we discussed above, only those nodes in T_0 and their relationships are interesting to clients. As such, we are able to take T_0 as a base to trim T by deleting the uninteresting nodes $v \in \{V - V_0\}$. When a node is to be deleted, the edges linked to the node should be changed. According to the location of v in T, we have the following three updating rules in Figure 3.3:

- **Rule 1:** If $v = v_r$ (node v is the root node), then all the edges $\{(v, v_i)|v_i \in Ch(v)\}$ need to be deleted.

- **Rule 2:** If $Ch(v) = \phi$ (node v is a leaf node), then the edge $\{(P(v), v)\}$ needs to be deleted.

- **Rule 3:** If $P(v) \neq \phi \wedge Ch(v) \neq \phi$ (node v is an internal node), then: (1) for all $v_i \in Ch(v)$, the cardinality of v_i is updated as $max\{Card(v), Card(vi)\}$; (2) all edges relating to node v, i.e. $\{(v, v_i)|v_i \in Ch(v)\} \cup (P(v), v)$ are deleted from E; (3) new edges linking the parent node and all the child nodes, i.e. $\{(P(v), v_i)|v_i \in Ch(v)\}$ are inserted into E and re-labeled to ad-edges.

The trimmed result remains as a schema tree if the remaining nodes are all connected, or becomes a schema forest containing multiple trees (e.g. we may employ a virtual root to connect the multiple trees in the schema forest). For example, we take T_0 in Figure 3.1(a) as a base to trim T in Figure 3.1(b), the trimmed tree of T is shown in Figure 3.1(c).

Definition 10 ***Trimming Schema Forest:*** *Given a domain XML schema tree $T_0(V_0, E_0, v_{r0}, Card)$ and a candidate source XML Schema tree $T(V_t, E_t, v_{rt}, Card)$, the trimmed result is represented as a forest $F(V, E, V_r, Card)$, where*

- *$V = V_0 \cap V_t$.*

- *E is the set of edges after applying one of Rules 1-3 for each $v \in \{V_t - V_0\}$. $E = E_c \cup E_d$ consists of two kinds of edges called pc-edges (E_c) and ad-edges (E_d), corresponding to the child and descendant axes of XPath.*

- *V_r is the set of root nodes of the trees resulted from deleting v_{rt} and some top level nodes in $\{V_t - V_0\}$. $|V_r|$ represents the number of trees in F.*

- *$Card$ is defined the same as that in Definition 7.*

Theorem 1 *Let $T_0(V_0, E_0, v_{r0}, Card)$ and $F(V, E, V_r, Card)$ be a domain XML schema tree and a source schema forest (or tree). Each of the rule in Rule 1 to Rule 3 preserves similarity when applying to F based on T_0 in terms of the SSD model.*

Proof sketch: Suppose that the transformed source schema forest is $F'(V', E', V'_r, Card)$ after applying a rule in Rule1 to Rule 3, we prove that $SSD(T_0, F')$=$SSD(T_0, F)$ in two parts. The first part is RIO. Obviously F' preserves all node in $V \cap V_0$ because Rule 1 to Rule 3 only filters out a node in $v \in \{V - V_0\}$. So we have $RIO(T_0, F')$=$RIO(T_0, F)$. The second part is SNP. For any two nodes $v_1, v_2 \in V \cap V_0$, we show that the relationship between v_1 and v_2 remains unchanged. For Rule 1/Rule 2, when a root/leaf node v is removed, obviously there is no change to the relationship of any remaining node with the other nodes. For Rule 3, when an indermediate node v is removed, $ad(v_1, v_2)$ and $ad(v_1, v_3)$ remain unchanged because the new edges (v_1, v_2) and (v_1, v_3) are re-labeled as ad-edges. Furthermore, $RCard(v_1, v_2)$ and $RCard(v_1, v_3)$ are also adjusted accordingly. So F and F' agree on the SNP value for any node pair (v_1, v_2) where $v_1, v_2 \in V \cap V_0$.

3.3.2 Trimming Algorithm

Algorithm 1 gives the whole trimming process in a top-down manner. The queue *tempQueue* is used to hold elements waiting to be processed. At the beginning of each loop, we use the function *GetElement*() to get an element v from *tempQueue* and insert its child elements into *tempQueue*. And then we check if v is in V_0 of the domain schema. If it does then if it is also a sub-root element we insert v into V_r. If v is not in V_0, there are three trimming cases: (1) v is a sub-root element - all the edges coming from v will be deleted (Lines 15-17, 30-31). (2) v is a leaf element - all the edges arriving to v will be deleted (Lines 18-20, 30-31). (3) v is an internal element - all the edges connecting to v will be deleted while a new set of *ad*-edges

Algorithm 1 Trimming Tree Algorithm

input:A domain schema tree $T_0(V_0, E_0, v_r, Card)$ and a source schema tree $T(V_t, E_t, v_{rt}, Card)$

output:A forest $F(V, E, V_r, Card)$

1: $V = V_t$;
2: $E = E_t$;
3: push v_{rt} into a temporary queue *tempQueue*;
4: **while** *tempQueue* $\neq \phi$ **do**
5: $v =$ *tempQueue*.GetElement();
6: **if** $Ch(v) \neq \phi$ **then**
7: insert $\{v_i \in Ch(v)\}$ into *tempQueue*;
8: **end if**
9: **if** $v \in V_0$ **then**
10: **if** $P(v) = \phi$ **then**
11: insert v into V_r;
12: **end if**
13: **else**
14: **if** $P(v) = \phi$ **then**
15: E = E - $\{e(v, v_i)|v_i \in Ch(v)\}$;
16: **else if** $Ch(v) = \phi$ **then**
17: $E = E - \{e(P(v), v)\}$;
18: **else**
19: **for all** $v_i \in Ch(v)$ **do**
20: $Card(v_i) = max\{Card(v), Card(v_i)\}$;
21: $E = E - \{e(v, v_i)\}$;
22: $E_d = E_d + \{e(P(v), v_i)\}$;
23: $P(v_i) = P(v)$;
24: $Ch(P(v)) = Ch(P(v)) + v_i$;
25: **end for**
26: **end if**$E = E - \{e(P(v), v)\}$; $V = V - v$;
27: **end if**
28: **end while**

will be created and inserted into edge set E (Lines 21-29, 30-31).

Corollary 1 *Let $T_0(V_0, E_0, v_{r0}, Card)$ be a domain schema tree, $T(V_t, E_t, v_{rt}, Card)$ be a candidate source Schema tree. Let $F(V, E, V_r, Card)$ the trimmed result source schema forest after applying a series of rules in Rule 1 to Rule 3. Then the order of applying these rules in Rule 1 to Rule 3 is not significant.*

This can be inferred from Theorem 1 because each application of a rule in Rule 1 to Rule 3 preserves similarity.

Corollary 2 *Algorithm 1 correctly trims a source schema tree $T(V_t, E_t, v_{rt}, Card)$ based on the domain schema tree $T_0(V_0, E_0, v_{r0}, Card)$.*

First, Algorithm 1 filters out all "noise" nodes that are in V_t but not in V_0. Second, from Theorem 1 and Corollary 2, we know that the order of applying rules is not significant and the trimmed schema forest preserves the similarity of T against T_0.

3.4 Computing Similarity

Given a candidate source schema T and a domain schema T_0, we can take T_0 as a base to trim T into a schema forest F by applying Algorithm 1. In this section, we compute the similarity of F against T_0. To speedup the computation, we develop a new coding scheme and algorithm to code F and T_0 first. Based on the coding information, we develop two algorithms with complexity analysis.

3.4.1 Encoding XML Schema

Most of the previous coding schemes are used to quickly determine the structural relationship among any pair of tree nodes. To the best of our knowledge, it is Dietz's

Algorithm 2 Coding A Forest

input: A forest $F(V, E, V_r, Card)$
output: A vector Vec of encoded nodes$(TreeID, pre, post, C, P, RD)$

1: $TreeID = pre = post = 0$;
2: new stack S;
3: **for all** root node v in V_r **do**
4: $C = 0$;
5: $TreeID++$;
6: DFTraverse($TreeID$,v);
7: **end for**
8: Return the vector Vec;

numbering scheme that was firstly used to determine ad relationship between tree nodes [36] where each node is assigned with a pair of numbers (pre, post). His proposition is: given two nodes x and y of a tree T, x is an ancestor of y iff x occurs before y in the preorder traversal of T and after y in the postorder traversal, i.e. $x.pre < y.pre$ and $x.post > y.post$. Region coding scheme is another popular scheme adopted in many work [27, 55] with a pair of numbers $(start, end)$. Element x is the ancestor of element y iff $x.start < y.start$ and $y.end < x.end$. In order to reduce update cost, Li et. al. [63] proposed a variant in the form of $(order, size)$ that reserves additional code space for elements.

However, most of the previous coding schemes were used to improve the query efficiency of individual XML documents, rather than to serve for comparison of a pair of schemas. To this end, we propose a coding scheme that extends Dietz's numbering scheme for specifying more information for schema comparison.

Definition 11 ***Coding Scheme:*** *Any node v in a schema forest F can be represented with a tuple (pre, post, C, P, RD), where*

- *pre: represents the position of v when it is traversed in pre-order.*
- *post: represents the position of v when it is traversed in post-order.*

- *C: records the cardinality information of v. If Card(v) = "*", v's pre is recorded; otherwise, if $\exists v_a$ (v_a is v's nearest ancestor and $Card(v_a)$ = "*"), $v'_a s$ pre is recorded; otherwise, let $C = 0$.*
- *P: records the parent's pre of v.*
- *RD: records the rightmost descendant's pre of v.*

We propose a coding scheme for determining the relationships between elements and locating the range of elements that need to be compared. It has many appealing features: (1) Given a forest of n nodes, the *pre* of all nodes is in the continuous range of $[1, n]$, therefore, the *pre* of a node can be served as the index for the node; (2) Useful information for schema comparison is recorded, including one index for parents ($P - Index$) and another for the rightmost descendant ($RD - Index$); (3) Cardinality information is also preserved.

The coding information can be used to optimize similarity optimization. For example, ad-relationship can be easily determined from the codes of a node pair. Definition 8 can be simplified as follows.

Relative Cardinality: Given any two nodes v_i, $v_j \in V$, if there exists a path from v_i to v_j, then

$$RCard(v_i, v_j) = \begin{cases} \text{“1”}, & v_i.C = v_j.C \\ \text{“*”}, & v_i.pre < v_j.C \leq v_j.pre \end{cases} \tag{3.6}$$

where $v_i.C = v_j.C$ means that $Card(v)$ = “1” holds for any node v on the path from v_i to v_j; $v_i.pre < v_j.C \leq v_j.pre$ means that either $Card(v_j)$ = “*” holds or $Card(v)$ = “*” holds for any node v on the path from v_i to v_j.

Algorithm 2 first initializes several global variables, including *pre* and *post* that are used to assign *pre* and *post* codes to nodes. Then it traverses each tree in the schema

Algorithm 3 DFTraverse($TreeID$,v)

```
1: v.TreeID = TreeID;
2: pre = pre + 1;
3: v.pre = pre;
4: v.RD = pre;
5: if Card(v)="*" then
6:    v.C = v.pre;
7:    C = pre;
8: else
9:    v.C = C;
10: end if
11: if Ch(v) ≠ φ then
12:    Push(v,S);
13:    while Ch(v) ≠ φ do
14:       x=GetLeftNode(Ch(v));
15:       Ch(v)=Ch(v)-{x};
16:       DFTraverse(TreeID,x);
17:    end while
18:    if S ≠ φ then
19:       v=pop(S);
20:    end if
21: end if
22: if S ≠ φ then
23:    x =pop(S);
24:    v.P = x.pre;
25:    x.RD = v.RD;
26:    C = x.C;
27:    Push(x,S);
28: else
29:    v.P = null;
30: end if
31: post + +;
32: v.post = post;
33: Vec[v.pre]=v;
```

forest F in a depth-first manner and encodes each node according to Definition 11. For each tree, we get the root of the tree from V_r and assign it a number $TreeID$ identifying the tree. Then we call the recursive function $DFTraverse()$ with $TreeID$ and the root as inputs to encode the nodes in the tree into a vector. Finally, the vector will be returned with all the encoded nodes.

In Algorithm 3, a recursive function $DFTraverse()$, with a tree id $TreeID$ and the root node v as inputs, is used to traverse the tree (or sub-tree) T in a depth-first manner and to encode the nodes in T. Lines 2-4 increase the *pre* by 1 and then assign the value of *pre* as the *pre* code of v ($v.pre$) and as the RD code of v ($v.RD$) temporarily. Lines 5-11 are used to assign the C code of v ($v.C$) as either the *pre* code of itself ($v.pre$) if $Card(v) =$ "*" or the C value (0 as initial value) carry-forwarded from its parent. Notice, if $Card(v) =$ "*", C will record the *pre* code of v and be passed down the tree as the C code of its descendants possibly. Lines 12-22 check if v has child nodes. If it does, v is pushed to stack S, then recursively process its child nodes. After all its child nodes have been processed, pop up v. Line 23-29 are used to assign the P code of v and pass RD code to its parent node. In case v has no parent node, the P code of v is set to *null* (Lines 30-32). Lines 33-35 are used to increase the global variable *post* and then set the *post* code of v ($v.post$).

3.4.2 Basic Algorithm

Let V and V_0 be sets of nodes of F and T_0, respectively. A node-to-node map $m : V \rightarrow V_0$ is required for similarity computation. This map can be built by using an element-level matcher [64], structure-level matcher [11], or linguistic matcher. These techniques compare the attributes, combinations or names of elements, and other textual descriptions respectively for finding the correspondent elements. We can treat m as another attribute of each $v \in V$. By using this attribute, we can conduct pair wise comparisons of the correspondent nodes. The detailed procedure of this basic

algorithm (BA) is shown in Algorithm 4.

Algorithm 4 Basic Algorithm BA

input:node sets V and V_0 with coding information
output: Similarity $simi$

1: $SimiOfPair = 0$;
2: $n = |V|$;
3: $n_0 = |V_0|$;
4: **for** $(i = 1; i < n; i++)$ **do**
5: $v_1 = V[i]$;
6: $v_{01} = V_0[v_1.m]$;
7: **for** $(j = i + 1; j \leq n; j++)$ **do**
8: $v_2 = V[j]$;
9: $v_{02} = V_0[v_2.m]$;
10: $SimiOfPair += SNP(v_1, v_2, v_{01}, v_{02})$;
11: **end for**
12: **end for**
13: return $simi = \frac{2 \times SimiOfPair}{n_0 \times (n-1)}$;

3.4.3 Improved Algorithm

BA uses coding information for computing the similarity of node pairs but fails to use it for reducing the number of similarity comparison. Given any node pair (v_1, v_2) in F, it is important to determine whether we have to check the comparison with its counterpart pair (v_{01}, v_{02}) in T_0. From Equation 3.3, we may observe that if $pc(v_1, v_2)$ or $ad(v_1, v_2)$ holds then we have to check the similarity of node pairs (SNP) with (v_{01}, v_{02}) because it is possible $pc(v_{01}, v_{02})$ or $ad(v_{01}, v_{02})$ also holds and hence necessary comparison is required by using Equation 3.2; otherwise, we do not have to check the details of the node pairs. With this in mind, we design the improved algorithm (IA) that makes full use of the coding information to improve the performance of BA. The detailed procedure is shown in Algorithm 5.

The algorithm proceeds in the *pre* order of V (Line 4). For $v_1 \in V$, its counterpart

Algorithm 5 Improved Algorithm IA

input:node sets V and V_0 with coding information
output: Similarity $simi$

1: SimiOfPair = ProcessedV = 0;
2: $n = |V|$;
3: $n_0 = |V_0|$;
4: **for** $(i = 1; i < n; i++)$ **do**
5: $v_1 = V[i]$;
6: $v_{01} = V_0[v_1.m]$;
7: $V_0[v_{01}.pre].m = -1$;
8: **for** $(j = i + 1; j \leq v_1.RD; j++)$ **do**
9: $v_2 = V[j]$;
10: $v_{02} = V_0[v_2.m]$;
11: $v_{02}.m = i$;
12: $SimiOfPair += SNP(v_1, v_2, v_{01}, v_{02})$;
13: **end for**
14: $UntouchedV_0$;
15: **for** $(j = v_{01}.pre + 1; j \leq v_{01}.RD; j++)$ **do**
16: **if** $(V_0[j].m \neq -1) \wedge (v_0[j].m \neq i)$ **then**
17: $UntouchedV_0 += 1$;
18: **end if**
19: **end for**
20: $j = v_{01}.P$;
21: $ParentV_0 = 0$;
22: **while** $(j \neq 0)$ **do**
23: **if** $(V_0[j].m \neq -1) \wedge (V_0[j].m \neq i)$ **then**
24: $ParentV_0 += 1$;
25: **end if**
26: $j = V_0[j].P$;
27: **end while**
28: $ProcessedV += 1$;
29: $SimiOfPair += n - ParentV_0 - UntouchV_0 - ProcessedV - (v_1.RD - v_1.pre)$;
30: **end for**
31: return $simi = \frac{2 \times SimiOfPair}{n_0 \times (n-1)}$;

$v_{01} \in V_0$ is selected and is marked as -1 for processed (Lines 5-7). We use the attribute m of nodes in V_0 for marking purpose. With the $RD - Index$ of v_1, we only need to check pairs (v_1, v_2) that satisfies $ad(v_1, v_2)$ and compute SNP with their counterpart pairs (v_{01}, v_{02}) (Lines 8-13). We use $v_1.pre$ which equals to i to temporally mark v_{02} so that we do not need to reset this mark after use (Line 11). We can easily get the count ($UntouchedV_0$) of those pairs (v_{01}, v_{02}) in T_0 that satisfy ad (v_{01}, v_{02}) yet inconsistent with their counterparts in F (Lines 14-19). Similarly, we can get the count ($ParentV_0$) of those pairs (v_{01}, v_{02}) in T_0 that satisfy $ad(v_{02}, v_{01})$ yet inconsistent with their counterparts in F (Lines 19-24) using $P - Index$ of v_{01}. The matching number of pairs in T_0 with all those pairs (v_1, v_2) in F where $ad(v_1, v_2)$ does not hold can be calculated in Line 25. The similarity of F w.r.t. T_0 can be obtained by Equation 3.5.

3.4.4 Complexity Analysis

The space complexity in both algorithms is equal to the sum of the sizes of node sets V and V_0, i.e. $O(n_1 + n_0)$. For BA, the time complexity is $O(n^2)$ because the algorithm is conducted by using pair wise comparison of the nodes in V. Let N_0 and N be the maximum out-degree in T_0 and F, respectively. Before we analyse the time complexity for IA, we need to use a property of tree: for an N-ary tree T with height H, the maximum number of nodes in T is $n = \frac{N^{H+1}-1}{N-1}$. IA mainly consists of three parts:

- Lines 3, 7-12: We know all nodes in V need to be processed. For any node v and its level i, the number of descendants of v is $\frac{N^{i+1}-1}{N-1} - 1$. And the number of nodes that are at the same level as v is N^{H-i}. If the maximum height of the tree is H, then the number of comparisons is $\sum_{i=0}^{H}(\frac{N^{i+1}-1}{N-1} - 1) \times N^{H-i}$. As $H = log_N^{(N-1)n-1}$, the time complexity is $O(N \times n \times log_N^{(N-1)n+1})$;

- Lines 3, 14-17: Similar to the above, the time complexity is $O(N_0 \times n_0 \times log_{N_0}^{(N_0-1)n_0+1})$;

- Lines 3, 19-23: each time Lines 19-23 is executed, the number of ancestors is less than H_0. So the time complexity is $O(n \times log_{N_0}^{(N_0-1)n_0+1})$.

So if the schema trees are relatively balanced, the overall time complexity is $O(N \times n \times log_N^{(N-1)n+1} + N_0 \times n_0 \times log_{N_0}^{(N_0-1)n_0+1} + n \times log_{N_0}^{(N_0-1)n_0+1})$. However, when the schema trees are unbalanced, the performance of the algorithm will be degraded. The worst situation (i.e. $N = 1$) is the same as that of BA.

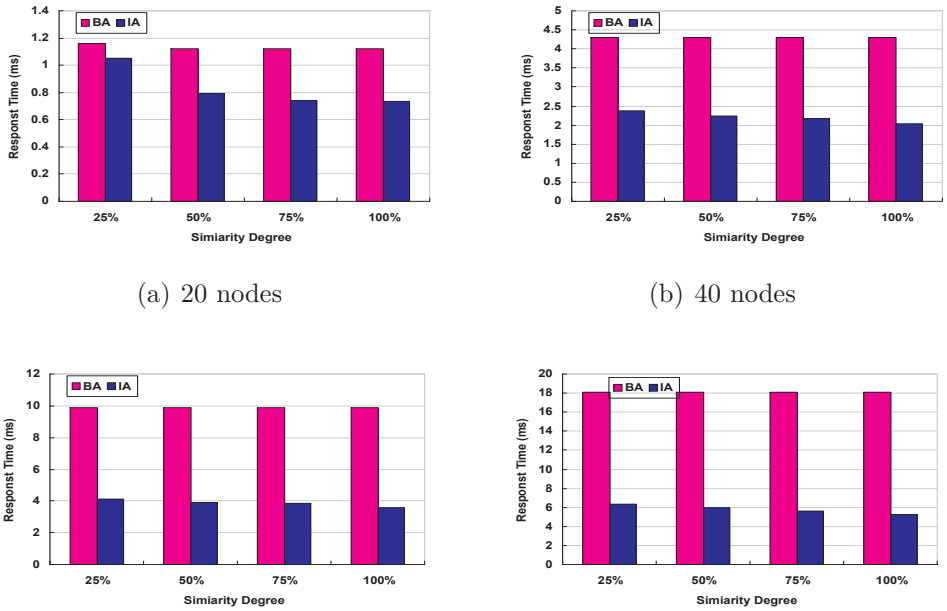
3.5 Experimental Results

Experiments are carried out on a Pentium IV 3.00GHz PC with 512MB main memory. The algorithms are implemented in C++. We use both synthetic datasets and a number of publicly available schemas [87] to compare the performance of our algorithms.

3.5.1 Sensitivity Test

In the first set of the experiments, we carry out a series of sensitivity analysis for different features of the data.

Sensitivity vs. Similarity Degree: Figure 3.4(a-d) compares the performance between BA and IA for various similarity degrees when the schema size is 20, 40, 60, and 80, respectively. The domain schemas are created based on *genexml.xsd* with adaptation of size while the candidate source schemas are manually created with differences from adapted domain schemas. The results show that BA is nearly not affected by the changes of similarity. For IA, however, the more similar the two



(a) 20 nodes

(b) 40 nodes

(c) 60 nodes

(d) 80 nodes

Figure 3.4: Response Time vs. Similarity Degree

schemas are, the faster. Moreover, the performance of IA is getting better with the size increases.

Sensitivity vs. Nested Level: We also use a set of synthetic datasets to evaluate the effect of nested level where every dataset includes 128 nodes and the level varies from 4 to 16. The results in Figure 3.5 illustrate that IA is much better than BA. The response time of IA is around 20% of that of BA. The performance of IA can be affected with the number of nested levels increases but still much better than that of BA. In real applications, it is seldom that the number of nested levels exceeds 20.

Sensitivity vs. Fanout: We finally use synthetic data to compare the performance of the two algorithms when the fanout varies. We generate four synthetic datasets that are of the same size of 128 nodes. The similarity degree is set to 100%

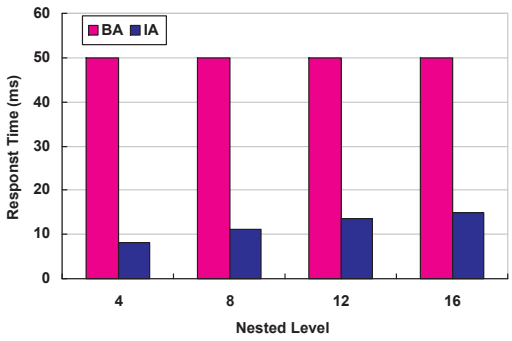


Figure 3.5: Response Time vs. Nested Level

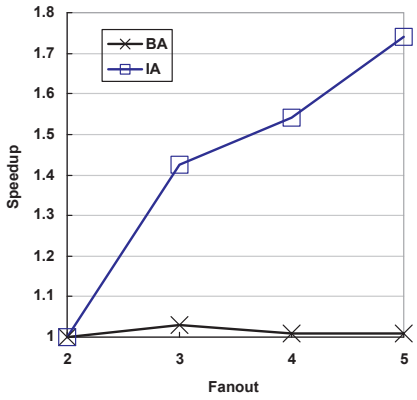


Figure 3.6: Response Time vs. Fanout

because we only consider the impact of fanout. Figure 3.6 shows the speedup of the two algorithms when fanout changes from 2 to 5. The experiment results illustrate the performance of IA is much better than that of BA when the fanout is equal to or greater than 3. There are two factors: the nested level will become smaller when the fanout is greater because we assume the size of schema is stable. On the other hand, the smaller nested structure is good to search its ancestors for every node when IA is carried out. But BA has to traverse every node that need to be compared, which cannot skip over any node.

3.5.2 Efficiency Test

We choose genexml.xsd as a base to evaluate our algorithms. The size varies from 20, 40, 60 and 80. Figure 3.7 shows IA is better than BA when the size is greater than 20. With the size increases, the processing time of BA increases greatly. However, the increasing trend is slow for IA. The experimental results in Figure 3.7 also show that IA needs less response time if the similarity degree is higher, e.g., when the schema size is less than 60, the response time is almost the same for schemas with similarity

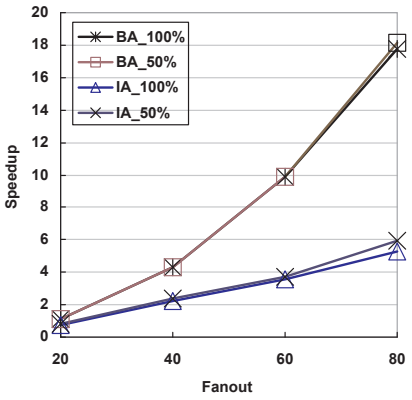


Figure 3.7: Response Time vs. the Size of Schema

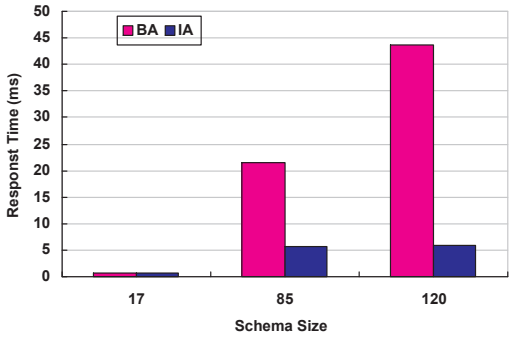


Figure 3.8: Response Time vs. the Size of Schema

degree 100% and 50%, respectively. When the size adds up to 80, we can see slight difference. It also shows that the change in similarity degrees has little impact on the performance of BA.

Figure 3.8 provides the experimental results for the three public datasets: *TPC-H-nested.xsd* (17), *genexml.xsd* (85), and *mondial-3.0.xsd* (120). In this set of experiments, the domain and source schemas are the same. The results illustrate that IA performs much better than BA, especially when the schemas contain more nodes. In real situation, potentially huge number of source schemas need to be compared with a domain schema. The performance gain of IA over BA will make a big difference.

3.6 Summary

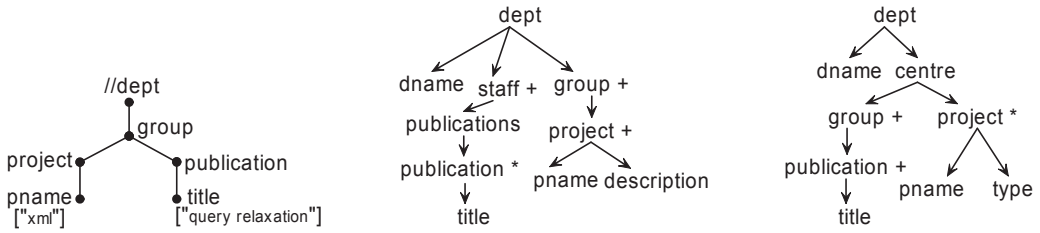
Compared with previous work in Section 2.1, we looked into the problem of efficiently computing structural similarity of potentially huge number of source schemas against a domain schema. This study is motivated from enhancing the quality of searching and ranking of huge number of XML source documents on the Web with the help of structural information, especially a domain schema against which queries are issued

and big number of source schemas which source documents may conform to. In this chapter, we proposed a new similarity model for measuring the structural similarity of a candidate source XML schema against a given domain schema and justified its effectiveness by comparing it with edit-distance based methods. To speed up similarity computation, we introduced a trimming process for filtering out uninteresting objects while preserving similarity. We also devised an efficient coding scheme. Two algorithms - the basic and the improved algorithms were developed with unnecessary comparisons removed in the improved algorithm. The experimental results showed that the improved algorithm outperforms significantly compared with the basic algorithm.

Chapter 4

Adaptive Relaxation of Structured Query

Searching XML data with a structured XML query can improve the precision of results compared with a keyword search. However, the structural heterogeneity of the large number of XML data sources makes it difficult to answer the structured query exactly. As such, query relaxation is necessary. Previous work on XML query relaxation poses the problem of unnecessary computation of a big number of unqualified relaxed queries. To address this issue, we propose an adaptive relaxation approach which relaxes a query against different data sources differently based on their conformed schemas. In this chapter, we present a set of techniques that supports this approach, which includes schema-aware relaxation rules for relaxing a query adaptively, a weighted model for ranking relaxed queries, and algorithms for adaptive relaxation of a query and top-k query processing. We discuss results from a comprehensive set of experiments that show the effectiveness and the efficiency of our approach.



(a) A query q (b) Schema d_1 for s_1 (c) Schema d_2 for s_2

Figure 4.1: A query and schemas of XML data sources s_1 and s_2

4.1 Introduction

As XML becomes the standard for representing web data, there is an increasing need to search and query XML data. Compared with a keyword search, a structured XML query allows a user to formulate the search requests more precisely. However, the structural heterogeneity of the potentially large number of XML data sources makes it difficult to answer a structured query exactly. The loosely-coupled nature of the data sources also makes it inapplicable for deploying the traditional federated database approach for integrating the XML data sources by defining a global schema. It would be ideal that a query could be smartly relaxed then be answered according to the data sources against which the query is issued.

Amer-Yahia et al. [8, 10] proposed a framework FleXPath for relaxing XML tree pattern queries (TPQs). Given a TPQ q, the closure of the structural and value-based predicates in q is first inferred and then is used to generate relaxed queries. The set of generated relaxed queries, including the query that includes the root of q, is complete. However, the relaxation process is basically blind and wild and the number of relaxed queries could be big. For a large number of heterogeneous XML data sources, many of the generated relaxed queries could be unqualified and will result in unnecessary cost of either computing or testing them.

As an example, we may issue a query against XML data sources maintained in

all Australian universities for searching those departments that have a group running project with a name containing "xml" and having publications with a title containing "query relaxation". As the number of universities is large and their data source structures may vary, the query is usually formulated according to users' understanding of a university. Figure 4.1(a) shows the query represented as a TPQ and it reflects the user's structural and value-based search requirements. Solely based on the query itself, FleXPath may need to consider 2^5 options, each could be a relaxed query that may be executed or tested against the university data sources. Some generated relaxed queries may be either too blind for some data sources thus return zero answers or too wild thus return answers that are far from what a user is expected. For example, the partial structures of the data source s_1 and s_2 for two universities are shown in schema d_1 in Figure 4.1(b) and schema d_2 in Figure 4.1(c), respectively. Obviously, the query itself will not return any result, and many relaxed queries will be generated by FleXPath from 2^5 options and then be evaluated or tested for both data sources. For example, among the relaxed queries, some of the useless relaxed queries for s_2 are listed in Figure 4.2. Actually, q_2 and q_3 in Figure 4.2 are useless for s_1 either.

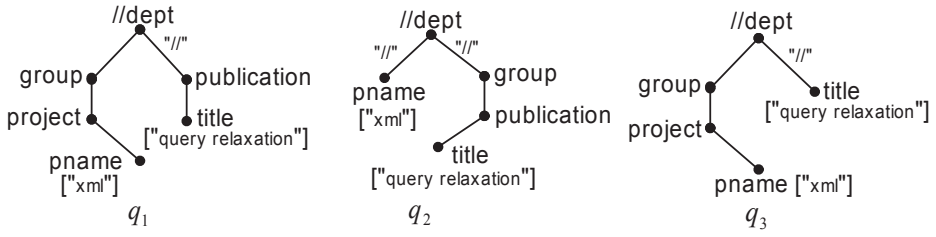


Figure 4.2: Relaxed queries of FleXPath

To deal with this problem, we propose an *adaptive query relaxation* (*AQR*) approach, which relaxes a query adaptively to each XML data source according to its conformed schema. Hence each relaxed query will be guaranteed to agree with the structural constraints imposed by the conformed schema of the data source, and as a

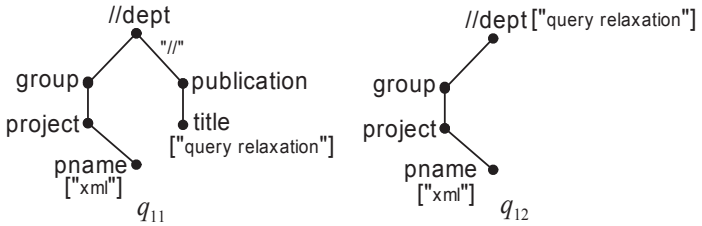


Figure 4.3: Relaxed queries of AQR for s_1

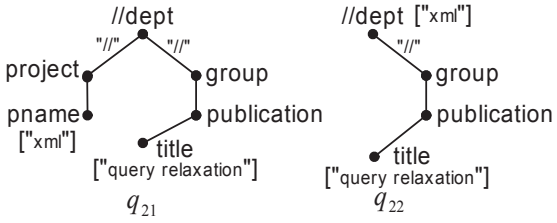


Figure 4.4: Relaxed queries of AQR for s_2

result, has high probability of generating an answer. For example, for schema d_1 in Figure 4.1(b) and schema d_2 in Figure 4.1(c), the relaxed queries generated by AQR are shown in Figure 4.3 and Figure 4.4, respectively.

AQR avoids blind relaxation. Each generated relaxed query for an XML data source is specific to the data source. In other words, a relaxed query that does not satisfy the structural constraints imposed by the conformed schema will not be generated. For example, for data source s_2, query q_1 in Figure 4.2 is useless and will not return any result because the edge between *group* and *project* in q_1 does not match d_2.

AQR also avoids wild relaxation. No unnecessary relaxation is needed because of the requirement that a data source has to conform with its schema. For example, the *-node *project* in d_2 implies the co-existence of *project* and *pname*. As such, for s_2, query q_2 in Figure 4.2 is too wild compared with query q_{21} in Figure 4.4. In

other words, after q_{21} is generated and evaluated, the time spent on generating and computing q_2 is unnecessary because no new result will be returned.

We need to explain that it is not groundless to leverage a schema to facilitate query relaxation, since most of commercial databases, such as IBM DB2 and Oracle, store and manage XML documents grouped by their XML schema or DTD. Without loss of generality, here we take DTD as the schema of XML data. In case the schema is not available for a data source, its structural information can be generated dynamically with data summarization tools [13, 14].

As a large number of data sources may be evaluated and the relaxed queries generated for these data sources may be different, it is desirable to first execute the relaxed query that is closest to the original query against the most promising data sources so that the most relevant answers will be returned first. This is especially important to evaluate a top-k query [69, 37]. For example, query q_{11} in Figure 4.3 is the closest query to the original query in all the generated relaxed queries in Figure 4.3 and Figure 4.4. One appealing feature of AQR is that the closest relaxed query is always associated with the data source that contains most relevant results because the query is generated based on the conformed DTD of the data source. In other words, a top-k query can be evaluated incrementally on the most relevant data source first and instead of rank returned results, we can rank the relaxed queries. To compare how much a relaxed query is close to the original query, we propose a penalty based ranking model to measure the difference between a relaxed query and the original query in AQR. For example, if the penalty for relaxing "/" to "//" is 0.1, we can compute the penalties of q_{11} and q_{12} in Figure 4.3 as 2 and 5, and the penalties of q_{21} and q_{22} in Figure 4.4 as 2.3 and 5.3. So q_{11} is the least penalized query or the closest query to the original query q. The details for computing the penalties can be found in Section 4.4.

To improve the accuracy and relevancy of the results, we allow a user to specify

weights on edges of a query and thus incorporate a weight into the ranking model. If the relationship between two nodes is less important than others, a smaller weight may be specified compared with the maximum weight 1. Our ranking model is based on the weight set on the original query and the penalty derived for a relaxed query. The ranking score of a relaxed query is calculated as the difference between the query weight for the original query and the penalty of the relaxed query. For example, when the weights for all edges are set to 1, the weight of the original query is 11. We know that the penalty for q_{11}, q_{12}, q_{21} and q_{22} are 2, 5, 2.3 and 5.3, so the scores for them are 9, 6, 8.7 and 5.7, respectively, i.e., the ranking list is [q_{11}, q_{21}, q_{12}, q_{22}]. If a user changes the weight of the edge between group and project to 0.5 while keep other edges as 1, the ranking list will be changed to [q_{21}, q_{11}, q_{22}, q_{12}]. The details for computing these ranking scores can also be found in Section 4.4.

In case the schema is not available for a data source, its structural information can be generated dynamically with data summarization tools [13, 14]. In this chapter, without loss of generality, we take DTD as the schema of XML data.

In summary, we claim the following contributions in this chapter:

- We propose and formalize the adaptive XML query relaxation problem w.r.t. different DTDs and devise a set of schema-aware relaxation rules.
- We develop a weight modification and penalty evaluation model to assess to what extent the original query is relaxed.
- We design a set of algorithms to describe how the rules and penalty model are leveraged in the process of relaxing queries.
- We run extensive experiments on XMark Benchmark to justify the efficiency and validity of our adaptive relaxation approach.

The rest of the Chapter is organized as follows. We introduce some relevant

definitions and give an overview of our AQR in Section 4.2. Section 4.3 discusses the relaxation rules in detail. Section 4.4 provides our weight modification and penalty evaluation models. The detailed descriptions of our adaptive relaxation algorithm are provided in Section 4.5. We present the results of extensive experiments and the conclusions in Section 4.6 and Section 4.7, respectively.

4.2 Overview

The goal of query relaxation is to relax the query constraints such that approximate answers can be returned if the original query returns no answer or not enough answers. This is especially useful when we query a big number of heterogeneous XML data sources using a single structured XML query. Given a TPQ q, FleXPath generates relaxed queries by enumerating all possible combinations starting from q itself to the root of q, thus resulting in large number of relaxed queries. Among these queries, some of them do not even match the structure of any XML data source so return no result; some of them may return result from some data sources, but return no results from others. Keep this in mind, our AQR approach does not enumerate the possible combinations for generating relaxed queries. Instead, we use structural information of data sources such as DTDs for customizing the generation of relaxed queries for different data sources. To achieve this, we design a set of adaptive relaxation rules to guide the generation of relaxed queries for different data sources based on their conformed DTDs. To evaluate a top-k query incrementally, we choose the closest or least penalized relaxed query first for execution. As such, we devise a comprehensive ranking model based on penalties. To improve the accuracy of the returned answers, we also allow users to specify parameters such as edge weights and coefficient. We see this as an important alternative to users because users are able to express customised requirements on their queries in terms of their preferences. We incorporate this

support into the ranking model and also extend the definition of a tree pattern query.

Definition 12 ***Weighted Tree Pattern Query*** *(WTPQ): A weighted tree pattern query q is defined as a tree $T(V, E, r, w)$, where V is a finite set of nodes. Each $v \in V$ is uniquely identified and may have search requirement of a term t denoted as $\tau(t)$. tagname(v) specifies the tag name of v. E consists of two kinds of edges called pc-edges and ad-edges, corresponding to the child and descendant axes of XPath. r is the root node that is a distinguished node in V corresponding to the answer node. For any $e(v_1, v_2) \in E$, $w(v_1, v_2) \in (0, 1]$ marks a weight for e with 1 as the default value. For any $v \in V$, $ch(v)$ gives the set of child nodes of v, $p(v)$ specifies the parent node of v. pc, ad specify parent-child and ancestor-descendant relationships between a pair of nodes (v_1, v_2), respectively. pc(v_1, v_2) if $e(v_1, v_2) \in E$ and e is an pc-edge. ad(v_1, v_2) if v_1 is an ancestor node of v_2, or $e(v_1, v_2) \in E$ and e is an ad-edge.*

In AQR, a DTD is defined as a directed acyclic graph (DAG) with a single root (for the document element). We will extend the definition to allow recursive DTD definition when we discuss the recursive relationship relaxation.

Definition 13 ***DTD Graph****: A DTD d is defined as a directed acyclic graph $G(V_d, A_d, r_d)$, where V_d is a finite set of nodes. Each node $v \in V_d$ specifies an element or an attribute in XML documents that conform to d and is uniquely identified. tagname(v) specifies the tag name of v. $ch(v)$ gives the set of child nodes of v. $p(v)$ yields the set of parent nodes of v. $opt(v)$ specifies if v is optional under $p(v)$. This corresponds to the cardinality requirement of a node. It will be set to true for "*" and "?", and to false for "+" and "1". bar(v) corresponds to "|" in DTD and specifies if v takes only one of the child nodes at a time. A_d is a finite set of arcs. pc(v_1, v_2) if $e(v_1, v_2) \in A_d$. ad(v_1, v_2) if a path exists from v_1 to v_2. pc, ad specify parent-child and ancestor-descendant relationships between a pair of nodes (v_1, v_2), respectively. r_d specifies the root node, $r_d \in V_d$ and $p(r_d) = \phi$.*

Now, we formulate our adaptive query relaxation problem as follows: Given a WTPQ $q = T(V, E, r, w)$ and a set of DTDs d_1, d_2, $\cdots$, d_n, we would like to find a set of relaxed queries $Q = Q_1 \cup Q_2 \cdots \cup Q_n$, where Q_i is the set of queries conforming to d_i. To determine Q_i, we firstly relax q into a relaxed query q_i' that preserves maximum query requirements of q w.r.t. d_i. Based on query requirements, q_i' may be further relaxed into a set of queries Q_i according to the cardinality information, such as "*" in d_i. .

4.3 Adaptive Relaxation Rules

As discussed above, AQR relaxes a query for an XML data source based on its conformed DTD. Basically, AQR avoids blind relaxation by filtering out those query nodes that do not appear in the DTD and adjusting the node relationships if they do not match the DTD; AQR also avoids wild relaxation by preserving the query requirements which are definitely satisfied by the DTD. Before we introduce the set of adaptive relaxation rules for these purposes, we need the following definitions.

Definition 14 ***Corresponding Node****: Let a WTPQ $q = T(V, E, r, w)$ and a DTD $d = G(V_d, A_d, r_d)$, $v' \in V_d$ is called the corresponding node of a node $v \in V$ if $tagname(v') = tagname(v)$. A node in V may not always have a corresponding node.*

Definition 15 ***Consistent Corresponding Node****: Let a WTPQ $q = T(V, E, r, w)$, a DTD $d = G(V_d, A_d, r_d)$, and $r', v' \in V_d$ are the corresponding nodes of $r, v \in V$, respectively, v' is called the consistent corresponding node of a node v if either $pc(r', v')$ or $ad(r', v')$ holds. The corresponding node r' of the root node r is always consistent.*

For a WTPQ $q = T(V, E, r, w)$ and a DTD d, the minimal requirement for the relaxed query q' of q w.r.t. d is that the root node r of q keeps in q', i.e., r has a

corresponding node r' in d. For any other node $v \in V$, we need to check if it has a consistent corresponding node in d. Now we define rules for relaxation.

4.3.1 Ontology Relaxation

Before we discuss ontology relaxation, we assume that a tagname uniquely represents a concept and different tagnames representing same concept can be renamed as a single tagname, e.g., a worker can be renamed as an employee. With this assumption, a tagname can uniquely identify a node in a DTD. This means that a corresponding node defined in Definition 14 is unique if exists.

For each $v \in V$, if its corresponding node $v' \in V_d$ does not exist, we can search, say from WordNet, to see if there exists a superclass (hypernym) of $tagname(v)$ that matches the *tagname* of a node $v'_{super} \in V_d$ and v'_{super} is under the corresponding node of r. If so, the tagname of v is renamed as that of v'_{super} in the relaxed query q' for d. We call v'_{super} a *relaxed consistent corresponding node* of v. A fixed penalty applies for this kind of relaxation.

A query q can be relaxed to q' for d if either the corresponding node or the relaxed corresponding node of the root node r exists in V_d.

4.3.2 Node Relaxation

When a node $v \in V$ can not find a consistent corresponding node or a relaxed consistent corresponding node in V_d, v needs to be deleted in the relaxed query q' for d. The following two situations need to be treated differently.

Case 1: If the node is a *leaf* node, i.e., $ch(v) = \phi$, we can directly delete the node v and the edge $e(p(v), v)$ in the relaxed query q'.

Case 2: If the node is an *internal* node, i.e., $ch(v) \neq \phi$ and $v \neq r$, we first delete the node v and the edge $e(p(v), v)$, then for each $v_i \in ch(v)$, we replace the edge

$e(v, v_i)$ with an *ad*-edge $e'(p(v), v_i)$ in the relaxed query q' regardless of whether e is a *pc*-edge or an *ad*-edge.

Since a node (an element or attribute) that does not exist in a DTD will not appear in the corresponding data sources that conform to the DTD, the adoption of the *node relaxation* rule can delete the node in advance, rather than leave it in the query which later will be evaluated uselessly on these data sources. Consequently, unqualified relaxed queries will not be generated in AQR. That will reduce the number of relaxed queries generated and improve the performance of relaxed query evaluation. The deletion of a node in the query may affect the original query requests of a user. To mark this change in the relaxed query, a structural penalty applies depending on the importance of the relationships between the node and other nodes. This will be discussed in detail in Section 4.4.

4.3.3 Term Relaxation

Assume a node v contains search requirements for terms t_1, ..., t_m, i.e., $v[\tau(t_1)$ *and* ... $\tau(t_m)]$, and its parent node $v_p[\tau(t_{p1})$ *and* ... *and* $\tau(t_{pn})]$. If v is deleted, its terms will be merged to its parent resulting $v_p[\tau(t_{p1})$ *and* ... *and* $\tau(t_{pn})$ *and* $\tau(t_1)$ *and* ... *and* $\tau(t_m)]$, where t_i $(1 \leq i \leq m)$ does not match any of t_{pj} $(1 \leq j \leq n)$.

While query requirements on XML structure are important for finding accurate information, search requirements on terms are fundamental to most queries from users. When a node that contains a term search request is deleted, we can apply the above term relaxation rule to keep the term search request by promoting it to its parent using operator "*and*", rather than deleting it together with the node. The semantics of this term promotion widens the search scope from a child node to its parent node when the child node does not appear in the DTD.

Since penalty has been applied to a node relaxation, no extra penalty applies to a term relaxation.

4.3.4 Inconsistent Edge Relaxation

The inconsistent appearance of nodes between a query q and a DTD d is handled by a node relaxation rule. Now we consider each edge $e(v_1, v_2) \in E$ of q. In q, a user specifies either $pc(v_1, v_2)$ (pc-edge) or $ad(v_1, v_2)$ (ad-edge). In the relaxed query q' of q, we try to keep this relationship as close as possible. However, even though there is no such close relationship between v_1' and v_2' in d where v_1' and v_2' are consistent corresponding nodes of v_1 and v_2, it is better to keep v_1 and v_2 in q' than just delete them, which can keep the maximum semantics of the original query. So we have the following definition.

Definition 16 ***eSibling:*** *Given a pair of nodes (v_1, v_2), if neither $pc(v_1, v_2)$ or $ad(v_1, v_2)$, nor $pc(v_2, v_1)$ or $ad(v_2, v_1)$ holds, v_1 and v_2 are said to satisfy an extended sibling relationship and is denoted as $eSibling(v_1, v_2)$.*

Note, an *eSibling* relationship permits that v_1 and v_2 appear at different levels of a tree or DAG. Unlike *pc* or *ad* relationships, we have $eSibling(v_1, v_2) = eSibling(v_2, v_1)$.

Now we consider an edge $e(v_1, v_2) \in E$ of q, and assume that the consistent corresponding pair of nodes (v_1', v_2') of (v_1, v_2) exist in DTD d. If e is a pc-edge and $pc(v_1', v_2')$ also holds, or e is an ad-edge and either $pc(v_1', v_2')$ or $ad(v_1', v_2')$ holds in d, we can keep e as it is in q'; otherwise, we need to handle the follwing two situations.

Case 1: Relaxing a *pc-edge* into an *ad*-edge. If e is a pc-edge and only $ad(v_1', v_2')$ (not $pc(v_1', v_2')$) holds in d, e needs to be relaxed from a pc-edge to an ad-edge in q'.

Case 2: Relaxing a pc-edge or an ad-edge into an *eSibling* relationship. In d, if neither $pc(v_1', v_2')$ nor $ad(v_1', v_2')$ holds (i.e., $eSibling(v_1', v_2')$ or $pc(v_2', v_1')$ or $ad(v_2', v_1')$ holds), e will not appear in q', instead, a new edge will be created to make v_1 and v_2 to have $eSibling(v_1, v_2)$.

For Case 2, we need to know how to create the new edge, i.e., to connect v_2 under a *common ancestor* of v_1 and v_2 based on the positions of v'_1 and v'_2 in d. Again, we try to keep the relationship as close as possible in q' with respect to d so a *nearest* common ancestor is sought after. The following definition serves for this purpose.

Definition 17 ***Nearest Common Ancestor*** *(NCA): Let a WTPQ $q = T(V, E, r, w)$, a DTD $d = G(V_d, A_d, r_d)$, and $v_1, v_2 \in V$ are a pair of nodes in q. NCA(v_1, v_2, q, d) is defined as a node in V, denoted as v_{nca}, which satisfies all the following conditions: (1) $pc(v_{nca}, v_1)$ or $ad(v_{nca}, v_1)$, and $pc(v_{nca}, v_2)$ or $ad(v_{nca}, v_2)$ hold. (2) v_1, v_2 and v_{nca} all have their consistent corresponding nodes v'_1, v'_2 and v'_{nca} in d. (3) $pc(v'_{nca}, v'_1)$ or $ad(v'_{nca}, v'_1)$, and $pc(v'_{nca}, v'_2)$ or $ad(v'_{nca}, v'_2)$ hold. (4) $\neg\exists v_x \in V$ such that v_x satisfies the above conditions, and $pc(v'_{nca}, v'_x)$ or $ad(v'_{nca}, v'_x)$ holds.*

Theorem 2 ***Existence of NCA****: Let a WTPQ $q = T(V, E, r, w)$, a DTD $d = G(V_d, A_d, r_d)$, and $e(v_1, v_2) \in E$ in q. NCA(v_1, v_2, q, d) exists if and only if $\exists v'_1, v'_2, r' \in V_d$ such that they are the consistent corresponding nodes of $v_1, v_2, r \in V$, respectively.*

Proof of "$\leftarrow$": Since v'_1 and v'_2 are the consistent corresponding nodes of v_1 and v_2, respectively, by Definition 15, we know that both v'_1 and v'_2 are under the consistent corresponding node r' of r. In other words, r' is a common ancestor of v'_1 and v'_2. This guarantees that the NCA v_{nca} exists with the root node r serves as a guard.

Proof of "$\rightarrow$": Since v_{nca} exists, by Definition 17, we know that the consistent corresponding nodes v'_1 and v'_2 exist for v_1 and v_2, respectively, with the consistent corresponding node r' of r as the pre-condition.

The algorithm for finding an NCA v_{nca} of (v_1, v_2) will be introduced in Section 4.5. Now we can create a new *ad-edge* $e_n(v_{nca}, v_2)$ in q' to replace $e(v_1, v_2)$. Furthermore, to guarantee relaxation precision, we also move the child nodes of v_2 adaptively. For each node $v_i \in ch(v_2)$, if its consistent corresponding node v'_i exists, and either $pc(v'_2, v'_i)$ or

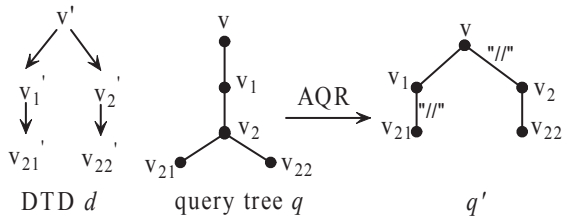


Figure 4.5: Inconsistent Relaxation

$ad(v_2', v_i')$ holds in d, no change is required to the edge $e_i(v_2, v_i)$; otherwise, $e_i(v_2, v_i)$ will be deleted and a new *ad-edge* $e_i(v_1, v_i)$ will be created.

We take the query tree q and DTD d shown in Figure 4.5 as an example. q needs to be relaxed using the inconsistent edge relaxation rule (Case 2) when v_2 is considered because the inconsistent relationship for (v_1, v_2) in q and d. Before moving node v_2 to put under node v - the NCA of v_1 and v_2, we first check the relationships between v_2 and v_2's child nodes v_{21} and v_{22}. Based on $pc(v_1'$, $v_{21}')$ and $pc(v_2'$, $v_{22}')$, v_{21} will be connected to v_1 and v_{22} will follow v_2, resulting the relaxed query q' shown in Figure 4.5.

Different penalties apply to Case 1 and Case 2 of the inconsistent edge relaxation rule. Detailed penalty computation will be discussed in Section 4.4.

4.3.5 Recursive Relationship Relaxation

In Definition 13, a DTD is defined as a directed acyclic graph. When a DTD includes some recursively defined elements, cycles may appear in the graph. Fortunately, cycles can be efficiently detected [89]. For the previous rules work properly, we can treat a cycle as a node for simplicity. However, when a user's query involves nodes in a cycle, proper processing is required. When a query path including nodes that appear in some cycles, we find that two patterns can be used for different treatments. One is *non-repetitive* while the other is *repetitive*. A node in a cycle only appears once

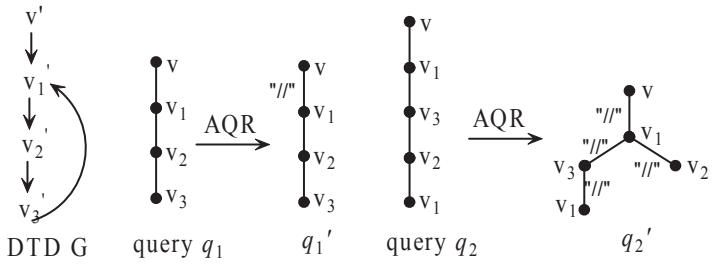


Figure 4.6: Recursive Relaxation

in a non-repetitive pattern while more than one in a repetitive one. The difference between these two patterns is that users expect recursively defined elements in source DTDs in the latter while not in the former. In either case, we call the first node that appear in both a cycle and the query as a *recursion start node* and the last node that connects to a recursion start node a *recursion entry node*. For example, v_1 and v are the recursion start and entry nodes, respectively in both queries q_1 and q_2 shown in Figure 4.6.

Case 1: Consider a non-repetitive pattern query q where the nodes in a cycle appear in the same sequence as that in the recursively defined DTD d. Let v_0 and v_1 be the recursion entry node and start node, respectively. If the edge $e(v_0, v_1)$ is a *pc-edge*, based on the assumption that the user is not aware of a recursively defined source DTD, $e(v_0, v_1)$ is relaxed to an *ad-edge* in the relaxed query q'. For example, the non-repetitive query q_1 is relaxed as q_1' in Figure 4.6.

Case 2: Consider a repetitive pattern query q where the nodes in a cycle appear in the same sequence as that in the DTD d but d does not include the corresponding cycle. We apply the node relaxation rule to retain only a single occurrence for those nodes that appear multiple times. If there are different term search conditions on each occurrence of the node, they are merged as the search conditions of the single occurrence of the node. This relaxation allows q to be evaluated on the sources that conform to d.

Case 3: Consider a repetitive pattern query q where the nodes in a cycle appear in the same sequence as that in the recursively defined DTD d. Let v_0 and v_1 be the recursion entry node and the recursion start node, respectively. Assume $v_1, \ldots, v_n, \ldots, v_1, \ldots, v_k$ $(k \leq n)$ the repetitive pattern, we relax the edge $e(v_0, v_1)$ the same as Case 1 and shorten the repetitive pattern to $v_1, \ldots, v_n, v_1, \ldots, v_k$ $(k \leq n)$.

Case 4: Consider either a non-repetitive or repetitive pattern q where the nodes in a cycle appear in a different sequence from that in the recursively defined DTD d. We can delete the inconsistent nodes using a node relaxation rule. Similarly we can adjust the inconsistent edges using the inconsistent edge relaxation rule with the difference that the recursion start node takes the role of the NCA node. For example, the repetitive query q_2 with the inconsistent node sequence compared with the DTD cycle is relaxed as q_2' in Figure 4.6.

Penalty allocations of the above relaxation rules are similar to that of non-recursive relaxation rules.

In summary, we adaptively relax users' queries based on source DTDs, which allows the relaxed queries to be more accurate and to preserve more in the original queries. Moreover, the relaxed queries are very succinct because the excess repeated nodes or edges are deleted or adjusted in advance by utilizing DTD information.

4.4 Weight and Penalty

In order to improve the precision of a user specified query, we allow users to assign weights to edges in the query to show their preferences for different paths. The weight information will serve as a foundation of associating each relaxed query with a reasonable penalty. Obviously, a less modified query with a low penalty is supposed to capture the user's original query aim more accurately.

4.4.1 Weight Model

Weights are assigned on edges in users' queries, as defined in Definition 12. With a weight specified on an edge $e(v_1, v_2)$, users can specify how close v_1 and v_2 are associated with each other. However, relaxing a query may call for structural changes in the query tree, weighted relationships between only adjacent nodes are inadequate to rectify edge weights and determine the penalty after a relaxation step, especially in node relaxation and inconsistent edge relaxation. For instance, when deleting a node v_1, new edge weight on $e(p(v_1), v_2)$ between v_1's parent $p(v_1)$ and v_1's child $v_2 \in ch(v_1)$ should be determined; similarly when moving a node v_2 under a new node v_{nca}=NCA(v_1, v_2, q, d), the relationship between v_{nca} and v_2 needs to be found out to determine the new edge weight for $e(v_{nca}, v_2)$. To this end, we introduce the concept of extended edge weight between a pair of nodes with *ad* relationship, $ad(v_i, v_j)$. The extended edge weight can be derived by multiplying weights along the path from v_i to v_j. If *eSibling*(v_i,v_j) holds, the extended edge weight for (v_i, v_j) is 0.

Definition 18 ***Extended Edge Weight:*** *Let a WTPQ $q = (V, E, r, w)$, for two nodes v_i, $v_j \in V$ $(i \neq j)$, the* extended edge weight *between v_i and v_j, denoted as $w_e(v_i, v_j)$, is defined as follows: if ad(v_i,v_j) or ad(v_j,v_i) holds, let w_1,w_2,...,w_n be weights on edges along the path from v_i to v_j, $w_e(v_i,v_j)=\prod_{t=1}^{n} w_t$; otherwise, $w_e(v_i,v_j)=0$. For convenience, define $w_e(v_i,v_i)=0$.*

According to the above definition, we find: (1) the more path steps there are between two nodes, the less related the two nodes may be; (2) nodes lying on different paths are not related with each other, i.e. nodes are only related with their ancestors and descendants. These two aspects reflect our common understandings about queries. With the concept of an extended edge weight, we can easily determine the weight between two nodes that will be connected by a new *ad*-edge. Extended edge

weights can be computed on the fly when required, or be calculated out beforehand, and then maintained dynamically.

Definition 19 ***Query Weight:*** *The whole weight of a query tree q,* query weight*, denoted as $w_q(q)$ is constructed by summing all extended edge weights:*

$w_q(q) = \sum_{\forall v_i, v_j \in V, ap(v_i, v_j)} w_e(v_i, v_j)$

where $ap(v_i, v_j)$ means either $ad(v_i, v_j)$ or $pc(v_i, v_j)$.

Query weight contains every extended edge weight, because the importance of a node in the query tree is reflected by the relationships between the node with all other nodes, not only with its parent node. Furthermore, it is obvious that query weight should be decreased after each relaxation step, since any relaxation that increases the value will be considered to add in additional relationships, and should not be carried out. This common sense is also implicitly reflected in our model.

Take the query q in Figure 4.1(a) as an example. Assume the weight of each edge is set as 1, we have $w_q(q) = w_e$(dept, group) + w_e(dept, project) + ... + w_e(group, project) + ... + w_e(project, pname) + w_e(publication, title) = 11.

4.4.2 Penalty Evaluation

A penalty needs to be determined when a query is relaxed into a new form, accompanied with possible weight modification. There are mainly three basic operations when utilizing rules to relax user's queries: (a) deleting a node, (b) relaxing a *pc*-edge into an *ad*-edge, and (c) moving a node. For example, (a) is used in node relaxation while all the three operations may be used in the recursive relationship relaxation.

Deleting a node: Let the deleted node be v, its parent node and one of the child nodes (if exists) be $p(v)$ and $v_c \in ch(v)$. In this case, the weight of the new *ad*-edge established between $p(v)$ and v_c will be assigned with their extended edge weight $w_e(p(v), v_c)$, i.e. $w(p(v), v_c) = w_e(p(v), v_c) = w(p(v), v) \cdot w(v, v_c)$. Moreover,

as the relationships between v and all the other nodes in the query disappear after this operation, the penalty is calculated as:

$$f_1 = \sum_{\forall v_i \in V, v \neq v_i} w_e(v, v_i) \tag{4.1}$$

After weight modification on the new edge, extended edge weight between any pair of nodes in the relaxed query keeps unchanged. And deleting a node at the conjunction of two branches may lead to heavier penalty than deleting a node on a single path. This also accords with common sense.

Relaxing a *pc*-edge into an *ad*-edge: Let the inconsistent *pc*-edge be $e(v_1,v_2)$. $pc(v_1, v_2)$ needs to be relaxed into $ad(v_1, v_2)$. Edge weight $w(v_1, v_2)$ will be reduced to $\lambda \cdot w(v_1, v_2)$, where $\lambda \in [0, 1]$ is a coefficient specified by users. λ shows, to what extent, a *pc* relationship in the query can be taken place by an *ad* relationship. Now considering v_2's ancestor v_i, v_1's descendant v_j, relationship between v_i and v_j is sort of weakened, as $e(v_1,v_2)$ along the path between them is relaxed; while such variation should not affect the other paths. And the penalty of this operation is:

$$f_2 = (1 - \lambda) \cdot \sum \{w_e(v_i, v_j) | (\forall i \forall j, ad(v_i, v_2) \wedge ad(v_1, v_j))\} \tag{4.2}$$

After weight modification on the edge, the extended edge weight between v_i and v_j, satisfying $ad(v_i, v_2) \wedge ad(v_1, v_j)$, will be changed.

Moving a node: Let the inconsistent edge and their NCA be $e(v_1,v_2)$ and v_{nca}, and let the path from v_{nca} to v_1 be v_{nca},v_{i_n},$\cdots$,v_{i_1},v_1. The new established *ad*-edge weight $w(v_{nca}, v_2)$ will be assigned with $w_e(v_{nca}, v_2)$. And v_2 will lose relationship with the nodes between v_{nca} and v_2. The penalty of moving node v_2 exclusively (without

considering descendants of v_2) is:

$$f_{v_2} = w_e(v_1, v_2) + \sum_{t=1}^{n} w_e(v_{i_t}, v_2) \tag{4.3}$$

However, as is discussed in Section 4.3.4 Case 2, for any subtree rooted at $v_{2c} \in ch(v_2)$, it may remain under v_1 or follow v_2 after inconsistent edge relaxation. Hence further weight modification and penalty evaluation about the subtree are triggered according to different cases. Let the subtree rooted at v_{2c} be q_s, the node set of the subtree q_s be V_s, the penalty related with subtree q_s be f_s, and penalties of all subtrees be $\sum f_s$, we discuss further adjustments as follows:

Case 1: subtree q_s rooted at v_{2c} will remain under v_1: new edge weight established between v_1 and v_{2c} is $w(v_1,v_{2c}) = w_e(v_1,v_{2c})$. Further penalty is the deletion of relationships between v_2 and nodes in V_s, for v_2 is also separated away from q_s.

$$f_s = \sum_{\forall v_j \in V_s} w_e(v_2, v_j) \tag{4.4}$$

Case 2: subtree q_s rooted at v_{2c} will follow v_2: No weight modification on edges are required for nodes in V_s. As to penalty, different from the above case, relationships between nodes in V_s and nodes along path v_{i_n} to v_1 are removed, while relationships between nodes in V_s and v_2 are kept.

$$f_s = \sum_{\forall v_j \in V_s} w_e(v_1, v_j) + \sum_{\forall v_j \in V_s} \sum_{t=1}^{n} w_e(v_{i_t}, v_j) \tag{4.5}$$

After edge weight modification, extended edge weights between the nodes separated into two branches are also needed to be rectified to 0. Total penalty of moving a node is the sum of two parts:

$$f_3 = f_{v_2} + \sum f_s \tag{4.6}$$

Two properties can be inferred on the penalty for moving a node:

Property 1 *Moving a node to an upper level will cause more penalty than to a lower level.*

Proof: Let v_2 be moved to v_{i_m} or v_{i_n} and the path from v_{i_m} to v_2 be v_{i_m}, $\cdots$, $v_{i_{n+1}}$, v_{i_n}, $\cdots$, v_{i_1}, v_1, v_2 ($m > n \geq 1$). Suppose v_2 has a single child v_{2c} (multiple children case can be proved similarly), let $\Delta f = f_3(v_2 \rightarrow v_{i_m}) - f_3(v_2 \rightarrow v_{i_n})$, both in case 1 and case 2, based on Equations 4.3, 4.4, 4.5, 4.6, we have $\Delta f > 0$.

$$\Delta f = \begin{cases} \sum_{t=n}^{m-1} w_e(v_{i_t}, v_2) & case1 \\ \sum_{t=n}^{m-1} w_e(v_{i_t}, v_2) + \sum_{\forall v_j \in V_s} \sum_{t=n}^{m-1} w_e(v_{i_t}, v_j) & case2 \end{cases}$$

Property 2 *Deleting a node will be more penalized than moving a node without considering child distribution.*

Proof: Let the path from v_{nca} to v_2 be v_{nca},v_{i_n},$\cdots$,v_{i_1},v_1,v_2. When v_2 is moved under v_{nca}, the difference between deleting v_2 and moving v_2 based on Equations 4.1, 4.3 is:

$$\Delta f = \sum_{\forall v_m, ad(v_m, v_{nca})} w_e(v_m, v_2) + w_e(v_{nca}, v_2) > 0$$

In Section 4.4.1, we have computed $w_q(q) = 11$ for the weight of the original query q, where the weight of each edge is set as 1. When q is relaxed to q_{11}, node *publication* is moved under node *dept* based on the inconsistent edge relaxation rule. During the procedure of relaxation, we compute the penalty with Equation 4.6 in which f_s is calculated with Equation 4.5. In this case, we have $f_{publication} = w_e(group, publication) = 1$ and $f_s = w_e(group, title) = 1$ since the edges *group*/*publication* and *group*/*title* are lost. Therefore, the weight of q_{11} can be computed as $w_q(q_{11}) = 11 - 2 = 9$. Similarly, we have $w_q(q_{12}) = 6$. When q is relaxed to q_{21}, the *pc*-edge between node *dept* and node *group* is relaxed to an *ad*-edge. Assume the coefficient λ is set to 0.9, the penalty of this relaxation can be computed as 0.3 with Equation 4.2. Similar to

q_{11}, the penalty caused by moving node *project* under node *dept* is also 2. So $w_q(q_{21})$ = 11 - 0.3 - 2 = 8.7. Similarly we have $w_q(q_{22}) = 5.7$. Now we can sort these four relaxed queries and get the ranking list [q_{11}, q_{21}, q_{12}, q_{22}]. However, if a smaller weight 0.5 is specified on the edge between *group* and *project* while others remain the weight 1, we can recompute the weights of these four relaxed queries and have $w_q(q_{11}) = 7$, $w_q(q_{12}) = 4$, $w_q(q_{21}) = 7.7$, $w_q(q_{22}) = 5.7$. Consequently, the ranking list is changed to [q_{21}, q_{11}, q_{22}, q_{12}].

4.4.3 Penalty Fitness Discussion

Evaluating penalty is subject to different weight models and penalty plans. It is difficult to reach a consensus for finding out a perfect model. In the former section, we have shown that our penalty deduced is sound and suits real cases by justifying the possession of some basic qualities that a good penalty strategy should provide. We will now give two other features with respect to weight and penalty rendered on an overall basis.

Free order of processing eSibling nodes: Query relaxation is done in a top-down manner. After processing node v, the order of processing nodes in $ch(v)$ should not make any difference. More generally speaking, for any two nodes of *eSibling* relationship, processing order should also be free. Suppose v_i, v_j be a pair of *eSibling* nodes, basic operations on v_i will affect v_j's extended edge weights. Then from penalty evaluation of basic operations about v_i, affected edge weights may be $w(p(v_i), v_i)$, $w(p(v_i), v_c)$, $w(v_{nca}, v_i)$, where $v_c \in ch(v_i)$ and v_{nca} is the NCA which v_i will be moved under. Given another node v_k, let $w_e(v_j, v_k)$ be affected by one of the edge weights above, then from the definition of the extended edge weight, there should exist an edge $e \in S = \{(p(v_i), v_i), (p(v_i), v_c), (v_{nca}, v_i)\}$ on the path from v_j to v_k. No matter which edge $e \in S$ is on the path, we will have $ad(v_i, v_j)$ or $ad(v_j, v_i)$. This contradicts to the supposition. Thus operations on v_i cannot affect v_j's extended edge weights,

and processing *eSibling* nodes is of free order.

Equal penalty for step-by-step computation and batch computation: Penalty can be computed at each step and accumulated together, or evaluated by comparing *query weight* between the final relaxed query and the original user specified query. Our model conforms to the fact well, as penalty is regarded as losing relationships and is implicitly defined by the decrease of query weight, in which all relationships between nodes in user's query are contained. An interesting question is in which way, penalty can be computed efficiently. Unluckily, the answer is not definite. Step-by-step computation suits the situation where few relaxation occurs, while batch overwhelms the former in case of more relaxation steps. In practice, it is difficult to determine how much relaxation will be done on the original query in advance. We reserve the problem as future work.

4.5 Adaptive Relaxation Process

Given a user's query q and a certain DTD d_i, we can relax q and generate a set of relaxed queries Q_i using the relaxation rules discussed in Section 4.3. Alternatively, we can first relax q and generate a relaxed query q_i that preserves maximum query requirements of q w.r.t. d_i. Then we can further relax q_i when needed, based on the cardinality and disjunctive information provided in d_i. In this section, we introduce the relaxation algorithm for generating q_i from q w.r.t. d_i.

4.5.1 AQR Algorithm

Our main algorithm relaxes the query tree q in a top-down manner, i.e. relaxation operations carried out on a certain node v are always handled before the operations of $p(v)$. This guarantees that edge adjustments brought in by later processed nodes will not violate the established relationships between earlier processed nodes. The

Algorithm 6 Adaptive Query Relaxation

input: q=$T(V, E, r, w)$, $w_e(q)$ and d=$G(V_d, E_d, r_d)$
output: relaxed query q' and $w_e(q')$

```
1: globalQueue.addElement(r); f=0;
2: while globalQueue is not empty do
3:    v = globalQueue.pop();
4:    v' = FindConsistentDTDNode(v, q, d);
5:    if v' ≠ NIL then
6:       if v' ∈ RecursiveTable then
7:          Taking v_p = p(v), v, q, v', and d as input to call Algorithm 8;
8:       else
9:          if existAncestor(v) then
10:            globalQueue.addElement(each v_c ∈ ch(v));
11:            processing repeated node with Case 2 in Section 4.3.5 while computing
               penalty f_1 with Equation 4.1 and f = f + f_x;
12:         else if pc(p(v), v) ∧ ad(p(v)', v') then
13:            relaxing pc(p(v), v) with Case 1 in Section 4.3.4 while computing penalty
               f_2 with Equation 4.2 and f = f + f_2;
14:            globalQueue.addElement(each v_c ∈ ch(v));
15:         else if
            pc(v', p(v)') ∨ ad(v', p(v)') ∨ eSibling(v', p(v)') then
16:            Taking v_p = p(v), v, v'_p = p(v)', v', q and d as input to call algorithm 9;
17:         end if
18:      end if
19:   end if
20: end while
21: return q and w_e(q') = w_e(q) - f;
```

implied subtree constituted by the relaxed nodes always conforms to the DTD during the whole relaxation process, which reflects the correctness of our AQR algorithm. The query tree is scanned in a way similar to breadth-first traversal. For each node in a query tree, ontology relaxation, node relaxation, recursive relationship relaxation and inconsistent edge relaxation are checked in order. Here recursive relationship relaxation may trigger other relaxation rules as well.

Algorithm 6 gives the whole relaxation process. The queue *globalQueue* is used to hold nodes waiting to be processed. At the beginning of each loop, globalQueue.pop()

Algorithm 7 Find Consistent Corresponding Node *FindConsistentDTDNode*()

input: current node v, DTD d, query q' and globalQueue
output: the consistent corresponding node of v

```
1: if checkConsistentNode(v, q', d) then
2:    return getConsistentDTDNode(v, q', d);
3: else
4:    Get the superclass node v_super of v;
5:    if checkConsistentNode(v_super, q', d) then
6:       Relaxing v with ontology relaxation rule in Section 4.3.1 and recording the
         corresponding penalty into f;
7:       return getConsistentDTDNode(v_super, q', d);
8:    else
9:       globalQueue.addElement(each v_c ∈ ch(v));
10:      Relaxing v with node relaxation rule in Section 4.3.2 and recording the cor-
         responding penalty into f;
11:      return NIL;
12:   end if
13: end if
```

serves a node to be relaxed. Firstly, we try to find its consistent corresponding node v' in DTD d using function *FindConsistentDTDNode*() in Algorithm 7. Node relaxation will apply when neither a consistent corresponding node is found in DTD, nor a superclass node is sought out. If v' can be found and is in a recursive circle in d, Algorithm 8 is called to relax recursive relationships. Otherwise, there are three other relaxation cases : (1) A repetitive pattern appears in the query tree with recursion start node v, but no recursive cycle including v' exists in DTD. Based on Case 2 in Section 4.3.5, we delete the repetitive node in the query if the node has already appeared once on the path from v to r (detected by function existAncestor(v)). (2) If node v is not repetitive, and there is an inconsistent edge between $p(v)$ and v on condition that $pc(p(v), v)$ holds in q but $ad(p(v)', v')$ in d, we will relax the *pc*-edge into *ad*-edge based on Case 1 in Section 4.3.4; (3) If node v is not repetitive and the inconsistent edge between v and $p(v)$ satisfies $pc(v', p(v)')$ or $ad(v', p(v)')$ or $eSibling(v', p(v)')$, we call Algorithm 9 to promote node v under the NCA of $p(v)$ and

Algorithm 8 Relaxing recursive relationship

input: entry node v_p, start node v, current query q', consistent corresponding nodes v' and DTD d
output: relaxed query q'

```
1: if pc(v_p, v) then
2:    relaxing pc edge with Case 1 in Section 4.3.5 while computing penalty f_2 with
      Equation 4.2 and f = f + f_2;
3: end if
4: qPatternBase = getPatternBase(v_p, v, q');
5: qPatternTail = getPatternTail(v_p, v, q');
6: while nextPattern(qPatternBase)≠qPatternTail do
7:    deleteNextPattern(qPatternBase) and compute penalty and add it into f;
8: end while
9: Connect qPatternBase to qPatternTail with ad edge;
10: DTDPattern = getDTDPattern(v', d);
11: if ¬exactMatch(qPatternBase, DTDPattern) then
12:    Call Case 4 in Section 4.3.5 and compute penalty and add it into f;
13: end if
14: return q';
```

v. In this way, we continuously deal with the nodes in *globalQueue* until the queue is empty. Finally, the relaxed query q' and its query weight $w_e(q')$ can be obtained.

Algorithm 7, containing ontology relaxation and node relaxation, is actually a function. It firstly check if the consistent corresponding node v' of v exists in d by calling *checkConsistentNode*(). If does, the node v' will be obtained by function *getConsistentDTDNode*(). Otherwise, it will continue to check if the superclass node v_{super} of v exists in d. After making sure v_{super} can be found in d, node v will be relaxed with ontology relaxation rule in Section 4.3.1 and return the consistent corresponding node v'_{super} of v_{super}. If no match either, node relaxation rule in Section 4.3.2 will be revoked and NIL will be returned.

4.5.2 Relaxing Recursive Relationship

Algorithm 8 handles recursive relationship relaxation. Let v_p and v be the recursion entry node and recursion start node in current query tree q', respectively. v' is the consistent corresponding node of v in DTD d. We firstly check the relationship of node pair (v_p, v). If $pc(v_p, v)$ holds in the query, we will generalize the *pc*-edge between v_p and v into an *ad*-edge based on Case 1 in Section 4.3.5. Then we determine the first repetitive pattern *qPatternBase* and the last part *qPatternTail* in query q' by calling functions *getPatternBase*() and *getPatternTail*(), respectively. After that, we delete the repetitive pattern between *qPatternBase* and *qPatternTail* based on Case 3 in Section 4.3.5 if the pattern is repeated for many times. Moreover, we create an *ad*-edge to connect *qPatternBase* with *qPatternTail*. Finally, we use function *getDTDPattern*() to obtain the current recursive pattern *DTDPattern* starting from node v' in DTD d. If *qPatternBase* and *DTDPattern* do not match exactly, we follow Case 4 in Section 4.3.5 to relax *qPatternBase* against *DTDPattern* using other relaxation rules. During the above process, the corresponding penalties will be computed and recorded.

4.5.3 Determining NCA and Moving Nodes

Algorithm 9 describes the procedure to determine the NCA v_{nca} for node pair (v_1, v_2) in query q and DTD d, and promote node v_2 as a descendant under v_{nca}. We begin to search v_{nca} from the parent node of v_1 using the function *checkCommonAncestor*(v', v'_1, v'_2, d) to check if the node v' satisfies *pc* or *ad* relationship with v'_1 and v'_2 in DTD d. Fortunately, lots of work can serve to efficiently judge the *pc* or *ad* relationship between any two nodes in a DTD DAG, in this chapter, we use an auxiliary structure called reachability matrix in [28, 32] to solve the problem. After v_{nca} is found, we further distribute the child nodes of v_2 according to different relationships between the child nodes to v_1 and to v_2. For each child node $v_{2c} \in ch(v_2)$, after checking the existence of its consistent corresponding node in DTD, we promote the subtree

Algorithm 9 Determine NCA and Move Nodes

input: a pair of nodes v_1, v_2 and their consistent corresponding nodes v'_1, v'_2, query q, source DTD d and globalQueue
output: relaxed query q'

```
1: v = p(v1);
2: v' = getConsistentDTDNode(v, q, d);
3: while ¬checkCommonAncestor(v', v'1, v'2, d) do
4:    v = p(v);
5:    v' = getConsistentDTDNode(v, q, d);
6: end while
7: for all v2c ∈ ch(v2) do
8:    v'2c = FindConsistentDTDNode(v2c, q', d);
9:    if v'2c ≠ NIL then
10:      if ¬pc(v'2, v'2c) ∧ ¬ad(v'2, v'2c) then
11:         Connect subtree rooted at v2c to v1;
12:      end if
13:      globalQueue.addElement(v2c);
14:   end if
15: end for
16: delete e(v1, v2), move the subtree rooted at v2 to NCA v and compute penalty
    and add it into f;
17: return q';
```

rooted at v_{2c} under v_1, if neither $pc(v'_2, v'_{2c})$ nor $ad(v'_2, v'_{2c})$ holds in DTD; otherwise we move the subtree together with v_2.

4.5.4 Generation of Relaxed Query Set

From the definition of DTD, we know "|" denotes disjunction semantics, and cardinality information, such as "*", "?", declares that a child element may occur zero/many times or zero/one time under its parent element. Therefore, documents conforming to the same DTD may also have different structures depending on the three signs. We omit the discussions about "+", for its semantics don't affect the form of the query tree q'. In this section, we continuously relax the intermediate query q' produced by the above algorithms into a set of sub queries Q by enumerating different semantic

combinations of the three signs. Precisely speaking, if "a|b" appears in a DTD, we keep either "a" or "b" for the relaxed queries in Q; if cardinality information "*", "?" w.r.t. a node v in q' also appears in a DTD, we add an additional query q'' with the subtree rooted at v in q' deleted.

At this stage, we utilize a DAG to maintain the set of relaxed queries, which are represented as nodes in the DAG, similar to the work [9]. However, the advantage of our work lies in that no unqualified queries to the current relaxation reference DTD will be generated. The size of our DAG depends on the real needs, rather than the number of elements in the original query.

Algorithm 10 illustrates the procedure of building a DAG. We firstly insert the relaxed query q' as DAG root node and push it into stack s. In each loop, we pop a query q from stack and seek out its corresponding DAG node *currentDAGNode*. For each node v in the query tree q, we check the information about "|", "*" or "?" in v's consistent corresponding node v': If $bar(v')$ is true and v has more than one child nodes in the query q, we call function *BarProcess*() to generate a set of new relaxed queries *newQuerySet*, where every relaxed query contains only one child node under v. After that, we check if each $q_i \in newQuerySet$ exists in current DAG by calling function *addQueryToDAG*. If does, we only need to set up an arc from *currentDAGNode* to the *existNode* using function *ToConnect*(). Otherwise, q_i will be inserted to DAG as a child of *currentDAGNode*. Let v_c be a child node of v. If $opt(v', v'_c)$ is true, implying "*" or "?" on v'_c, we directly generate a new query q_{new} by deleting the subtree rooted at v_c in q using function *DeleteProcess*(), and add it into DAG with checking its existence beforehand. Finally, we get a set of relaxed queries Q represented as nodes in the DAG.

Algorithm 10 Generating Relaxed Query Set

input: query q', a DTD d
output: A set of relaxed queries managed by DAG

```
1: SetDAGRoot(q');
2: push q' in stack s;
3: while stack s is not empty do
4:    pop a query q from s;
5:    currentDAGNode = GetDAGNode(q, DAG);
6:    for all node v in q do
7:       v' = FindConsistentDTDNode(v, q, d) by calling for Algorithm 7;
8:       if bar(v') and |ch(v)| > 1 then
9:          newQuerySet = BarProcess(q, v);
10:         addQueryToDAG(each q_i ∈ newQuerySet, currentDAGNode) by calling
            for the following function;
11:      else
12:         for all v_c ∈ ch(v) do
13:            v'_c = FindConsistentDTDNode(v_c, q, d);
14:            if opt(v', v'_c) then
15:               q_new= DeleteProcess(q, v_c);
16:               addQueryToDAG(q_new, currentDAGNode);
17:            end if
18:         end for
19:      end if
20:   end for
21: end while
```

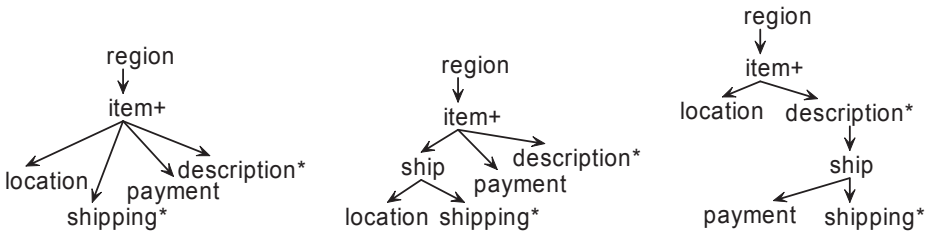
Function addQueryToDAG(query q, currentDAGNode);

```
1: if CheckQueryInDAG(q, DAG) then
2:    existNode = GetDAGNode(q, DAG);
3:    ToConnect(currentDAGNode, existNode);
4: else
5:    insert q as a child to currentDAGNode;
6:    push q into stack s;
7: end if
```

4.6 Experimental Results

We ran the experiments on an Intel P4 3GHz PC with 512M memory. Wutka DTDparser [2] was used to analyze the source DTDs and extract their structural information. All relaxed queries were evaluated as XPath patterns in Oracle Berkeley DB XML [1].

Dataset and Queries: We used XMark XML data generator [3] to create a set of XML documents with different size from 5MB to 40MB, which conform to *auction.dtd* [80]. These XML documents can be used to test the efficiency of AQR. In order to compare the effectiveness between AQR and FleXPath, we selected the 10MB document as a base to generate other three 10MB documents that have different structures. The generated three documents conform to the DTDs d_1 in Figure 4.7(a), d_2 in Figure 4.7(b) and d_3 in Figure 4.7(c), respectively. To keep readers clear, the three documents are named as xmark10MBd1.xml, xmark10MBd2.xml and xmark10MBd3.xml, respectively.



(a) Schema d_1 (b) Schema d_2 (c) Schema d_3

Figure 4.7: Partial Schemas of Generated 10MB Documents

With the structures of the three DTDs in mind, we first designed a query q with some keywords as follows.

- q: //item[./description [./payment. contains('Creditcard') and ./ship [./location. contains('United States') and ./shipping. contains('international')]]]

This query implies that the users are interested in the items: (1) the returned items can be paid with credit card; (2) the returned items can be shipped world wide, i.e., international; and (3) the returned items maybe export to or import from United States. Due to the different structures between the query and the three DTDs, the query q will be relaxed differently for each document in AQR. By analyzing the relaxed queries and corresponding answers, we can show the benefits of AQR in an intuitive way.

To study the efficiency of AQR, we further designed a set of queries with complex structures based on *auction.dtd* as follows.

Table 4.1: Designed Queries for Testing Relaxation

q_1:	//item [./description /parlist and ./mailbox /mail]
q_2:	//item [./description /parlist /mailbox /mail [./text and ./from and ./to]]
q_3:	//item [./description /parlist /listitem and ./mailbox /mail /text[./keyword and ./emph] and ./name and ./payment]
q_4:	//item [./description /xxx /yyy and ./mailbox /mail /text[./keyword and ./emph] and ./name and ./payment]
q_5:	//item[./description [./xxx /yyy and ./mailbox /mail /text [./keyword and ./emph]] and ./name and ./payment]
q_6:	//item[./description [./xxx /yyy and ./mailbox /mail /text [./keyword /keyword /keyword and ./emph /xxx]] and ./name and ./payment]

In these queries in Table 4.1, we focused on the structural requirements by considering the structural difference between the queries and the DTD, such as the edge "parlist/mailbox" does not exist in the DTD, the edge "description/parlist" satisfies disjunctive semantics and the nodes "mail" and "text" satisfy optional semantics. We further added "xxx" and "yyy" into some of the queries as noise nodes that do not appear in the DTD. We would like to guarantee certain level of computational scale so keywords are not included.

		k=39	k=46	k=55	k=60
xmark10MBd1	FleXPath	6 queries	7 queries	10~15 queries	17~27 queries but no new results
	AQR	1 queries	2 queries	3 queries	no new queries
xmark10MBd2	FleXPath	5 queries	6~8 queries	9~15 queries	22~32 queries but no new results
	AQR	1 queries	2 queries	3 queries	no new queries
xmark10MBd3	FleXPath	4~7 queries	no new queries	15~20 queries with 1 new result	no new queries
	AQR	1 queries	no new queries	2 queries with 1 new result	no new queries

Figure 4.8: Relaxed queries of q over different documents with the increase of k

4.6.1 Effectiveness of Relaxation

To demonstrate the effectiveness of AQR, we compared it with FleXPath to show the advantage of leveraging DTD in relaxing users' queries. For the query q designed for testing effectiveness, the maximum number of results is the same as that of the answers when we evaluate the simple query "//item['*United States*', '*Creditcard*', '*international*']", which returns those *item* nodes containing all the three keywords. To check if a relaxed query will generate further results, we checked these documents and found that the maximum number of results are 55, 55, 46 for *xmark10MBd1*, *xmark10MBd2* and *xmark10MBd3*, respectively. Figure 4.8 shows the number of relaxed queries to be generated with the increase of k value when we evaluate q as a top-k query over the three documents.

When $k = 39$, FleXPath has to generate 6, 5, 4∼7 relaxed queries and evaluate all of them over the three XML documents, respectively. However, only one relaxed query needs to be generated and evaluated for each document with AQR. When $k = 46$, FleXPath needs to generate 7 relaxed queries for *xmark10MBd1*, 6∼8 relaxed queries for *xmark10MBd2* and 4∼7 relaxed queries for *xmark10MBd3*. However, AQR only generates and evaluates 2, 2 and 1 relaxed queries for the three XML documents

to obtain the same number of results. Interestingly, we find that the numbers of relaxed queries generated over *xmark10MBd3* for k=39 and k=46 are the same. This is because the relaxed queries generated for k=39 can also be used for k=46, i.e., to return top 46 results. Compared with $k = 39$ and $k = 46$, FleXPath generates a lot more new relaxed queries for k=55, most of them either return no new result or output the same results repeatedly. In contrast, AQR only generates 1 new relaxed query for each document, but produces the same set of results as FleXPath. When we process *xmark10MBd3* for k=55, both approaches return no new result, yet generate new relaxed queries. However, AQR only generates 1 new query while FleXPath generates 8∼16 new queries.

When k=60, no new result can be found from all the three documents by both approaches because the current value of k (i.e., 60) is larger than the maximum number of results (i.e., 55) for all three documents. In this case, AQR stops generating new queries while FleXPath continues generating 2∼17 and 7∼23 new queries for *xmark10MBd1* and *xmark10MBd2*, respectively.

From the experiments, we find that FleXPath generates far more relaxed queries compared with AQR. The reason behind this finding is that to meet a large k, FleXPath has to relax a user's query and evaluate it until the root node of the query if necessary. However, AQR is able to stop unnecessary query relaxations early as possible for a particular data source with the guideline of its conformed DTD.

4.6.2 Validation of Weight Modification

In this section, we will show the impact of the specified query weight on the returned search results. Let's take the query in Figure 4.9 hiding the term information from q and two of the source schemas d_2 in Figure 4.7(b), d_3 in Figure 4.7(c) as an example. The relaxed queries with AQR are illustrated in Figure 4.10 and Figure 4.11.

Firstly, assume all the edges in the query q in Figure 4.9 hold the default edge

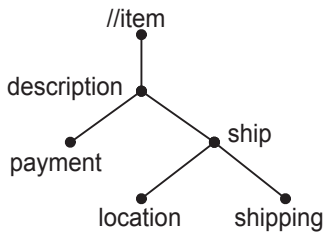


Figure 4.9: Query q with the default edge weight 1.0

weight value 1. According to the AQR algorithm, we can get the most relevant relaxed queries q_{21} and q_{31} for the corresponding schemas d_2 and d_3, respectively. The weight of the query q_{21} is 7 while the weight of the query q_{31} is 9. Therefore, we should evaluate the query q_{31} before the query q_{21} since the query with higher query weight preserves more request information of the original query than the query with less query weight. After we evaluate the query q_{31}, we may continuously relax the query q_{31} into the query q_{32} by deleting the node *shipping* if more results are required for users. The weight of the new relaxed query q_{32} is 6 that is less than that of the query q_{21}. Subsequently, we need to evaluate the query q_{21}. Based on the similar procedure, we can get a fixed order to process the relaxed queries. In this case, the sequence is q_{31} (weight=9) $\rightarrow$ q_{21} (weight=7) $\rightarrow$ q_{32} (weight=6) or q_{22} (weight=6) $\rightarrow$ q_{23} (weight=5) $\rightarrow$ q_{24} (weight=4) $\rightarrow$ q_{33} (weight=1).

Now let's modify the weight of edge (e.g., ship/shipping) in the query. To make the description simple, the two queries in Figure 4.12 and Figure 4.13 express different users' preferences from the query in Figure 4.9. Although the same set of relaxed queries will be generated with AQR, which are shown in Figure 4.10 and Figure 4.11, the query with different edge weights may produce different search results due to different query processing order. q_{21} and q_{31} are still the most relevant relaxed queries with regards to schemas d_2 and d_3, respectively.

For the query in Figure 4.12, the weight of the query q_{21} is 5.2 while the weight of

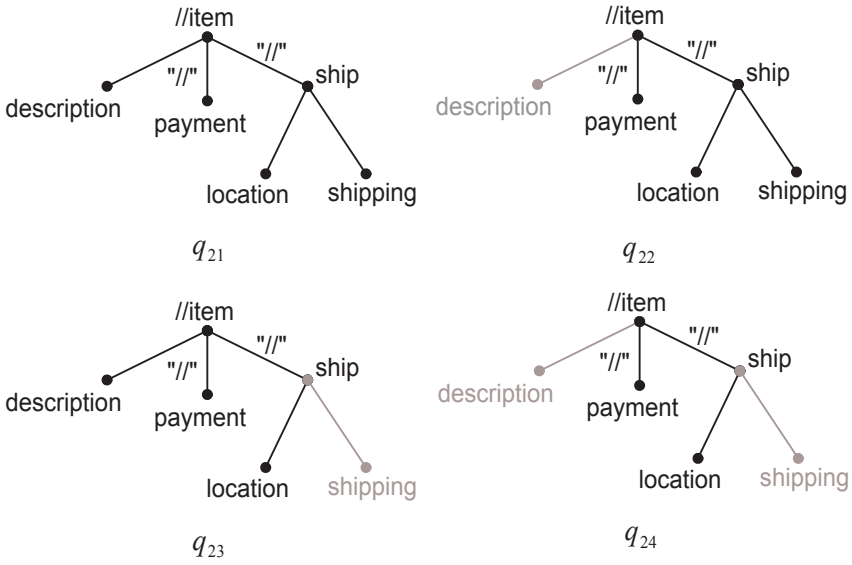


Figure 4.10: Relaxed queries w.r.t. schema d_2

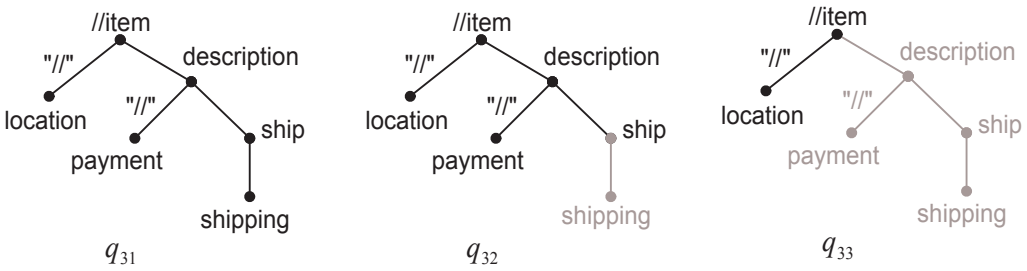


Figure 4.11: Relaxed queries w.r.t. schema d_3

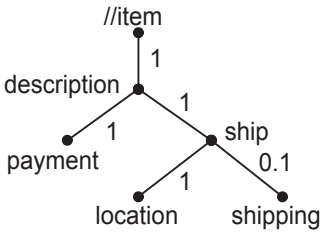


Figure 4.12: Query q_1 with $w_e(ship, shipping)$=0.1

the query q_{31} is 6.1 (we assume $\lambda = 0.9$ that means the edge weight of description//-payment will be reduced from 1 to 0.9 due to edge relaxation pc-to-ad). Therefore, we should evaluate the query q_{31} before the query q_{21}. If required, we may continuously relax the query q_{31} into the query q_{32} by deleting the node *shipping*. The weight of the new relaxed query q_{32} is 5.8 that is still larger than that of the query q_{21}. Subsequently, q_{32} starts to be evaluated. If more results are needed, we may relax the query q_{32} to the query q_{33} whose weight is 1. At the next step, it turns to evaluate q_{21} to get more results. Based on the similar procedure, we can get a different fixed order to process the relaxed queries, which is q_{31} (weight=6.1) $\rightarrow$ q_{32} (weight=5.8) $\rightarrow$ q_{21} (weight=5.2) $\rightarrow$ q_{23} (weight=5) $\rightarrow$ q_{22} (weight=4.2) $\rightarrow$ q_{24} (weight=4) $\rightarrow$ q_{33} (weight=1).

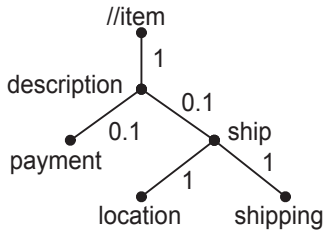


Figure 4.13: Query q_2 with $w_e(description, payment) = w_e(description, ship) = 0.1$

For the query in Figure 4.13, the weight of the query q_{21} is 3.4 while the weight of the query q_{31} is 2.68 (we assume $\lambda = 0.9$ that means the edge weight of description//payment will be reduced from 0.1 to 0.09 due to edge relaxation pc-to-ad). Therefore, we should evaluate the query q_{21} before the query q_{31}. If required, we may continuously relax the query q_{31} into the query q_{32} by deleting the node *shipping*. The weight of the new relaxed query q_{32} is 1.48 that is less than that of the query q_{21}. Subsequently, we need to evaluate the query q_{21}. If more results are needed, we may relax the query q_{21} to the query q_{22} with the weight 2.4. Because the query q_{22} (weight = 2.4) has higher query weight than the query q_{32} (weight = 1.48), we continue to

evaluate the query q_{22}. Based on the similar procedure, we can get a different fixed order to process the relaxed queries, which is q_{21} (weight=3.4) → q_{31} (weight=2.68) → q_{22} (weight=2.4) → q_{23} (weight=2.3) → q_{32} (weight=1.48) → q_{24} (weight=1.3) → q_{33} (weight=0.1).

AQR		k=39	k=40	k=86	k=93
q	xmark10MBd2	q31	q21	q32	q22
	xmark10MBd3				
q_1	xmark10MBd2	q31	q32	q21	q23
	xmark10MBd3				
q_2	xmark10MBd2	q21	q31	q22	q23
	xmark10MBd3				

Figure 4.14: Result comparison with the increase of k when the edge weights are assigned differently

Figure 4.14 shows that when k = 39, q and q_1 can generate the same results conforming to the structure of q_{31} while q_2 will generate another set of results conforming to the structure of q_{21}. Similarly, we find that when k = 40, q and q_2 have the same results while q_1 does not; when k = 86, q and q_1 have the same results while q_2 does not again; when k creases to 93, the three queries q, q_1 and q_2 will have different sets of search results, respectively. Therefore, we can say that a query with different edge weights may return different sets of results to users.

4.6.3 Efficiency of Relaxation

To demonstrate the efficiency of the AQR algorithm, we chose the SSO algorithm of FleXPath[10] for comparison.

Varying query size: We selected four queries q_1, q_2, q_3 and q_6 consisting of different number of nodes, i.e. 5, 8, 11 and 14, respectively, and then relaxed them against *auction.dtd*. q_4 and q_5 were not used because they have the same number of nodes (i.e., 11) as q_3. Figure 4.15 shows the number of relaxed queries and valuable

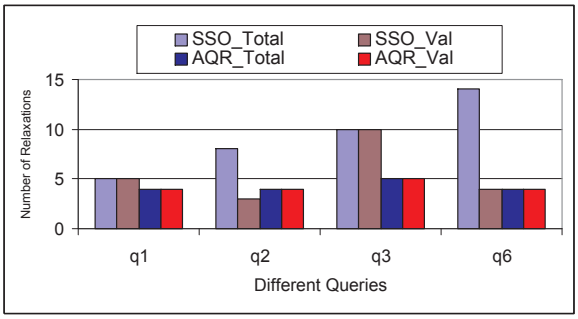


Figure 4.15: Relaxed Number vs. Different Queries

relaxed queries (which can return at least one answer). Most of the relaxed queries produced by AQR are valuable, while for SSO, only part of relaxed queries can return query results, such as q_2 and q_6, which do not totally conform to *auction.dtd*. The valuable relaxed query sets for AQR and SSO may not be identical. The set of queries generated by AQR is contained in that generated by SSO. This is because SSO also generates some queries that are too wild, such as "//item" and "//item/description", and the answers returned by these wild queries may not relevant and thus not significant to users. This shows that AQR can guarantee the quality of relaxed queries.

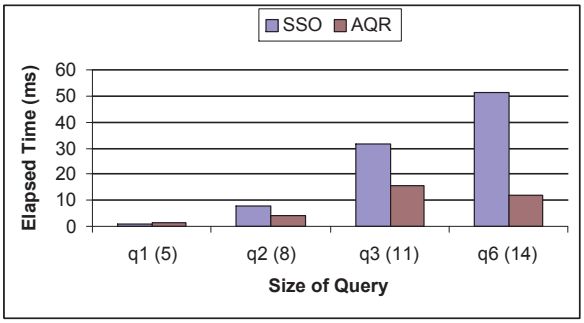


Figure 4.16: Elapsed Time vs. Query Size

From Figure 4.16, both algorithms can handle q_1 with the same time cost. This

is because q_1 matches the DTD well and the number of nodes is also small. For q_2, q_3 and q_6, the elapsed time goes up in both algorithms as the number of query nodes increases. AQR is superior in efficiency, because by utilizing DTD, AQR avoids generating a large number of unqualified queries. In addition, although the size of q_6 is larger than that of q_3, AQR needs less time to relax q_6 than q_3. The reason is that the recursive nodes *keyword* can be efficiently relaxed based on our recursive relaxation rule.

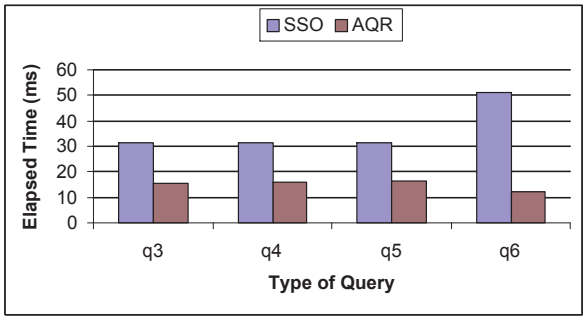


Figure 4.17: Elapsed Time vs. Query Type

Varying query types: To verify the effectiveness for processing different types of queries, we took q_3, q_4, q_5 because they have same number of nodes but different types, and q_6 with the recursive relationship and bigger number of nodes (i.e., 14). From the results in Figure 4.17, we find the elapsed time of AQR increases slowly in relaxing the former three queries, and falls down when processing q_6, this is because recursive relaxation reduces the cost of evaluating q_6. AQR guarantees that no noise nodes or edges exist in relaxed queries. As to SSO, it took almost same time to relax q_3 to q_5 due to the similar size of these queries. However, the elapsed time increases for processing q_6 because SSO has to spend more time for useless relaxation of noise nodes, and relaxation on recursion and inconsistent edges.

4.7 Summary

Different from previous work, the purpose of our adaptive query relaxation is to find the set of most relevant answers that best match users' intention specified in a query from large number of heterogeneous data sources. It would be very time-consuming and thus not acceptable to relax the query blindly and wildly. For this purpose, AQR chooses to relax users' queries based on source DTDs, which guarantees the relaxed queries are best suited for those sources that conform to the DTDs. During query relaxation, no unqualified or unnecessary relaxed queries will be produced. In addition, different from [10, 66], we compute the ranking score for each relaxed query by taking into account both the relaxing operations and the weights of edges, which allows to specify and adjust search requirements specific to users. Adaptive relaxation algorithms were implemented and illustrated through a comprehensive set of experiments to show the effectiveness and efficiency of AQR.

Chapter 5

Top-K Query Scheduling Strategies

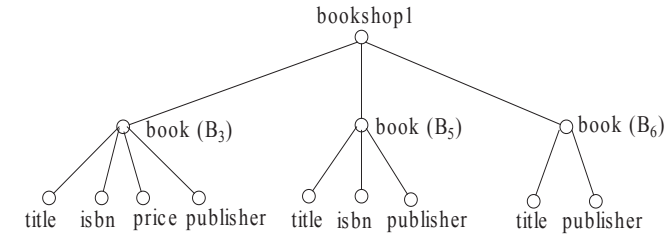
An important issue arising from XML query relaxation is how to efficiently search the top-k best answers from a large number of XML data sources, while minimizing the searching cost, i.e., finding the k matches with the highest computed scores by only traversing part of the documents. This chapter resolves this issue by proposing two methods - a brute-force strategy and a bound-threshold based scheduling strategy. Both strategies can be used to answer a top-k XML query as early as possible by dynamically scheduling the query over XML documents. The first method will generate all the relaxed queries in advance and then evaluate them one by one in a sequence until the k most relevant results are determined. In comparison, the second method will select the most relevant document to be evaluated according to the intermediate results at a time. Therefore, the total amount of documents that need to be visited can be greatly reduced by skipping those documents that will not produce the desired results with the bound-threshold strategy. Furthermore, most of the candidates in each visited document can also be pruned. Most importantly, the partial results can be output immediately during the query execution, rather than waiting for the end of all results to be determined. Our experimental results show that our query scheduling and processing strategies are both practical and efficient.

5.1 Introduction

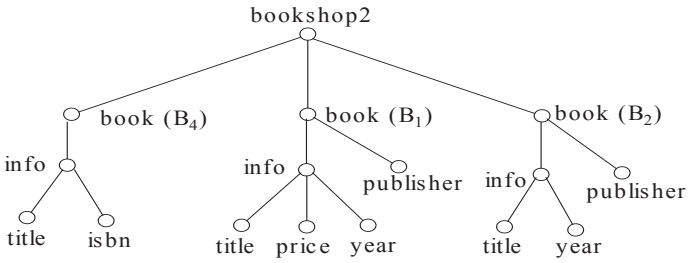
Over decades, processing top-k query has been extensively studied in different research areas, such as relational databases [21, 17, 26], multimedia databases [41, 44, 42, 70, 22, 20] and keyword search [86, 57]. Recently, Efficiently computing top-k answers to XML queries is gaining importance due to the increasing number of XML data sources and the heterogeneous nature of XML data. Top-k queries on approximate answers are appropriate on structurally heterogeneous data. On the one hand, it is difficult for users to formulate their queries exactly and search the exact answers. On the other hand, an XML query may have a large number of answers, and returning all answers to the user may not be desirable. The top-k approach can limit the cardinality of answers by returning k answers with the highest scores.

The problem of finding top-k answers within a large XML repository has been studied in [69], where an adaptive strategy is proposed for filtering out some unqualified candidates. However, its searching overhead is expensive due to frequent adaptivity among possible candidates and dynamic sort of partial matches. Furthermore, this work only considers query evaluation in a single XML document. For many applications, it is more meaningful to find top-k results from multiple heterogeneous XML data sources. In this chapter, we target this problem by proposing a brute-force strategy (in short BF strategy) and a bound-threshold based scheduling strategy (in short BT strategy) with the help of schema information of each XML data source. We are not required to evaluate a top-k query over all data sources. Instead, we schedule the query to the most relevant ones by leveraging the schema information, which can produce top k results as early as possible and output each result immediately after it is generated. Compared with BT strategy, BF need to generate all the relaxed queries and most of them are not qualified.

Example 1 *Consider two bookshop XML data sources in Figure 5.1 that maintain*



(a) Bookshop S_1

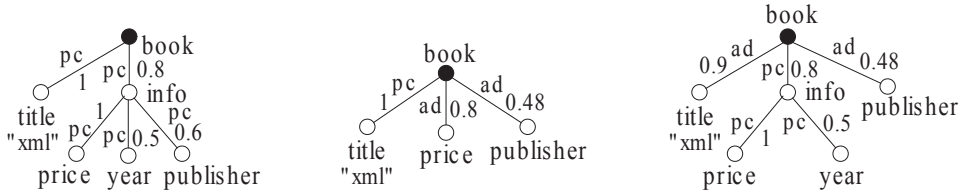


(b) Bookshop S_2

Figure 5.1: Bookshop Example

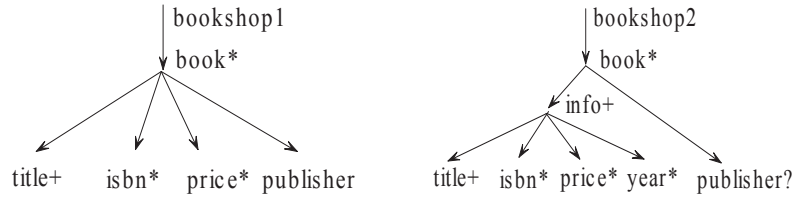
the partial or full information of each book: title, isbn, price, publisher and year. To search for two books (top-k=2) that contain "XML" in their titles and also include other specific information: expected price, published time and publisher, we can represent it as a tree pattern query q in Figure 5.2(a) where nodes are labeled by element tags, leaf nodes are labeled by tags and values, and edges are XPath axes (e.g. pc for parent-child, ad for ancestor-descendant). The root of the tree (shown in a solid circle) represents the distinguished node.

A naive solution to the above top-2 query is to retrieve the two most relevant books from each source and then select the more relevant ones by comparing their scores. However, this approach is not desirable for a large number of data sources due to amount of unnecessary processing cost. To solve this problem, we deploy XML schema information because a schema embodies to some extent the maximal structural information in the corresponding data sources that conform to the schema.



(a) User's query tree q (b) Relaxed query to the 1st schema q_1 (c) Relaxed query to the 2nd schema q_2

Figure 5.2: Query Tree and Relaxed Forms to different schemas



(a) Schema of the 1st Shop d_1 (b) Schema of the 2nd Shop d_2

Figure 5.3: Bookshop Schema Example

For example, in Example 1, the two bookshop sources conform to schema d_1 in Figure 5.3(a) and d_2 in Figure 5.3(b), respectively. Apparently, we can see that d_2 is more similar to the query q than d_1 in Figure 3. Consequently, we may expect to get more relevant results from S_2 than S_1 at the first instance. To achieve this, query relaxation can be used[79, 8]. The top-2 query q against d_1 and d_2 can be relaxed into q_1 and q_2, which are shown in Figure 5.2(b) and Figure 5.2(c), respectively. In other words, q_2 is more similar to q than q_1, therefore, we may schedule to evaluate q_2 on S_2 first. If we are able to get enough qualified books from S_2, we do not need to evaluate q_1 on S_1 at all. By a qualified book, we mean that the book contains more required information than any book in all other sources. In the example, two approximate results B_1 and B_2 in S_2 are qualified because both of them contain more information than any book in S_1 with regards to the original query q. As such, these two books may be returned as the results for the top-2 query.

However, not every approximate result returned from q_2 is qualified. For example, if we are evaluating a top-3 query, B_4 in S_2 may not be qualified because it contains less information than B_3 in S_1. This is because that a result (XML fragment) conforming to a schema may not necessarily contain all structural information of the schema. For a schema represented in DTD, we are allowed to specify disjunctive semantics (denoted as "|") and optional semantics (denoted as "*" or "?"). In other words, XML documents conforming to the same schema may vary in their structures. If we fail to find any or enough qualified results in one data source, we may try the next most relevant data source, say S_1 in the example to work out the results or rest of the results, say B_3 in the example for the top-3 query.

From the motivating example, it is not hard to find that for a large number of data sources, the processing time can be reduced significantly by adaptively scheduling user's query on the most relevant data source at the time and progressively evaluating it according to schema information. Bearing this in mind, we design our upper/lower bounds and threshold for the BT-based strategy.

We make the following contributions in this chapter:

- It proposes a BT-based scheduling strategy for efficiently searching top k results where we can skip most of the XML data sources according to schema information and also prune most candidates in each visited data source.
- It guarantees that results generated can be output immediately without waiting for the end of query evaluation.
- It provides stress tests and large-scale performance experiments that demonstrate the scalability and significant benefits of the proposed scheduling strategies.

The rest of this chapter is organized as follows. In Section 5.2, we first introduce preliminary knowledge for query relaxation and then formalize top-k problem over

a large number of XML data sources. The detailed discussions of the brute-force strategy and the BT-based scheduling strategy and their corresponding algorithms are given in Section 5.3 and Section 5.4, respectively. The experimental results are reported in Section 5.5. Finally, we conclude the study in Section 5.6.

5.2 Preliminary and Problem Statement

XML Query Relaxation: In this chapter, we represent XML queries as Tree Patterns [59] and allow users to add a weight on each edge to show their preferences on different path steps. We relax an XML query to different data sources based on the structural information provided in their corresponding DTDs while compute the changing weight of edges in query.

To guarantee that a relaxed query can be evaluated on a source, we need to collect information such as whether a node in the query appears in a DTD, the relationships between two query nodes, and the cardinality of a query node in a DTD, etc. Given a query q and a DTD d, the target of query relaxation is to find the relaxed query q' that is best suited to be evaluated on sources that conform to d by calling a set of relaxation operations: *deleting* a node, *generalizing* a pc-edge into a ad-edge and *promoting* a node [79, 8]. The minimal requirement for the relaxed query q' is that the root node r is kept in q'. For example, the query q in Figure 5.2(a) does not match exactly with the schema d_1 of the 1st shop in Figure 5.3(a). Therefore, in order to provide considerate and reliable service for users, relaxing the query against DTDs for the conformed documents is strongly in demand. According to the structural information in the schema d_1, we firstly delete the nodes *info* and *year*. And then the nodes *price* and *publisher* are promoted under the distinguished node *book* where they are connected with ancestor-descendant (ad) edges. The relaxed query is shown in Figure 5.2(b).

Problem Statement: Consider a weighted top-k query q and a large number of data sources $\{S_1, S_2, ..., S_n\}$ that conform to different DTDs $\{d_1, d_2, ..., d_n\}$ respectively. Let $\{q_1, q_2, ..., q_n\}$ be the set of weighted relaxed queries of q with regards to the set of DTDs. Our aim in this chapter is to efficiently search top k results by scheduling the evaluation of $\{q_1, q_2, ..., q_n\}$ over the set of data sources.

As stated in the problem statement, a set of relaxed queries $\{q_1, q_2, ..., q_n\}$ can be generated from the original query q based on the conforming DTDs $\{d_1, d_2, ..., d_n\}$, respectively. To start with, we require to rank the similarity between each q_i and q. During the query evaluation on a data source, we also check if a returned result is qualified or not. In this regard, we need a scoring function.

In a tree pattern query q, a user may specify, on an edge $e(v_1, v_2)$, how close v_1 and v_2 are associated with each other. To compute the weight of a query q, a naive function is to combine the weights of all edges in the query together where the edges are assumed to be independent from each other [8]. However, according to common understandings about XML queries, we find (1) the more path steps there are between two nodes, the less related the two nodes may be; (2) nodes lying on different paths are not related with each other, i.e., nodes are only related with their ancestors and descendants. Keeping these two features in mind, we introduce the concept of extended edge weight between a pair of nodes with *ad* relationship, $ad(v_i, v_j)$. The extended edge weight can be derived by multiplying weights along the path from v_i to v_j. Extended edge weights can be computed on the fly when required, or be calculated out beforehand, and then maintained dynamically.

Based on Definition 18 in Chapter 4, we can score the weight of a tree pattern query q by summing all extended edge weights of q, i.e., $score(q) = \sum_{\forall v_i, v_j \in V, ap(v_i, v_j)} w_e(v_i, v_j)$ where $ap(v_i, v_j)$ means either $ad(v_i, v_j)$ or $pc(v_i, v_j)$. Similarly, we can measure the similarity of a potential result rooted at any node v in source S with q by summing the weights of those extended edges that match q. We denote this as $score(v, q)$.

For q_1 and q_2 in Figure 5.2, we have $score(q_1) = w_e(book, title) + w_e(book, price) + w_e(book, publisher) = 1 + 0.8 + 0.48 = 2.28$ and $score(q_2) = w_e(book, title) + w_e(book, info) + w_e(book, publisher) + w_e(book, price) + w_e(book, year) + w_e(info, price) + w_e(info, year) = 0.9 + 0.8 + 0.48 + 0.8 \times 1 + 0.8 \times 0.5 + 1 + 0.5 = 4.88$. For the potential results, we have $score(B_1, q_2) = score(q_2)$ because B_1 covers all edges of q_2; $score(B_2, q_2) = score(q_2) - w_e(book, price) - w_e(info, price) = 3.08$; Similarly, we have $score(B_3, q_1) = score(q_1) = 2.28$ and $score(B_4, q_2) = 1.7$, and scores for B_5 and B_6 are less than that of B_4.

5.3 Brute-Force Scheduling Strategy

To find top k results for WTPQ q from data sources s_1, s_2, ..., s_n, we may take a brute-force strategy. The first step is to generate all relaxed queries for each data source. That is, for each q_i, we generate the set of relaxed queries $Q_i = \{q_{i1}, q_{i2}, \ldots, q_{im_i}\}$ based on d_i. In this step, the score of each q_{ij} will also be returned. Next, we can rank the generated relaxed queries in $Q_1, ..., Q_n$ for all data sources. Finally, we can evaluate the queries in order from the ranked list of queries until we get k results.

The generation of Q_i from q_i is based on the consideration of the optional semantics "*" and "?" and the disjunctive semantics "|" defined in the DTD d_i. For example, the node *project* in d_2 in Figure 4.1(c) is a *-node. This means that data source s_2 may have either many projects or zero project. We need to generate a relaxed query q_{22} from q_{21} to include the case of zero project. The relaxation of a query node with "?" defined in its correspondent node in DTD is similar to a *-node. If a query node a defined in its correspondent node in DTD has a disjunctive relationship "|" with another node b, i.e., "a|b", we generate two relaxed queries, one containing "a" while the other containing "b".

If q includes quite a few nodes with their correspondent nodes having disjunctive

and optional semantics in d_i, the size of Q_i may be increased. In this case, it is time-consuming to generate and sort all possible relaxed queries. To deal with this issue, we propose an effective scheduling strategy to adaptively evaluate the relaxed queries in the following section.

5.4 BT-based Scheduling Strategy for top-k Queries

In this section, the BT-based scheduling strategy for evaluating top-k queries will be discussed in detail. Specifically, in Section 5.4.1 we first provide the motivation of BT-based scheduling strategy. After that, we introduce data source and result determination properties that can be applied to schedule query evaluation over different data sources in Section 5.4.2. Then, in Section 5.4.3 static/dynamic strategies are proposed to evaluate the edges, which can reduce unnecessary computational cost. Finally, in Section 5.4.4 we design a set of algorithms for the BT-based scheduling.

5.4.1 Motivation

Assume we have the scores of the relaxed queries $\{q_1, q_2, ..., q_n\}$ as $score(q_1)$, $score(q_2)$, ..., $score(q_n)$ respectively. The BT-based strategy we propose is based on the concepts of upper/lower bounds [18] and threshold. We initialize the upper bound $U(i)$ and lower bound $L(i)$ of each source S_i as $score(q_i)$ and zero, respectively. To start our adaptive scheduling, we first choose the data source to be evaluated as S_{k1} if $U(k_1) = max\{U(i)|1 \leq i \leq n\}$ (i.e., the highest upper bound) and the threshold $\sigma = U(k_2) = max\{U(i)|1 \leq i \leq n \wedge i \neq k_1\}$ (i.e., the next highest upper bound). Then we start to evaluate q_{k1} on S_{k1} by probing an edge $e(v_1, v_2)$ of q_{k1} at a time. If $e(v_1, v_2)$ cannot be found in S_{k1}, $U(k_1)$ will be decreased; otherwise, $L(k_1)$ will be increased. The probing continues for next edge of q_{k1} until either $L(k_1) \geq \sigma$ or $U(k_1) < \sigma$. If $L(k_1) \geq \sigma$, all the candidates may become possible results depending on the value of

k required in top-k. If the number of candidates equals to k, all the candidates can be returned as qualified results and the process stops; if the number of candidates is less than k, all the candidates can also be returned as qualified results (with the adjustment of k) but the probing process continues on S_{k1}; otherwise, more probing is required to refine the qualified results. If $U(k_1) < \sigma$, we will continue the process. The next data source to be evaluated will be S_{k2} and the threshold will be chosen based on the updated list of the upper bounds. The process stops until k results are returned.

In our example, S_2 is chosen as the the data source to be evaluated first because $U(2) = score(q_2) > U(1) = score(q_1)$ at the beginning. If we have a top-2 query, B_1 and B_2 in S_2 will be returned as qualified results because both $score(B_1, q_2)$ and $score(B_2, q_2)$ are no less than the threshold $U(1) = score(q_1)$. If we have a top-3 query, we will first have B_1 and B_2 in S_2 as qualified results but the probing in S_2 continues until B_4 is met. At this time, $U(2)$ is decreased to $score(B_4, q_2) = 1.7$, which is less than the threshold $U(1) = score(q_1) = 2.28$. So the next source to be evaluated is switched to S_1, and its threshold is $score(B_4, q_2) = 1.7$. Since $score(B_3, q_1)$ (=2.28) is greater than the new threshold, B_3 becomes the third qualified result.

If the number of the data sources is large, we can avoid to evaluate most of the data sources based on the BT scheduling strategy. In addition, the qualified results can be returned immediately without waiting for all results to be determined.

5.4.2 Data Source and Result Determination Properties

From the motivation of the BT-based scheduling strategy in Section 5.4.1, we can get the following two properties.

Property 3 ***Data Source Determination and Switching:*** *At any time of query evaluation, we always evaluate the data source S_{k1} that has the highest upper bound*

$U(k_1) = max\{U(i)|1 \leq i \leq n\}$. When an edge $e(v_1, v_2)$ in q_{k1} is evaluated on the data source S_{k1}, if it turns out that $e(v_1, v_2)$ cannot be successfully evaluated on the fragments rooted from all of the distinguished nodes of S_{k1}, then the upper bound $U(k_1)$ will be decreased by $U(k_1) = U(k_1) - score(v_2, q_{k1}) - w_e(v_1, v_2)$. Suppose that the threshold $\sigma = U(k_2)$, then we have:

- *If the updated upper bound $U(k_1)$ is still larger than or equal to the threshold σ, then we need to continuously evaluate other edges in the query over the current data source S_{k1}.*
- *If the updated upper bound $U(k_1)$ becomes lower than the threshold σ, then the current data source S_{k1} needs to be suspended and query evaluation will be switched to the data source S_{k2}.*

Property 4 ***Result Determination:*** *When an edge $e(v_1, v_2)$ in q_{k1} is evaluated on the data source S_{k1}, if it turns out that $e(v_1, v_2)$ can be successfully evaluated on the fragments rooted from some of the distinguished nodes of S_{k1}, then the lower bound $L(k_1)$ will be increased by $L(k_1) = L(k_1) + w_e(v_1, v_2)$. Suppose that the threshold $\sigma = U(k_2)$ and the updated lower bound becomes larger than σ. Then we can affirm that some candidates generated so far in S_{k1} must be qualified as top-k results. We divide the set of candidates in S_{k1} into two groups G_1 that satisfies $e(v_1, v_2)$ and G_2 that does not, then the two groups will have different upper/lower bounds. Suppose that $\sigma \geq U(k_1)(G_2)$, then we have:*

- *If $|G_1| = k$, all the candidates in group G_1 can be returned as the qualified results and searching task would be terminated.*
- *If $|G_1| < k$, all the candidates in group G_1 can be returned as the qualified results and the k value will be decreased by $k = k - |G_1|$. Then the group G_2 should be evaluated if it is not suspended. If all the other groups in the data sources*

have been suspended, then we should switch to the next data source based on Property 3.

- *If $|G_1| > k$, we will evaluate other edges in the query q_{k1} on G_1 to find the top k results.*

Consider the top-2 query in our example again. We first evaluate q_2 on S_2 because $U(2)$ is larger than $U(1)$ (Property 3). Then we will choose some edges in q_2 to be evaluated, such as $(book, title)$, and $(book, info)$. All the edges can be found in the candidates of S_2. After that, the lower bound of the data source will increase to 1.7 (i.e., $L(2) = 0.9 + 0.8$). Then suppose we continue to evaluate $(info, year)$, at this point, we have two groups. The group G_1 of B_1 and B_2 satisfies $(info, year)$ while the group G_2 of B_4 does not. The lower bound of G_1 is increased to 2.6 ($L(2)(G_1) = 0.9 + 0.8 + 0.8 \times 0.5 + 0.5$) while the upper bound of G_2 is decreased ($U(2)(G_1) =$ 4.48 - 0.9 = 3.58). When $(info, price)$ is evaluated, the upper bound of G_2 is further dropped to 1.78 (3.58 - 0.8 $\times$ 1 - 1). To this point, the 2 candidates in G_1 can be output as qualified results because $L(2)(G_1) > \sigma$ and $U(2)(G_2) < \sigma$. The process stops here.

5.4.3 Edge Selection and Unqualified Edge Reduction

According to the above properties, the relationships among $U(k_1)$, $L(k_1)$ and σ need to be checked during query evaluation. Obviously, changing the value of the three variables will produce different query evaluation sequences over the large number of data sources. But the changing is likely to be influenced to some extent by the next edge that will be evaluated. Therefore, the selection of next edge can also affect the performance of query evaluation. In this section, we first introduce three ways to determinate the next edge. Then we discuss how to filter unqualified edges during query evaluation.

Intuitively, there are three processing strategies for determining next edge: *random* i.e., the next edge can be evaluated at random; *min_weight* i.e., the edge with the minimal weight can be evaluated first and *max_weight* i.e., the edge with the maximal weight can be evaluated first. For the first two strategies, the possibility that some data sources would be visited frequently is likely to be increased to some extent, which may lead to unnecessary costs. For the third one, at every time the edge with the maximal weight is selected to be evaluated, so that it has the higher possibility to increase the score of $L(k_1)$ if the edge can be found, otherwise, the score of $U(k_1)$ would be decreased at most. Both of the trends are likely to locate the data sources as early as possible that can return the answers. Therefore, the last one would yield a better performance.

Besides next edge selection, the determination of selection range is another important factor to improve the performance of query evaluation. A simple method is to consider all edges together at the beginning and rank them based on the weights of their corresponding subtrees, denoted as *static* style. Although it makes next edge selection very easy in real application, some edges that should be filtered out based on the intermediate feedbacks have to be still evaluated. Therefore, an optimized approach is proposed to incrementally expand the selection range, denoted as *dynamic* style. The reason that the dynamic approach can do better than the static one depends on the disjunctive and optional semantics in DTD. For example, if an edge (e.g. x/y) in a query is specified as optional in a DTD and does not exist in a data source conforming to the DTD, then all the edges coming from the element y are not required to be evaluated because they cannot exist in the current fragments. Therefore, if we expand the selection range in a dynamic style, some edges can be filtered beforehand based the intermediate results.

5.4.4 BT-based Scheduling Algorithms

Algorithm 11 BT-based Scheduling Strategy

input: a set of weighted relaxed queries $\{q_1, q_2, ..., q_n\}$ rooted at $\{r_1, r_2, ..., r_n\}$ and a set of data sources $\{S_1, S_2, ..., S_n\}$

output: top k results

1: call for the function *computingScore*() in Algorithm 14 to compute query weight as upper bound for each data source and denote the two highest upper bound as $U(k_1)$ and $\sigma = U(k_2)$ where $U(k_1) \geq U(k_2)$, $L(k_1) = 0$;
2: //{S_{k1} will be first evaluated}
3: put all candidates in S_{k1} into group G;
4: **if** $ch(r_{k1}) \neq \phi$ **then**
5: list l = sortAllChildNodes($ch(r_{k1})$);
6: ScheEval(l, q_{k1}, G, $U(k_1)(G)$, $L(k_1)(G)$, σ) in Algorithm 12;
7: **end if**

We use Algorithm 11 to initialize query evaluation over the data source S_{k1}. Algorithm 14 is used to compute the weight of each subtree in the query q_{k1} and $score(q_{k1})$ is taken as the initial value of the upper bound $U(k_1)$. Based on the BT scheduling strategy, we always evaluate the query q_{k1} on the data source S_{k1} with the highest upper bound $U(k_1)$ at any point. Then all the candidates in the data source S_{k1} can be clustered initially into one group G by using index or other technologies. After that, we will evaluate the edges in the query q_{k1} in a similar *breadth-first search* (BFS). To this end, three functions are deployed during query evaluation: *sortAllChildNodes*() sorts a list of nodes based on the weight of the subtrees rooted at these nodes where any traditional sorting algorithm can be applied (e.g., Insert Sort in [31]); *mergesort*() merges two sorted lists like Merge Sort in [31], which can improve the sorting efficiency because the previous list has been sorted before; *getFirstNode*() gets the first node from the sorted list l. At last, we will call for Function *ScheEval*() to probe a data source. Based on the evaluated results, we determine how to proceed at next step. The detailed procedure is described in Algorithm 12.

In Algorithm 12, we first get a node v with the function *getFirstNode*() and

Algorithm 12 ScheEval(a list l, query q, group G, $U(k_1)(G)$, $L(k_1)(G)$, σ)

```
1: while l ≠ φ do
2:   node v = getFirstNode(l) and delete the node v from the list l;
3:   evaluate the edge e(v'parent, v) in query q over the candidates G;
4:   if No candidates in G satisfy the edge e then
5:     U(k1)(G) = U(k1)(G) − score(v, q) − we(e);
6:     if U(k1)(G) < σ then
7:       suspend the current group G;
8:       if σ == U(k1)(Gx) then
9:         Switching to probe the group Gx in the current data source Sk1;
10:        ScheEval(l, q, Gx, U(k1)(Gx), L(k1)(Gx));
11:      else
12:        Switching to the next data source Sk2 due to σ = U(k2);
13:      end if
14:    end if
15:  else if All candidates in G satisfy the edge e then
16:    L(k1)(G) = L(k1)(G) + we(v'parent, v);
17:    if L(k1)(G) ≥ σ then
18:      determineCandidates();
19:    else
20:      list l' = sortAllChildNodes(ch(v)) and list l = mergeSort(l, l');
21:    end if
22:  else
23:    //{Partial candidates in G satisfy the edge e}
24:    divideGroup(e, q, G) into two groups G1 that satisfies the edge and G2 that
       doesn't and putActiveGroup(G2);
25:    U(k1)(G2) = U(k1)(G2) − score(v, q) − we(v'parent, v);
26:    L(k1)(G1) = L(k1)(G1) + we(v'parent, v);
27:    if U(k1)(G2) > σ then
28:      σ = U(k1)(G2);
29:      //{The group G2 in k1 data source would be evaluated at next step.}
30:    end if
31:    if L(k1)(G1) ≥ σ then
32:      determineCandidates();
33:    else
34:      list l' = sortAllChildNodes(ch(v)) and list l = mergeSort(l, l');
35:    end if
36:  end if
37: end while
```

evaluate the edge $e(v'parent, v)$ over the group of candidate nodes G. There are three possibilities. **(1)** Line 4 - 14: If no candidates in G satisfy the evaluated edge e, then the upper bound $U(k_1)(G)$ for the group will get a penalty $score(v)$, i.e., subtracting the score of the subtree rooted at v from the current upper bound. After that, we will compare the updated $U(k_1)(G)$ with the threshold σ. If $U(k_1)(G)$ is lower than σ, the current group will be suspended. And then previous groups or next data source will be evaluated depending on the conditions $\sigma = U(k_1)(G_x)$ or $\sigma = U(k_2)$, respectively. **(2)** Line 15 - 21: If all candidates in G satisfy the evaluated edge e, then the lower bound $L(k_1)(G)$ for the group will be increased by summing the extended weight $w_e(v'parent, v)$ of the edge. If $L(k_1)(G)$ is higher than σ, it means that the current group contains part or all results that can be determined by Function *determineCandidates*() in Algorithm 13. Otherwise, we open the child nodes of the node v to expand the current range of edges because the group of candidates can not be determined based on the current edge e so far. **(3)** Line 22 - 36: Most of the time, only part candidates in G satisfy the edge e, e.g., a subgroup G_1 of candidates satisfy while another subgroup G_2 of candidates do not. We use the function $divideGroup(e, q, G)$ to divide the group G of candidates into G_1 and G_2. Then we compute the upper bound, lower bound for each group. For G_1, its upper bound $U(k_1)(G_1)$ does not change, but its lower bound $L(k_1)(G_1)$ will increase. For G_2, its upper bound $U(k_1)(G_2)$ will decrease, however its lower bound $L(k_1)(G_2)$ keeps unchanged. Obviously, we have $U(k_1)(G_1) > U(k_1)(G_2)$. Therefore, we prefer searching in group G_1 to G_2 while cache group G_2 with Function *putActiveGroup*(). If $\sigma < U(k_1)(G_2)$, we should take $U(k_1)(G_2)$ as the new threshold for the current group G_1. And if $L(k_1)(G_1)$ is greater than or equal to the updated threshold σ, we will call for Function *determineCandidates*() in Algorithm 13. Otherwise, a new edge need to be evaluated on the current group of candidates.

Algorithm 13 can be designed to determine the correct ones from the group if we

Algorithm 13 Function: determineCandidates()

```
1: if |G| = k then
2:    return k results while Stop searching;
3: else if |G| < k then
4:    return λ results and k = k - |G|;
5:    if σ == U(k1)(Gx) then
6:       Switching to probe the group Gx in the current data source Sk1;
7:       ScheEval(l, q, Gx, U(k1)(Gx), L(k1)(Gx));
8:    else
9:       Switching to the next data source Sk2 due to σ = U(k2);
10:   end if
11: else
12:   list l' = sortAllChildNodes(ch(v)) and list l = mergeSort(l, l');
13:   ScheEval(l, q, G, U(k1)(G), L(k1)(G));
14: end if
```

find that a group of candidates in a data source would contain the correct answers for top-k query. There are three ways to process the candidates in G. (1) If $|G| = k$, the group of candidates are correct answers for top-k query and searching is terminated; (2) If $|G| < k$, the group of candidates are part of the correct answers and the value of k will be decreased by $k = k$ - $|G|$. At next step, we would probe the previous groups G_x in the current data source S_{k1} if we have $\sigma = U(k_1)(G_x)$ or switch to the next data source S_{k2} if we have $\sigma = U(k_2)$ (3) Otherwise, we will expand the edges and continuously evaluate them over the group G for determining the k best ones.

Algorithm 14 is used to mark the weight for each subtree in *deepth-first search* style. For each internal node v (i.e., $ch(v) \neq \phi$), we should push it into the stack S while update its score by computing the extended edge weight between the node v and its ancestor. For each leaf node or internal node that its child nodes have been processed, we will pop the node from the stack S while update its parent's score by propagating its score to its parent. Two important functions *getEdgeScore*() and *updateEdgeScore*() are used to retrieve and update the score of each node, respectively.

Algorithm 14 ComputingScore()

input: a weighted query rooted at r
output: a query that every subtree is marked with scores

```
1: push(the root r, a stack S);
2: while the stack is not empty S ≠ φ do
3:    v = getStackTop(S);
4:    existEdgeScore = getEdgeScore(v);
5:    if ch(v) ≠ φ then
6:       for all v_c ∈ ch(v) do
7:          newEdgeScore = getEdgeScore(v_c);
8:          currentEdgeScore = existEdgeScore × newEdgeScore;
9:          updateEdgeScore(v_c, currentEdgeScore);
10:         push v_c into the stack S;
11:         ComputingScore(v_c);
12:      end for
13:   else
14:      pop(a node, a stack S);
15:      v_x = getStackTop(S);
16:      xEdgeScore = getEdgeScore(v_x);
17:      updateEdgeScore(v_x, xEdgeScore + existEdgeScore);
18:   end if
19: end while
```

5.5 Experimental Results

The presented algorithms for the BT strategy are implemented in a Java prototype using JDK 1.4. B+-tree indexes are used to access the nodes in each data source. Wutka DTDparser [1] is used to analyze the source DTDs and extract their structural information. We run our experiments on an Intel P4 3GHz PC with 512M memory.

Table 5.1: Designed Queries

q_1:	//item [./description /parlist]
q_2:	//item [./description /parlist /mailbox /mail [./text]]
q_3:	//item [./mailbox /mail /text [./keyword and ./xxx] and ./name and ./xxx]

[1] Wutka DTD parser. http://www.wutka.com/dtdparser.html.

Dataset and Queries: We use XMark XML data generator [2] to generate a number of data sets, of varying sizes and other data characteristics, such as the fanout (MaxRepeats) and the maximum depth, using the *auction.dtd* and changed versions by deleting some nodes. We also use the XMach-1 [3] and XMark [4] benchmarks, and some real XML data. The results obtained are very similar in all cases, and in the interest of space we present results only for the largest auction data set that we generated. We evaluate the presented algorithms using the set of queries shown in Table 5.1 where the symbol "xxx" is added as noise node that do not appear in the DTD. In our query set, we consider the structural difference between the query and the DTD, such as the edge "parlist/mailbox" does not exist in the DTD. It will be adjusted by calling for previous query relaxation. We also take into account two semantics in DTD, such as the edge "description/parlist" satisfies disjunctive semantics and the nodes "mail" and "text" satisfy optional semantics.

Test Results: In our experiments, we implement the brute-force (BF) strategy and the BT-based scheduling strategy together. Our test results show that the BT scheduling strategy is faster than the BF strategy to search top-k matches over multiple data sources. Especially, when the number of data sources or the value of top-k are large, more benefits would be gained.

Figure 5.4 shows that dynamic sort-based BT scheduling strategy can improve the performance more than static sort-based BT scheduling strategy, in terms of evaluation of unqualified edges for some documents where the three queries are evaluated over 5, 10, 15 and 20 number of XML documents respectively and top-k is set as 80. In the following paragraph, we mainly choose the experiments about dynamic sort-based BT scheduling strategy in different conditions. Figure 5.5 shows BT scheduling strategy outperforms BF strategy greatly. Two appealing features can be obtained:

[2]Xmark XML data generator. http://monetdb.cwi.nl/xml/index.html.
[3]XMach-1. http://dbs.unileipzig. de/en/projekte/XML/XmlBenchmarking.html.
[4]The XML benchmark project. http://www.xml-benchmark.org.

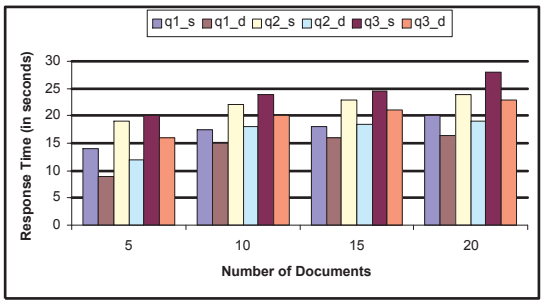


Figure 5.4: Static Sort vs. Dynamic Sort

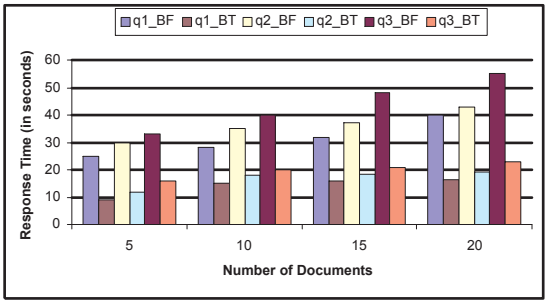


Figure 5.5: BF Schedule vs. BT Schedule

one the one hand, the larger the number of XML documents to be searched, the more benefits the BT scheduling strategy can gain; on the other hand, the BT scheduling strategy has good scalability, i.e., the increasing trends will become slow after the number of XML documents is relatively large. For example, the trends evaluating the three queries over 10, 15, and 20 documents are much slower than the trend between 5 and 10 documents. This is because a larger number of documents have higher possibility to contain noise documents.

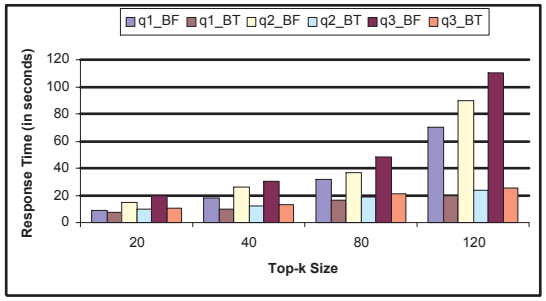


Figure 5.6: Varying Top-k Size

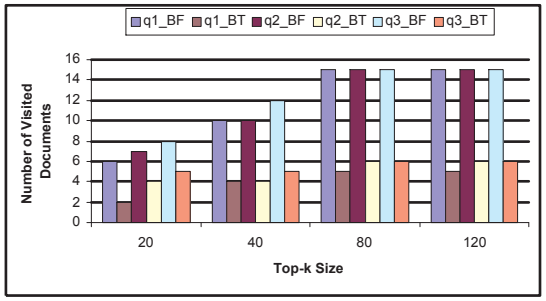


Figure 5.7: Varying Top-k Size

Figure 5.6 and Figure 5.7 illustrate the performance when we vary the size of top-k value across 20, 40, 80 and 120 where all the three queries are evaluated over 15 documents. From Figure 5.6, the both strategies can gain similar time cost when the top-k value is small. But the gap between the BT scheduling strategy and the

BF strategy will become much larger when top-k is 120. In addition, Figure 5.7 shows the number of documents that need to be visited in order to answer the three queries. Obviously, for the BF strategy, most of the documents require to be checked. However, for the BT scheduling strategy, only part of the documents are visited during query evaluation. Furthermore, the same number of documents are traversed for q_2 and q_3 when top-k is 80 or 120. This is because some elements in query like *mailbox* are distributed in the same documents when we design our data sets.

5.6 Summary

The primary contribution of this chapter lies in the two proposed methods - the brute-force strategy and the BT-based scheduling strategy. Especially, based on the BT strategy, we are able to avoid the evaluation of big number of data sources, and prune unqualified results in the visited data sources. Besides, the strategy also satisfies monotonic feature for returning qualified results. The experimental results demonstrated the BT scheduling strategy can gain more benefits when the value of top-k and the number of data sources are large. Additionally, the results also shown that the BT scheduling strategy can skip most of data sources during query evaluation. Therefore, it is appropriate and practical for the BT scheduling strategy to be applied to XML searching system.

Chapter 6

Effective Processing of XML Keyword Search

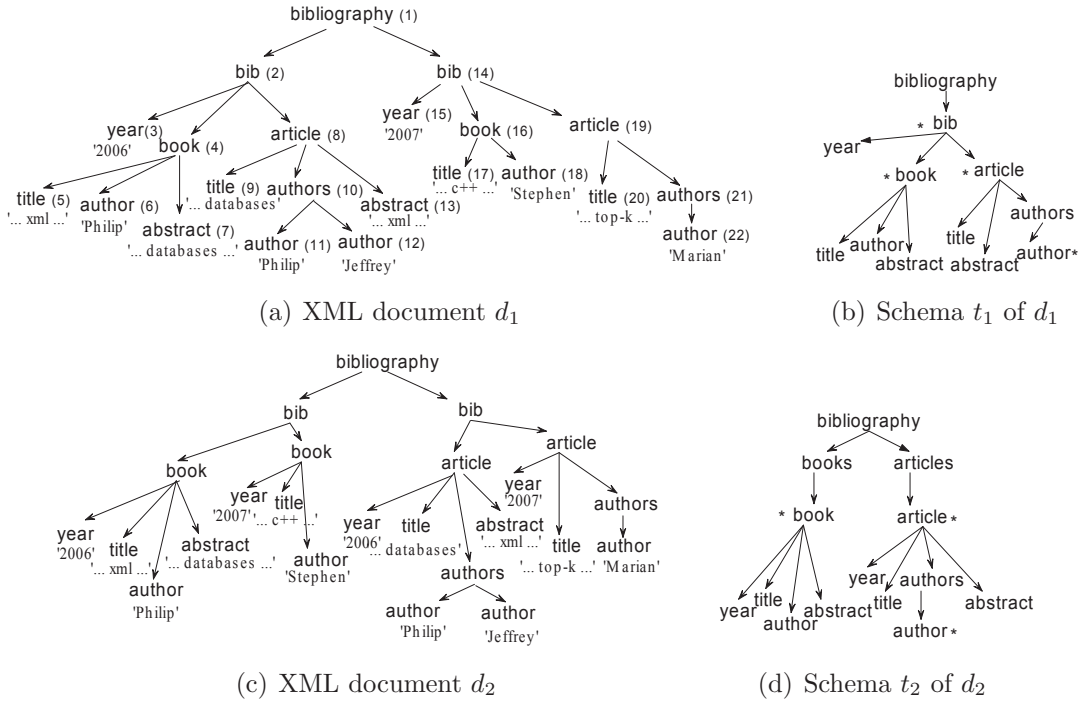
Recently, keyword search has attracted a great deal of attention in XML database. Most of previous work address this problem by selecting keyword-matched data nodes and merging them in a meaningful way. However, it is hard to directly improve the relevance and performance of keyword search because lots of keyword-matched nodes may not contribute to the results. To address this challenge, in this chapter we will design an adaptive XML keyword search approach, called *XBridge*, that can derive the semantics of a keyword query consists of different label-term pairs and generate a set of effective structured queries by analyzing the given keyword query and the schemas of XML data sources. To efficiently answer keyword query, we only need to evaluate the generated structured queries over the XML data sources with any existing XQuery search engine. In addition, we will extend our approach to process top-k keyword search based on the execution plan to be proposed. The quality of the returned answers can be measured using the context of the keyword-matched nodes and the contents of the nodes together. The effectiveness and efficiency of *XBridge* is demonstrated with an experimental performance study on real XML data.

6.1 Introduction

Keyword search is a proven user-friendly way of querying XML data in the World Wide Web [81, 85, 52, 30, 65]. It allows users to find the information they are interested in without learning a complex query language or knowing the structure of the underlying data. However, the number of results for a keyword query may become very large due to the lack of clear semantic relationships among keywords. There are two main shortcomings: (1) it may become impossible for users to manually choose the interesting information from the retrieved results, and (2) computing the huge number of results with less meaning may lead to time-consuming and inefficient query evaluation. As we know, users are able to issue a structured query, such as XPath and XQuery, if they already know a lot about the query languages and the structure of the XML data to be retrieved. The desired results can be effectively and efficiently retrieved because the structured query can convey complex and precise semantic meanings. Recently, the study of query relaxation [8, 62] can also support structured queries when users cannot specify their queries precisely. Nevertheless, there are many situations where structured queries may not be applicable, such as a user may not know the data schema, or the schema is very complex so that a query cannot be easily formulated, or a user prefers to search relevant information from different XML documents via one query.

Consider the example in Figure 6.1 showing the same bibliography data arranged in two different formats: XML document d_1 conforming to t_1 organizes publications based on the year of publication while XML document d_2 conforming to t_2 organizes publications according to their type (*book* or *article*). Both of them are modeled as the conventional labeled trees. Suppose a user wants to see the publications that is written by *Philip* as an author in *2006*, and the term *xml* appears in the title.

In this case, a user can quickly issue a keyword query "*Philip*, *2006*, *xml*" to obtain



(a) XML document d_1 (b) Schema t_1 of d_1

(c) XML document d_2 (d) Schema t_2 of d_2

Figure 6.1: XML Documents with Different Schemas

a list of answers. Node #4 *book* and node #8 *article* will be returned as relevant answers. From Figure 6.1(a), we can see that only node #4 *book* satisfies the searching requirement. For node #8 *article*, only the abstract contains the term xml, as such it does not meet the users' original intention. It would be impossible for an IR-style keyword query to differentiate the semantics, e.g. one term can represent different meanings in different positions. In addition, when the size of XML documents becomes larger, it is difficult to choose the meaningful answers from the large number of returned results. As an alternative, users can construct an XQuery to represent this simple query and specify the precise context. But there are two challenges: first, they have to know that "publication" in the schema is actually presented as *book* and *article* in both schemas; second, they have to know that *title* and *author* are the child elements of "publication", while *year* could be either a child or a sibling. Writing an accurate XQuery is non-trivial even for this simple example due to the complex

structure of XML schemas. Therefore, it is highly desirable to design a new keyword search system that not only permits users specify more expressive queries, but also implement keyword search as efficiently as structured queries.

To address this problem, a formalized keyword query consisting of a set of label-term pairs is deployed in [30] and [92]. In [30], labels in the given keyword query are used to filter the node lists. In [92], labels are used to construct answer templates that includes all combinations together according to the schema of XML data stream. When the data stream is coming, all matched nodes will need to be maintained until the template-matched results are generated or the end of the stream is reached. Different from them, in this work we develop a keyword search system called *XBridge* that first infers the context of the set of labels and the required information to be returned according to XML data schema. And then it may generate a set of precise structured queries and evaluate them by using existing XML search engines. To evaluate the quality of the results, in *XBridge* we propose a scoring function that takes into account the structure and the content of the results together. In addition, we also design an execution plan to retrieve the more qualified results as soon as possible, which is suitable to process top-k keyword search.

```
For $b in bibliography/bib
For $b2 in $b/book
Where $b/year = '2006'
        and contains($b2/title, 'xml')
        and contains($b2/author, 'Philip')
Return $b
```

```
For $b in bibliography/bib
For $a in $b/article
Where $b/year = '2006'
        and contains($a/title, 'xml')
        and contains($a/authors/author, 'Philip')
Return $b
```

Figure 6.2: Structured Queries w.r.t. XML Schema t_1

Consider the same example again, the user may change to issue "*author*:*Philip*, *year*:*2006*, *title*:*xml*" as a keyword query to search the relevant publications. For this keyword query, *XBridge* is able to automatically construct different structured queries for XML documents conforming to different XML schemas. For example, for

```
For $b in bibliography/books/book
Where $b/year = '2006'
        and contains($b/title, 'xml')
        and contains($b/author, 'Philip')
Return $b
```

```
For $a in bibliography/articles/article
Where $a/year = '2006'
        and contains($a/title, 'xml')
        and contains($a/authors/author, 'Philip')
Return $a
```

Figure 6.3: Structured Queries w.r.t. XML Schema t_2

the source schemas of the two XML documents shown in Figure 6.1, we can construct two sets of structured queries as shown in Figure 6.2 and Figure 6.3, respectively. After that, we can evaluate the structured queries to answer the original keyword query. The book node #4 will be returned as answers. We do not need to identify whether or not the title node #5 and the author node #11 belong to the same publications. As such, the processing performance would be improved greatly due to the specific context in structured queries.

This work makes the following contributions:

- For different data sources, *XBridge* can infer different semantic contexts from a given keyword query with label-term, which can be used to construct adaptive structured queries.
- A scoring function is proposed to evaluate the quality of the answers by considering the context of the keyword-matched nodes and the contents of the nodes in the answers.
- An execution plan, adapting to the proposed scoring function, is designed to efficiently process top-k keyword search.
- Experiments show that *XBridge* can obtain improved performance over previous keyword search approaches.

The rest of this chapter is organized as follows: In Section 6.2, we discuss and define the syntax of a keyword query. Section 6.3 provides the definition of XML

schema and presents how to identify the context of terms and derive the returned nodes by considering keyword queries and XML schemas together. Section 6.4 proposes a scoring function to evaluate the quality of returned answers by considering contexts and contents of the answers w.r.t. keyword query. Section 6.5 shows the structure of *XBridge* and describes the algorithms for constructing and evaluating the generated structured queries. The experimental results are reported in Section 6.6. Finally, we conclude the study of this work in Section 6.7

6.2 Query Syntax

The query language of a standard search engine is simply a list of terms. In some search engines, each term can optionally be prepended by a plus sign ("+"). Terms with a plus sign must appear in satisfying document, whereas terms without a plus sign may or may not appear in a satisfying document (but the appearance of such term is desirable). This functionality has been deployed in XSEarch [30]. In addition to specifying terms, the query language in XSEarch [30] allows the user to specify labels and label-term combinations that must or may appear in a satisfying document. Formally, a search term has the form $l : k$, $l :$ or $: k$ where l is a label and k is a term. A search term may have a plus sign prepended, in which case it is a required term. Otherwise, it is an optional term. Besides the above two points, the query model in our work *XBridge* permits users to distinguish the semantics of predicates from the returned nodes. For example, given a pair $l :$, it is hard to know whether we should consider it as a predicate or a required node to be returned. To make the query expression more clear, we extend this form to $l : *$ and $l :?$. The former means that one node should exist in the returned answers where the node's tagname is same to the label l and the node's values may be anything. The latter shows that the information of the nodes will be returned as answers if the nodes' tagname is same

to the label l.

When the kind of pair : k exists in the given keyword query, the number of plausible structured queries would be increased greatly. This is because the term k may appear anywhere in data source. The possibility will become much higher to combine the labels matching the term k with other labels in the given keyword query. We cannot directly construct structured queries for this case. To address this issue, we can apply for the technique in [19] to determine the best matched label with the term k and the other keyword pairs. Here, we assume all the labels are provided in the input keyword query.

The formal definition is as follows.

Definition 20 *(Keyword Query) a keyword query is defined as a set of label-term pairs* $q = \{l_i : k_i | 1 \leq i \leq n\}$ *where* l_i *is a label and* k_i *consists of three optional symbols: t, * or ?.* $l_i : k_i$ *means that the value containing the term* k_i *is bound to the label* l_i *as the tagname,* $l_i : *$ *means that a value with* l_i *as the tagname must appear, and* l_i :? *means that the content with* l_i *as the tagname is expected in the result.*

In summary, users can issue a keyword query q by following the three basic forms $l : k$, $l : *$ or l :?. If all pairs in query q conform to $l : k$ or $l : *$, we will derive the type of returned nodes as the approaches in [67]. The detailed procedure will be introduced in the following sections.

6.3 Identifying Context and Returned Nodes of Keyword Query

In this section, we show how to identify the context and the types of return nodes for a keyword query w.r.t. XML schema. XML schema is the foundation to construct structured queries from a keyword query. In case the schema is not available, we can

infer the schema based on data summarization, such as [94, 14]. In this chapter, we use XML schema tree to represent the structural summary of XML documents.

Definition 21 *(XML Schema Tree) An XML Schema Tree is defined as* $T = (V, E, r, Card)$ *where* V *is a finite set of nodes, representing elements and attributes of the schema* T*;* E *is set of directed edges where each edge* $e(v_1,v_2)$ *represents the parent-child (containment) relationship between the two nodes* v_1, $v_2 \in V$*, denoted by* $P(v_2) = v_1$ *or* $v_2 \in Ch(v_1)$*;* r *is the root node of the tree* T*;* $Card$ *is a set of mappings that maps each* $v \in V$ *to* $\{1, *\}$ *where "1" means that* v *can occur once under its parent* $P(v)$ *in a document conforming to* T *while "*" means that* v *may appear many times.*

6.3.1 Identifying Context of Keywords

From the set of labels given in a keyword query defined in Definition 20, we can infer the contexts of the terms for a data source based on its conformed XML schema. As we know each node in an XML document, along with its entire subtree, typically represents a real-world entity. Similarly, given a list of labels l_1, ..., l_n and an input XML schema tree T, an entity of these labels can be represented with a subtree of T such that it contains at least one node labeled as l_1, ..., l_n. We define the root node of the subtree as a master entity.

Definition 22 *(Master Entity) Given a set of labels* $\{l_i | 1 \leq i \leq n\}$ *and an XML schema tree* T*, the master entity is defined as the root node of the subtree* T_{sub} *of* T *such that* T_{sub} *contains at least one schema node labeled as* l_1*, ...,* l_n*.*

Based on Definition 22, a master entity may contain one or more than one schema nodes taking a label as their tagnames. If one master entity node only contains one schema node for each label, we can directly generate FOR and WHERE clauses. For example, let q(*year*:*2006*, *title*:*xml*, *author*:*Philip*) be a keyword query over the XML

document d_2 in Figure 6.1(c). Based on the schema t_2 in Figure 6.1(d), we can obtain two master entities *book* and *article*. Since the master entity *book* only contains one node labeled as *year*, *title* and *author* respectively, we can directly construct "For $b in bibliography/bib/book" and "Where $b/year='*2006*' and contains($b/title, '*xml*') and contains($b/author, '*Philip*')". Similarly, we can process another master entity *article*. The constructed queries are shown in Figure 6.3.

If one master entity node contains more than one nodes taking the same label, to construct FOR and WHERE clauses precisely, we need to identify and cluster the nodes based on the semantic relevance of schema nodes within the master entity. To do this, we may deploy the ontology knowledge to precisely estimate the semantic relevance between schema nodes. However, the computation adding additional measurement may be expensive. Therefore, in this chapter we would like to infer the semantic relevance of two schema nodes by comparing their descendant attributes or subelements. For example, given any two schema nodes v_1 and $v_2 \in$ a master entity T_{sub}, we can infer that the two schema nodes v_1 and v_2 are semantic-relevant nodes if they hold: $semi(v_1, v_2) \geq \sigma$. Here σ is the similarity threshold. If σ is set to 0.8, then it means that v_1 and v_2 contain 80% similar attributes or subelements.

Consider the same query q and the document d_1 in Figure 6.1(a). Based on the schema t_1 in Figure 6.1(b), we know there exists one master entity *bib* that contains one node with the label *year* and two nodes with the same labels *title* and *author* respectively. In this case, we first cluster the five nodes as C: { *year*, {*title*, *author*}$_{book}$, {*title*, *author*}$_{article}$ }$_{bib}$, and then identify whether or not the subclusters in C are semantic-relevant schema nodes. For instance, although {}$_{book}$ and {}$_{article}$ have different labels, both of them contain the same nodes *title* and *author*, i.e., the two nodes *book* and *article* contain 100% similar attributes. Therefore, the cluster C is partitioned into two clusters: C_1: { *year*, {*title*, *author*}$_{book}$ }$_{bib}$ and C_2: { *year*, {*title*, *author*}$_{article}$ }$_{bib}$. When all subclusters cannot be partitioned again, we

can generate different sets of FOR and WHERE clauses. For C_1, we have "For \$b in bibliography/bib" for $\{\}_{bib}$ and "For \$b2 in \$b/book" for $\{\}_{book}$ together. And its WHERE clause can be represented as "Where \$b/year='2006' and contains(\$b2/title, 'xml') and contains(\$b2/author, 'Philip')" according to the labels in the subclusters of C_1. Similarly, we can process C_2 to generate its FOR and WHERE clauses. Figure 6.2 shows the expanded structured queries.

The contexts of the terms can be identified by computing all the possible master entities from the source schemas first, and then specifying the precise paths from each master entity to its labels by checking the semantic-relevant schema nodes. Once the contexts are obtained, we can generate the FOR and WHERE clauses of the structured queries for a keyword query. By specifying the detailed context in FOR clauses, we can limit the range of evaluating the structured queries over the XML data sources, which can improve the efficiency of processing the keyword queries.

6.3.2 Identifying Returned Nodes

Given a keyword query $q = \{l_i : k_i | 1 \leq i \leq n\}$ and an XML schema tree $T = (V, E, r, Card)$, we may retrieve a set of master entities $V_m \subseteq V$ for q w.r.t. T based on the above discussion. In this section, we will derive the returned nodes only by identifying the types of the master entities V_m. For any master entity $v_m \in V_m$, if Card(v_m) = "*", we can determine that the node v_m can be taken as return nodes in the corresponding RETURN clauses because the node represents the real entity at the conceptual level. However, if Card(v_m) = "1", the node v_m may not represent an entity. In this case, we probe its ancestor nodes until we find its nearest ancestor v_a, such that Card(v_a) ="*".

Consider another keyword query q(*title*:*xml*, *author*:*Philip*) over the XML document d_1 conforming to t_1. We are able to obtain two master entities *book* and *article*. Since Card(*book*) = "*" and Card(*article*) = "*", we can take them as the return

nodes in the corresponding RETURN clauses. However, if users issue a simple query q(title:xml) over d_1, the master entity of this query is the *title* that is only an attribute of the book or article nodes. In this case, we can trust that users would like to see the information of the whole entity (book or article), rather than one single attribute. Therefore, to generate meaningful RETURN clauses, we have to extend the title node to its parent book or article nodes as the return nodes because book or article nodes belong to *-node type.

If there are some label-term pairs in the form of $\{l_{j_k} :?|1 \leq k \leq m \leq n \wedge 1 \leq j_k \leq n\}$ in $q = \{l_i : k_i|1 \leq i \leq n\}$, instead of returning the master entity of q, we will compute the master entity of $\{l_{j_k}\}$ and use it to wrap all return values of l_{j_k} in the RETURN clause. This is because users prefer to see those nodes with the labels $\{l_{j_k}\}$ as the tagnames according to Definition 20.

6.4 Scoring Function

Given a keyword query q and an XML schema tree T, a set of structured queries Q may be constructed and evaluated over the data source conforming to T for answering q. The answer to the XML keyword query q may be a big number of relevant XML fragments. In contrast, the answer to the top-k keyword query is an ordered set of fragments, where the ordering reflects how closely each fragment matches the given keyword query. Therefore, only the top k results with the highest relevance w.r.t. q need to be returned to users. In this section, our scoring function consists of the context of the given terms and the weight of each term.

Let a fragment A be an answer of keyword query q. It is true that we can determine a structured query q_i that matches the fragment A. This is because we first construct the structured query q_i from the keyword query q and then obtain the fragment A by evaluating q_i over XML data. Therefore, we can compute the context score of the

fragment A by considering the structure of the query q_i.

Definition 23 *(Context Score) Assume the structured query q_i matching the answer A consists of the labels $\{l_i|1 \leq i \leq n\}$. Let its master entity be v_m and an XML schema be $T = (V, E, r, Card)$, we can obtain a list of nodes $V' \subseteq V$ that match each label respectively.*

$$ContextScore(A, q_i) = \frac{1}{n} \times \sum_{i=1}^{n} K_{context}(v_i, v_m) \tag{6.1}$$

where $K_{context}(v_i, v_m)$ is computed based on the distance from the node $v_i \in V'$ matching the label l_i to its master entity v_m, i.e., $LengthOfPath^{-1}(v_i, v_m)$.

In order to effectively capture the weight of each individual node, we are motivated by the $tf * idf$ weight model. Different from IR research, we extend the granularity of the model from document level to element level.

Definition 24 *(TF value)*

$$tf(v_i, t_i) = |\{v \in V_d | tag(v) = tag(v_i) \& t_i \in v\}| \tag{6.2}$$

Intuitively, the tf of a term t_i in a node v_i represents the number of distinct occurrences within the content of v_i.

Definition 25 *(IDF value)*

$$idf(v_i, t_i) = log(\frac{|\{v \in V_d : tag(v) = tag(v_i)\}|}{|\{v \in V_d : tag(v) = tag(v_i) \& t_i \in v\}|}) \tag{6.3}$$

Intuitively, the idf quantifies the extent to which the nodes v with the same tagname as v_i in XML document node set V_d contain the term t_i. The fewer v_i nodes whose contents include the term t_i, the larger the idf of a term t_i and a node v_i.

Without loss of generality, we will also assume that the weight value of each node is normalized to be real numbers between 0 and 1.

Definition 26 *(Weight of Individual Keyword) The answer A contains a set of leaf nodes as tagnames with the given labels* $\{l_i|1 \leq i \leq n\}$ *where each leaf node should contain the corresponding term at least once a time. For each node* v_i *with label* l_i*, we have:*

$$\omega(v_i, t_i) = \frac{tf(v_i, t_i) \times idf(v_i, t_i)}{max\{tf(v_i, t_i) \times idf(v_i, t_i)|1 \leq i \leq n\}} \tag{6.4}$$

Definition 27 *(Overall Score of Answer) Given a generated structured query* q_i *and its answer A, the overall score of the answer can be computed as:*

$$Score(A, q_i) = \frac{1}{n} \times \sum_{i=1}^{n} \frac{\omega(v_i, t_i)}{LengthOfPath(v_i, v_m)} \tag{6.5}$$

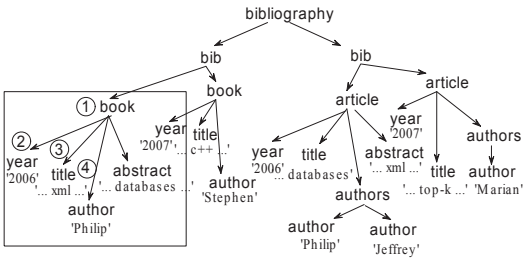


Figure 6.4: XML Document d_2

Now let me show how we compute the overall score of an answer. Assume q is a keyword query that consists of three pairs "*author*:*Philip*, *year*:*2006*, *title*:*xml*". We evaluate the keyword query q over the XML document in Figure 6.4. After that, the fragment *book* in the box will be returned as an answer. Based on the pre-computation, we can get the weight of each keyword-matched node in the fragment where we assume each term only occurs once a time in the corresponding nodes.

According to the pre-computed information in Table 6.1 and Table 6.2, we have $\omega(year, 2006) = 1 \times 0.301 = 0.301$, $\omega(title, xml) = 1 \times 0.602 = 0.602$ and $\omega(author, Philip)$

$tf(v_i, t_i)$	value
(year:2006)	1
(title:XML)	1
(author:Philip)	1

Table 6.1: tf values of *book* in the box

$idf(v_i, t_i)$	value
(year:2006)	$\log(\frac{4}{2})$= 0.301
(title:XML)	$\log(\frac{4}{1})$= 0.602
(author:Philip)	$\log(\frac{5}{2})$ = 0.398

Table 6.2: idf values of *book* in the box

$= 1 \times 0.398 = 0.398$. The maximal weight is 0.602. In addition, we also need to normalize each weight value into (0,1) by dividing by the maximal value. So the final normalize weight values are $\omega(year, 2006) = 0.5$, $\omega(title, xml) = 1$, and $\omega(author, Philip)$ = 0.661 respectively. Then we can compute the overall score by combining the weight of each node and its context. $Score(book, q_i) = \frac{1}{3} \times (1 \times 0.5 + 1 \times 1 + 1 \times 0.661)$ = 0.720. If we have more results, we can compute its overall score in the same way.

According to Equation 6.1, Equation 6.4, Equation 6.5, we find that the overall score of an answer is equal to its context score when the weight of each keyword is set to 1 (the maximal value). Therefore, the context score can be taken as the upper bound of the answer. Based on the relationships among the three equations, we can design an execution plan to determine whether the current results are more relevant to the user's query than the rest to be searched. The detailed procedure has been provided in the following section.

6.5 Implementation of XML Keyword Query

Figure 6.5 shows the structure of *XBridge* system. Given a keyword query, we first construct a set of structured queries based on the labels in the keyword queries

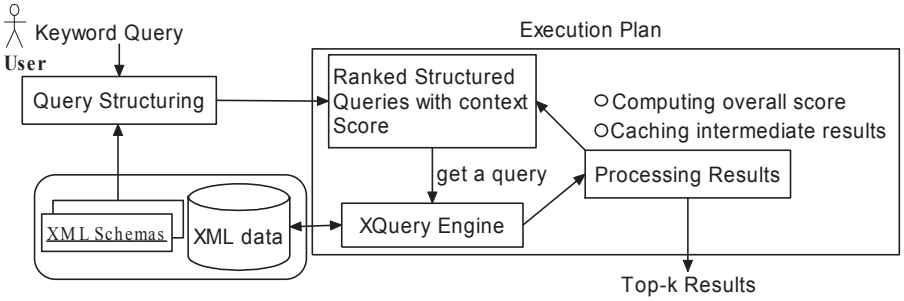


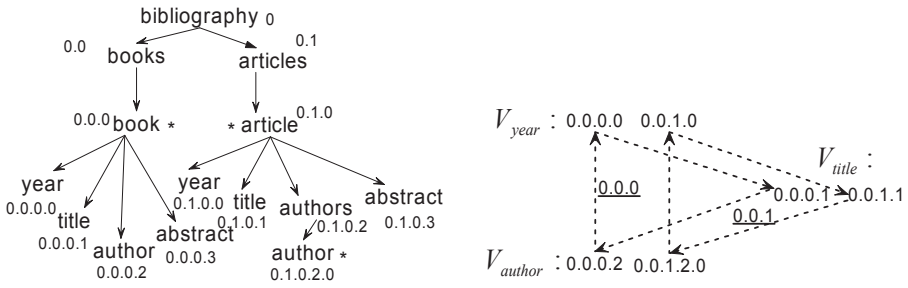
Figure 6.5: Architecture of *XBridge* System

and the XML schemas, i.e., query structuring. Then the set of queries Q will be sorted according to their context score based on Definition 23, i.e., structured query ranking. After that, we get a query $q_i \in Q$ with the highest context score from the set of structured queries and send it to XQuery engine. We will evaluate the query q_i over XML data and retrieve all the results matched q_i. At last, we will process all the current results, i.e., computing the overall score for each result and caching the k results with the higher overall scores. After we complete the evaluation of the query q_i, we need start a new loop to process another structured query that comes from the current set $\{Q - q_i\}$ until the top k qualified results have been found. In the following sections, we will discuss the detailed procedures of query structuring and execution plan.

6.5.1 Query Structuring

Query structuring is the core part of *XBridge* where we compute the master entities, infer the contexts and derive the return nodes. To search the master entities quickly, Dewey number is used to encode the XML schema. For example, Figure 6.6(a) shows the schema for XML document d_2 in Figure 6.1(c) and gives the Dewey number for each schema node. Given two nodes *year* (0.0.0.0) and *title* (0.0.0.1), we can say the node *book* (0.0.0) is their nearest common ancestor (NCA) node because the Dewey

numbers of *year* and *title* share the same prefix 0.0.0 and no other common prefixes exist in the subtree rooted at the node *book* (0.0.0). Therefore, we use Dewey number to compute the master entities for a given keyword query.



(a) Example of Dewey number (b) Candidates matching to labels

Figure 6.6: Example of Implementation

Given a keyword query $q = \{l_i : k_i | 1 \leq i \leq n\}$ and an XML schema tree $T = (V, E, r, Card)$, we first retrieve a list of nodes $V_i \subseteq V$ for each l_i in T and sort them in a descending order, such that we have $0.0.0.0 \prec 0.1.0.0$ in V_{year}. To compute master entities, we propose two approaches:

- **Pipeline:** We take each node v_i from the list V_i where $|V_i| = \min\{|V_1|, |V_2|, ..., |V_n|\}$ and compute the NCA of v_i and the nodes in the other lists V_j $(1 \leq j \leq n \wedge j \neq i)$ in a sequence. At last, we preserve the NCA nodes as the master entity candidates in V_m.

- **Pipeline+σ:** When we get any node v_{ix} from V_i, we may take its next node v_{ix+1} as a threshold σ if the node v_{ix+1} exists in V_i. During the computation of NCA, we only probe the subset of nodes V in the other lists such that for $v \in V$ we have $v \prec v_{ix+1}$.

As an optimization approach, the second one can reduce the negative computations of NCA while it can obtain the same structured queries as the first one does.

Algorithm 15 Constructing structured queries

input: a query $q = \{l_i : k_i | 1 \leq i \leq n\}$ and an schema $T = (V, E, r, Card)$
output: a set of structured queries Q

1: Retrieve a list of nodes $V_i \subseteq V$ for each l_i in T;
2: Compute the master entities V_m by calling for **Pipeline+σ** approach;
3: **for** each master entity $v_m \in V_m$ **do**
4: Generate **FOR clause** with v_m, i.e. "For \$$x$ in $r/.../v_m$";
5: Cluster the nodes matching with labels l_i in the subtree of v_m w.r.t. domain knowledge D by calling for Cluster_Domain($\{l_i\}$, v_m, D);
6: **for** each cluster **do**
7: Construct a set of **FOR clauses** for the entity nodes representing the similar semantic in each cluster;
8: Generate **WHERE clause** with n paths from v_m to each node l_i;
9: Generate **RETURN clause** by identifying the types of v_m (Card(v_m)=* or 1) and k_i ("?" symbol exists or not) and put the structured query into Q;
10: **end for**
11: **end for**
12: **return** the set of structured queries Q;

Since the subtree of a master entity v_m may cover one or more than one schema nodes labeled with the same label and the nodes may have different semantic relevances, we develop a function Cluster_Domain() to identify and cluster the nodes matching with the labels l_i $(1 \leq i \leq n)$ in the subtree according to the labels and the data types of their attributes or subelements. For each cluster c, we do not generate a structured query if the cluster c only contains a part of the labels in the given keyword query. Otherwise, we construct a structured query for the cluster c. In this case, we first generate a set of FOR clauses according to the classified clusters in the cluster c and then construct a WHERE clause for c. Finally, a RETURN clause is derived by identifying the type of the master entity node v_m. After all the clusters are processed, we may generate a set of structured queries Q. The detailed procedure has been shown in Algorithm 15.

Let us look at the procedure of Algorithm 15 with an example. Users issue a keyword query q(year:2006, title:xml, author:Philip) over XML document d_2 that

conforms to the schema in Figure 6.6(a). At the beginning, we retrieve the relevant nodes for the given labels, such as V_{year}, V_{title} and V_{author} shown in Figure 6.6(b). Based on Line 2, we can obtain the master entities, i.e., V_m={0.0.0, 0.1.0}. For the *book* node 0.0.0, we generate a FOR clause: "For $b in bibliography/bib/book". Based on Line 5-8, we know the three nodes *year* 0.0.0.0, *title* 0.0.0.1 and *author* 0.0.0.2 belong to one entity *book* 0.0.0. Therefore, we can directly produce a WHERE clause: "Where $b/year='2006' and contains($b/title, 'xml') and contains($b/author, 'Philip')". After that, we generate a RETURN clause: "Return $b" because the *book* node is *-node. Similarly, we can generate another structured query for master entity *article* 0.1.0. The constructed queries have been shown in Figure 6.3.

6.5.2 Execution Plan for Processing Top-k Query

Given a keyword query q and XML documents D conforming to XML schemas, we may generate a set of structured queries Q. To obtain top-k results, a simple method is to evaluate all the structured queries in Q and compute the overall score for each answer. And then we select and return the top k answers with the k highest scores to users. However, the execution is expensive when the number of structured queries or retrieved results is large.

To improve the performance, we design an efficient and dynamic execution plan w.r.t. our proposed scoring function, which can stop query evaluation as early as possible by detecting the intermediate results. Our basic idea is to first sort the generated structured queries according to their context scores, and then evaluate the query with the highest score where we take the context score of the next query as the current threshold. This is because the weight of each keyword is assumed to be set as 1 (the maximal value). In this case, the overall score would be equal to the context score. Therefore, if there are k or more than k results, we will compute their overall scores based on Equation 6.5 and cache k results with the k highest scores.

Algorithm 16 Dynamic Execution Plan

input: A set of ranked structured queries Q and XML data D
output: Top k qualified answers

```
1: Initialize the answer set SA=null;
2: Initialize a boolean symbol flag = false;
3: while Q ≠ null and flag ≠ true do
4:    Get a structured query q = getAQuery(Q) where q has the maximal context
      score;
5:    if |SA| = k and (minScore{A ∈ SA} ≥ ContextScore(q)) then
6:       flag = true;
7:    else
8:       Issue q to any XQuery search engine and search the matched fragments F_m;
9:       for ∀A ∈ F_m do
10:         Compute overall score Score(A, q);
11:         if |SA| < k then
12:            Input the result A into SA;
13:         else if ∃A' ∈ SA and Score(A', q') < Score(A, q) then
14:            Update the intermediate results updateSA(A, SA);
15:         end if
16:      end for
17:   end if
18: end while
19: return Top k qualified answers SA;
```

At the same time, we will compare the scores of the k results and the threshold. If all the scores are larger than the threshold, we will stop query evaluation and return the current k results, which means no more relevant results exist.

Algorithm 16 shows the detailed procedure of our execution plan. At the beginning, the function getAQuery() is called to get the structured query q with the maximal context score from the query set Q. Then we evaluate the query q with any existing XQuery search engine and retrieve all the matched results F_m. If the answer set SA is empty or contains less than k results, we can directly get the k or partial qualified results from F_m and cache them into temporary answer set SA. If the answer set SA has contained k temporary results, we need to compare the existed results in

SA and the new retrieved ones in F_m. If we find that there are more qualified results in F_m, we will use them to replace the less qualified results in SA, i.e., updating the temporary answer set updateSA(). Before we start another new iteration, i.e., we get a new structured query q' from Q, we need to compare the context score of the query q' with the overall scores of the temporary results in SA. If the overall score of any result is larger than or equal to the context score of q', then we can guarantee all the qualified answers have been found. Therefore, it is not necessary to process the query q' and the rest of the queries in Q. Otherwise, we need to evaluate the query q' continuously.

6.6 Experiments

We implemented *XBridge* in Java using the Apache Xerces XML parser and Berkeley DB[1]. To illustrate the effectiveness and efficiency of *XBridge*, we also implemented the Stack-based algorithm[49] because the other related approaches in [90, 67, 83] are biased to the distribution of the terms in the data sources. All the experiments were carried out on a Pentium 4, with a CPU of 3GHz and 1GB of RAM, running the Windows XP operating system. We selected the Sigmod Record[2] XML document (500k) and generated three DBLP[3] XML documents as the dataset where we evaluated the following keyword queries in Table 6.3.

Table 6.3: Keyword Queries

q_1:	$(author : David, title : XML)$
q_2:	$(year : 2002, title : XML)$

q_1 means users want to search the publications that are written by David and their

[1]http://www.sleepycat.com/
[2]http://www.sigmod.org/sigmod/record/xml/index.html
[3]http://dblp.uni-trier.de/xml/

titles contain XML. q_2 means users want to know the publications that are published in year 2002 and their titles contain XML. Table 6.4 gives the sizes of datasets and the number of each keyword pair in different datasets.

XML Data	Data Size	# *author:David*	# *title:XML*	# *year:2002*
SigmodRecord.xml	0.49M	74	117	0
dblp_01.xml	23.26M	1953	522	4871
dblp_02.xml	51.20M	4586	1476	11949
dblp_03.xml	76.48M	7279	2196	17202

Table 6.4: The Number of Keyword-matched nodes in Different Documents

Keyword queries	SigmodRecord	dblp_01	dblp_02	dblp_03
q_1 (author:David, title:XML)	3	2	21	33
q_2 (year:2002, title:XML)	0	45	153	214

Table 6.5: The Number of Results Returned by XBridge

Keyword queries	SigmodRecord	dblp_01	dblp_02	dblp_03
q_1 (author:David, title:XML)	74	522	1476	2196
q_2 (year:2002, title:XML)	0	522	1476	2196

Table 6.6: The Number of Results Returned by Stack-based Algorithm

Table 6.5 and Table 6.6 show the number of results returned by XBridge and Stack algorithm, respectively. Comparing the two tables, we find that the results of XBridge is a small part of Stack algorithm. In addition, the small part of results is more meaningful than the rest in the results of Stack algorithm. For exmaple, in Sigmod Record, there are 74 articles that are written by David and 117 articles that their titles contained XML. But for the two keywords only 3 results are found by XBridge while 74 relevant results will be returned by Stack algorithm. This situation may become worse in DBLP XML documents. For example, dblp_01.xml contains 1953 author:David pairs and 522 title:XML pairs, but for q_1 only 2 results can be returned by XBridge while 522 relevant results are found by Stack algorithm. This

is because Stack algorithm does not consider the semantic relevance during query evaluation.

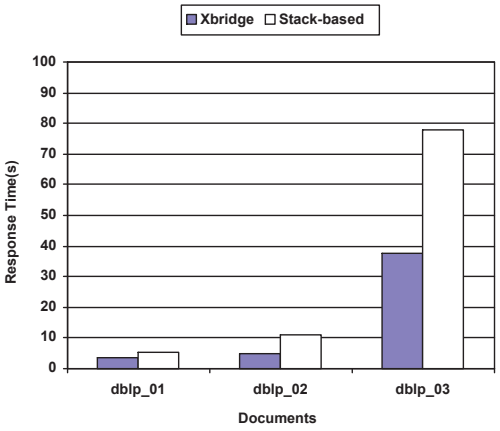


Figure 6.7: Keyword Query q_1

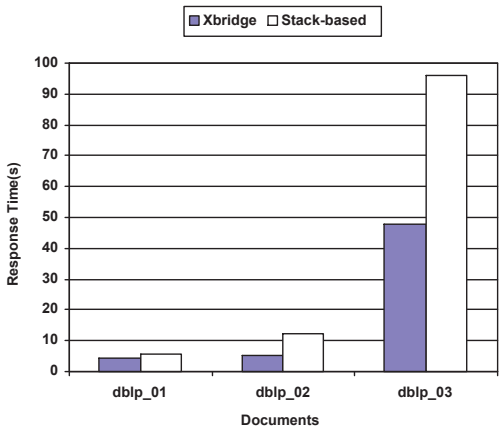


Figure 6.8: Keyword Query q_2

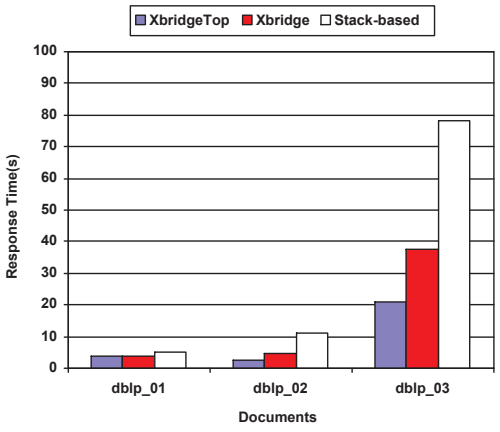


Figure 6.9: Keyword Query q_1 $(k = 10)$

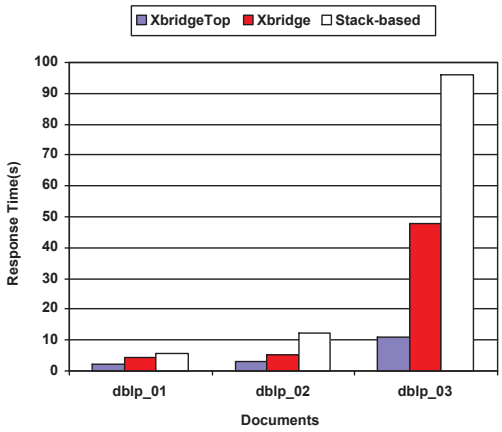


Figure 6.10: Keyword Query q_2 $(k = 20)$

As we know, the Stack-based algorithm may avoid some unnecessary computations by encoding the documents with the Dewey scheme. But it has to preserve all possible intermediate candidates during query evaluation and lots of them may not produce the results. However, *XBridge* can infer the precise contexts of the possible results based

on the source schemas before query evaluation. Therefore, *XBridge* can outperform the Stack-based algorithm. For example, the Stack-based algorithm spent 188ms to evaluate q_1 on SigmodRecord.xml while $XBridge$ only used 78ms to process the same keyword search. In addition, if the size of an XML document increases, most of the time it may contain more nodes that match a single keyword. But the number of return nodes that match all the keywords may not be increased significantly. In this case, the performance of query evaluation may be relatively decreased.

Figure 6.7 and Figure 6.8 illustrate the time cost of both methods when we evaluate q_1 and q_2 on the given three DBLP datasets respectively. From the experimental results, we find that in $XBridge$ the change in time is not obvious when the size of dataset is less than 50M. But when the size of document is nearly 70M, the processing speed was decreased by 80%. This is because compared with dblp_01 or dblp_02, dblp_03.xml contains a huge number of nodes that match each single keyword but fail to contribute to return nodes. The figures also shows that XBridge would outperform over Stack-based algorithm in our experiments.

Figure 6.9 and Figure 6.10 compare the performance of the methods when k value is set as 10 or 20 respectively. Since Stack-based algorithm has to retrieve all the results and then select the k qualified answers, its response time for top-k query is nearly same to process general query. However, $XBridge$ depends on the dynamic execution plan, which can stop query evaluation as early as possible and guarantee no more qualified results exist in the data source. Generally, $XBridge$ is suitable to process top-k keyword search and most of the time, it only needs to evaluate parts of the generated structured queries. But when the number of relevant results is less than k, $XBridge$ also needs to evaluate all the generated structured queries. For example, as we know the document dblp_01 only contains 2 qualified answers for the keyword query q_1. When the specified value of k is 10, there is no change of the response time. Therefore, in this special case, its efficiency cannot be improved.

6.7 Summary

The main contribution of this work is to propose the *XBridge* system for processing keyword search by constructing effective structured queries. It can improve the relevance and performance of keyword search greatly by specifying the precise contexts of the constructed structured queries. In addition, we also provide a scoring function that considers the context of the keywords and the weight of each keyword in the data source together. Especially, an execution plan for processing top-k keyword search is designed to adapt to our proposed scoring function.

Chapter 7

Conclusions and Future Work

We first give a report on the major conclusions of this thesis in Section 7.1, and then propose some directions for future work in Section 7.2.

7.1 Summary of this Thesis

The research objective described in this thesis is to investigate a series of problems related to adaptive query processing in XML.

We first studied the features of XML schema and proposed a structural similarity model to compare the *S*tructures of XML *S*ource schemas with an XML *D*omain schema (SSD). SSD provided an accurate similarity measure by taking into account two main factors that contribute to the structural similarity or difference: element coverage and consistency of relationships of element pairs. In addition, it also considered the difference of element cardinality in the model. The computed similarity can be used to decide which data source is more relevant to users' query requirements, i.e., the most relevant data source should be accessed as early as possible during query evaluation. To speed up similarity computation, we introduced a trimming process for filtering out uninteresting objects while preserving similarity. Meanwhile, we also

proposed a coding scheme to serve for comparison of a pair of schemas by extending Dietz's numbering schema that is often used to improve the query efficiency of individual XML documents. Two algorithms - the basic and the improved algorithms were developed with unnecessary comparisons removed in the improved algorithm. We conducted a thorough experimental evaluation of the two approaches using synthetic and public datasets. Our evaluation showed that the improved algorithm outperforms significantly compared with the basic algorithm for different features of the data.

Second, we presented an *A*daptive *Q*uery *R*elaxation (AQR) for evaluating XML query over heterogeneous XML data sources. AQR consists of a series of relaxation operations: Ontology Relaxation, Node Relaxation, Term Relaxation, Inconsistent Edge Relaxation, and Recursive Relationship Relaxation. It can avoid blind relaxation by filtering out those query nodes that do not appear in the DTD and adjusting the node relationships if they do not match the DTD. It can also avoid wild relaxation by preserving the query requirements which are definitely satisfied by the DTD. All of these advantages can improve the performance of query relaxation and evaluation. In AQR, we devised a penalty/ranking model that is extended from the model SSD. Compared with SSD, the penalty/ranking model is more accurate to detect the structural differences of the original query and its counterpart in data source, such as AQR considers the distance of two nodes that satisfy ancestor-descendant relationship while SSD does not. In addition, a set of adaptive relaxation algorithms are proposed to efficiently relax a user's query and all of these algorithms were tested and compared with FleXPath method [10] using the generated XML documents. From the perspective of effectiveness, we find that FleXPath generated far more relaxed queries compared with AQR. The reason behind this finding is that to get a large number of results, FleXPath has to relax a user's query and evaluate it until the root node of the query if necessary. However, AQR is able to stop unnecessary query

relaxations early as possible for a particular data source with the guideline of its conformed DTD. From the perspective of efficiency, AQR can obtain better performance than FleXPath in most cases where more relaxations are necessary to answer a user's query.

Third, based on the adaptive query relaxation, we proposed two scheduling strategies to support efficient top-k query evaluation - a brute-force scheduling strategy and a BT-based scheduling strategy. The brute-force scheduling strategy is a simple but not naive approach, which first generates all the relaxed queries according to the optional and disjunctive semantics of the corresponding XML elements, and then ranks the generated queries based on their weights. After that, we will evaluate the generated queries one by one in the descendant order. It is possible for the brute-force strategy to generate a large number of unqualified relaxed queries, which is the main shortcoming of the method. To efficiently address the issue, we also proposed BT-based scheduling strategy by computing the upper/lower bound and threshold values. The upper bound value is obtained by computing the weight of each relaxed query w.r.t. a data source while the lower bound value can be derived during query evaluation. By monitoring the updated values of lower/upper bound and threshold, we can determine which data source should be evaluated first and what kind of intermediate candidates are the top k most relevant results as early as possible, rather than accessing all the data sources and working out all the candidates. To formalize the scheduling procedures, we proposed two corresponding properties: *Data Source Determination and Switching* and *Result Determination*. Additionally, we also deployed an adaptive query relaxation strategy to filter out some unqualified edges in the query for some data sources based on schema information, which can further improve query evaluation efficiency. The experimental results demonstrated the BT scheduling strategy can gain more benefits when the value of top-k and the number of data sources are large.

Finally, we studied the problem of keyword search in XML. we designed an adaptive XML keyword search approach, called *XBridge*, that can derive the semantics of a keyword query and generate a set of effective structured queries by analyzing the given keyword query and the schemas of XML data sources. To derive the meaningful context for a keyword query, we recognized the different semantics of two schema nodes by deploying the ontology knowledge or comparing their descendant attributes/subelements. In XBridge, we proposed a scoring function that takes into account the structure of the results and the weight of each corresponding keyword-matched node in the results together, which can evaluate the quality of the results. To improve the performance of evaluation, we also designed an execution plan to retrieve the more qualified results as early as possible. Given a top-k keyword query, the k relevant results with the higher ranking scores would be searched and output efficiently where the relevance of the results is measured by our proposed ranking function.

7.2 Future Work

Because in this work we use the existing XQuery search engine to evaluate the structured queries, it is possible to repeatedly visit some nodes or compute some relationships. To further improve the performance of keyword search, in the future we will design a more efficient query evaluation plan that can reduce or eliminate the repeated computations.

Parallel Processing of Keyword Query: Nowadays, keyword search has been studied extensively in XML. Its appeal stems from the fact that keyword queries can be easily posed without knowing a query language and the schema or structure of the data being searched. For XML, where the data is viewed as a hierarchically-structured rooted tree, a natural keyword search semantic is to return all the nodes

in XML tree that contain all the keywords in their subtrees. However, this simple search semantics always result in a great number of intermediate results, many of which can not produce the final answers or are only remotely linked to the nodes containing the keywords. To address this problem, in this thesis we have proposed a method *XBridge* for processing keyword search where we used the existing XQuery search engine to evaluate the constructed answer templates. Although it improves the relevance and performance of keyword search, it is still possible to visit some nodes or compute some relationships repeatedly. To further improve the performance, we will design an optimal keyword query evaluation plan that can efficiently compute the qualified results and prune the unqualified keyword-matched nodes or intermediate results as early as possible. To construct the evaluation plan, we will analyze the relationships among all possible answer templates. As we know, in large XML data some of the answer templates are independent from each other while others may be dependent through some relations. The independent answer templates can be done in a parallel way while the dependent ones can be processed in an optimal schedule based on the overlapped relations. Therefore, our future approach shoud be adaptive to the XML data by combining the merits of the independent and dependent answer templates together, rather than merging all the keyword-matched nodes in a same way.

Efficient Processing of Multiple Keyword Queries: Another issue is motivated by the real application of efficiently processing a set of keyword queries over large XML data sources. As we know, most previous keyword search methods are based on the similar strategy that first retrieves the matched nodes and then produces the results by merging the retrieved nodes together. However, the straightforward strategy may become impractical when a set of keyword queries is coming at the same time. For instance, many users may access the same database at the same time. If we

process the set of queries one by one, some of them may wait for a long time. To address this issue, one way is to transform the set of keyword queries into different sets of structured queries. Based on the relationships among the structured queries, we can answer the set of keyword queries by processing the structured queries together. However, when the relationships are very complex, the transformation-based strategy may become too expensive to meet users' needs. Another way is to build an efficient index that can be used to identify the probabilistic reachability of each keyword query. Given a set of keyword queries, we first derive their corresponding reachabilities and then infer their relationships. Based on the relationships, one optimal execution plan may be designed. To do this, the first challenge is how to identify the probabilistic reachability over XML data for a keyword query. The second challenge is how to infer the relationships among the given set of keyword queries based on their reachabilities.

Bibliography

[1] Oracle Berkeley DB XML 2.3. http://www.oracle.com/database/berkeley-db/xml/index.html.

[2] Wutka dtd parser. http://www.wutka.com/dtdparser.html.

[3] XMark XML data generator. http://monetdb.cwi.nl/xml/index.html.

[4] *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*. Morgan Kaufmann, 2002.

[5] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.

[6] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[7] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, pages 141–, 2002.

[8] Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree pattern relaxation. In *EDBT*, pages 496–513, 2002.

[9] Sihem Amer-Yahia, Nick Koudas, Amélie Marian, Divesh Srivastava, and David Toman. Structure and Content Scoring for XML. In *VLDB*, pages 361–372, 2005.

[10] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *SIGMOD Conference*, pages 83–94, 2004.

[11] Sonia Bergamaschi, Silvana Castano, and Maurizio Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, 1999.

[12] Elisa Bertino, Giovanna Guerrini, and Marco Mesiti. A matching algorithm for measuring the structural similarity between an XML document and a DTD and its applications. *Inf. Syst.*, 29(1):23–46, 2004.

[13] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise dtds from xml data. In *VLDB*, pages 115–126, 2006.

[14] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML Schema Definitions from XML Data. In *VLDB*, pages 998–1009, 2007.

[15] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, pages 431–440, 2002.

[16] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. extensible markup language (xml) 1.0 (second edition). w3c recommendation, October 2000.

[17] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[18] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–, 2002.

[19] Pável Calado, Altigran Soares da Silva, Rodrigo C. Vieira, Alberto H. F. Laender, and Berthier A. Ribeiro-Neto. Searching web databases by structuring keyword-based queries. In *CIKM*, pages 26–33, 2002.

[20] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.

[21] Michael J. Carey and Donald Kossmann. On saying "enough already!" in sql. In *SIGMOD Conference*, pages 219–230, 1997.

[22] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[23] Sudarshan S. Chawathe. Comparing hierarchical data in external memory. In *VLDB*, pages 90–101, 1999.

[24] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD Conference*, pages 26–37, 1997.

[25] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD Conference*, pages 493–504, 1996.

[26] Chung-Min Chen and Yibei Ling. A sampling-based estimator for top-k query. In *ICDE*, pages 617–627, 2002.

[27] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *VLDB* [4], pages 263–274.

[28] Byron Choi, Gao Cong, Wenfei Fan, and Stratis D. Viglas. Updating Recursive XML Views of Relations. In *ICDE*, 2007.

[29] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting Changes in XML Documents. In *ICDE*, pages 41–52, 2002.

[30] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, pages 45–56, 2003.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[32] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.

[33] Shaul Dar, Gadi Entin, Shai Geva, and Eran Palmon. Dtl's dataspot: Database exploration using plain language. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 645–649. Morgan Kaufmann, 1998.

[34] Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors. *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 2003.

[35] Claude Delobel and Marie-Christine Rousset. A uniform approach for querying large tree-structured data through a mediated schema. In *International Workshop on Foundations of Models for Information Integration*, 2001.

[36] Paul F. Dietz. Maintaining order in a linked list. In *STOC*, pages 122–127. ACM, 1982.

[37] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[38] Hong Hai Do and Erhard Rahm. Coma - a system for flexible combination of schema matching approaches. In *VLDB*, pages 610–621, 2002.

[39] Angela C. Duta, Ken Barker, and Reda Alhajj. Ra: An xml schema reduction algorithm. In *ADBIS*, 2006.

[40] Ronald Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[41] Ronald Fagin. Fuzzy queries in multimedia database systems. In *PODS*, pages 1–10. ACM Press, 1998.

[42] Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

[43] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[44] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, New York, NY, USA, 2001. ACM Press.

[45] Sergio Flesca, Giuseppe Manco, Elio Masciari, Luigi Pontieri, and Andrea Pugliese. Detecting Structural Similarities between XML Documents. In *WebDB*, pages 55–60, 2002.

[46] Anna Formica. Similarity of xml-schema elements: A structural and information content approach. *Computer Journal*, 2007.

[47] Norbert Gövert and Gabriella Kazai. Overview of the Initiative for the Evaluation of XML retrieval (INEX) 2002. In *INEX Workshop*, pages 1–17, 2002.

[48] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.

[49] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD Conference*, pages 16–27, 2003.

[50] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, pages 305–316, 2007.

[51] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[52] Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, and Divesh Srivastava. Keyword Proximity Search in XML Trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[53] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB* [4], pages 670–681.

[54] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword Proximity Search on XML Graphs. In Dayal et al. [34], pages 367–378.

[55] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In Dayal et al. [34], pages 253–263.

[56] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[57] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD Conference*, pages 779–790, 2004.

[58] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[59] Laks V. S. Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng (Jessica) Zhao. On testing satisfiability of tree pattern queries. In *VLDB*, pages 120–131, 2004.

[60] Mong-Li Lee, Liang Huai Yang, Wynne Hsu, and Xia Yang. Xclust: clustering xml schemas for effective integration. In *CIKM*, pages 292–299, 2002.

[61] Guoliang Li, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, pages 31–40, 2007.

[62] Jianxin Li, Chengfei Liu, Jeffrey Xu Yu, and Rui Zhou. Efficient Top-k Search across Heterogeneous XML Data Sources. In *DASFAA*, pages 314–329, 2008.

[63] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 361–370. Morgan Kaufmann, 2001.

[64] Wen-Syan Li and Chris Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data Knowl. Eng.*, 33(1):49–84, 2000.

[65] Yunyao Li, Cong Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.

[66] Shaorong Liu, Wesley W. Chu, and Ruzan Shahinian. Vague Content and Structure (VCAS) Retrieval for Document-centric XML Collections. In *WebDB*, pages 79–84, 2005.

[67] Ziyang Liu and Yi Chen. Identifying meaningful return information for XML keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

[68] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *VLDB*, pages 49–58, 2001.

[69] Amélie Marian, Sihem Amer-Yahia, Nick Koudas, and Divesh Srivastava. Adaptive Processing of Top-K Queries in XML. In *ICDE*, pages 162–173, 2005.

[70] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top- queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[71] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[72] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*, pages 117–128, 2002.

[73] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29, 1999.

[74] Davood Rafiei and Alberto O. Mendelzon. Efficient Retrieval of Similar Time Sequences Using DFT. In *FODO*, pages 249–257, 1998.

[75] Davood Rafiei and Alberto O. Mendelzon. Efficient retrieval of similar time series. pages 75–89, 2000.

[76] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[77] Mayssam Sayyadian, Hieu LeKhac, AnHai Doan, and Luis Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, pages 346–355, 2007.

[78] Torsten Schlieder. Similarity Search in XML Data using Cost-Based Query Transformations. In *WebDB*, pages 19–24, 2001.

[79] Torsten Schlieder. Schema-driven evaluation of approximate tree-pattern queries. In *EDBT*, pages 514–532, 2002.

[80] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, Hong Kong, China, 2002.

[81] Albrecht Schmidt, Martin L. Kersten, and Menzo Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. In *ICDE*, pages 321–329, 2001.

[82] Stanley M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[83] Chong Sun, Chee Yong Chan, and Amit K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[84] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[85] Anja Theobald and Gerhard Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *EDBT*, pages 477–495, 2002.

[86] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for topx search. In *VLDB*, pages 625–636, 2005.

[87] Yannis Velegrakis, Renée J. Miller, and Lucian Popa. Mapping adaptation under evolving schemas. In *VLDB*, pages 584–595, 2003.

[88] Jason Tsong-Li Wang, Kaizhong Zhang, Karpjoo Jeong, and Dennis Shasha. A system for approximate tree matching. *IEEE Trans. Knowl. Data Eng.*, 6(4):559–571, 1994.

[89] Herbert Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *JACM*, 19(1):43–56, 1972.

[90] Yu Xu and Yannis Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD Conference*, pages 537–538, 2005.

[91] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. Similarity evaluation on tree-structured data. In *SIGMOD Conference*, pages 754–765, 2005.

[92] Weidong Yang and Baile Shi. Schema-aware keyword search over xml streams. In *CIT*, pages 29–34, 2007.

[93] Shanzhen Yi, Bo Huang, and Weng Tat Chan. Xml application schema matching using similarity measure and relaxation labeling. *Inf. Sci.*, 169(1-2):27–46, 2005.

[94] Cong Yu and H. V. Jagadish. Schema summarization. In *VLDB*, pages 319–330, 2006.

[95] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.

[96] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[97] Xuan Zhou, Julien Gaugaz, Wolf-Tilo Balke, and Wolfgang Nejdl. Query relaxation using malleable schemas. In *SIGMOD Conference*, pages 545–556, 2007.

Appendix A

Author's Publications

1. Jianxin Li, Chengfei Liu, Jeffrey Xu Yu and Rui Zhou, "Efficient Top-k Search across Heterogeneous XML Data Sources", The 13th International Conference on Database Systems for Advanced Applications (DASFAA), pp.314-329, New Delhi, India, 2008.

2. Jianxin Li, Chengfei Liu, Jeffrey Xu Yu, Jixue Liu, Guoren Wang and Chi Yang, "Computing Structural Similarity of Source XML Schemas against Domain XML Schema", the 9th Australasian Database Conference (ADC), pp.155-164, Wollongong, NSW, Australia, 2008.

3. Jianxin Li, Chengfei Liu, Rui Zhou and Bo Ning, "Processing XML Keyword Search by Constructing Effective Structured Queries", The Joint International Conferences on Asia-Pacific Web Conference (APWeb) and Web-Age Information Management (WAIM), pp.88-99, Suzhou, China, 2009.

4. Rui Zhou, Chengfei Liu and Jianxin Li, "Holistic Constraint-Preserving Transformation from Relational Schema into XML Schema", The 13th International Conference on Database Systems for Advanced Applications (DASFAA), pp.4-18, New Delhi, India, 2008.

5. Rui Zhou, Chengfei Liu, Junhu Wang, Jianxin Li, "Containment between Unions of XPath Queries", The 14th International Conference on Database Systems for Advanced Applications (DASFAA), pp.405-420, Australia, 2009.

6. Chi Yang, Chengfei Liu, Jianxin Li, Jeffrey Xu Yu and Junhu Wang, "Semantics based Buffer Reduction for Queries over XML Data Streams", the 9th Australasian Database Conference (ADC), pp.145-153, Wollongong, NSW, Australia, 2008.

7. Rui Zhou, Chengfei Liu, Jianxin Li and Junhu Wang, "Filtering Techniques for Rewriting XPath Queries Using Views", WISE, pp.307-320, Auckland, New Zealand, 2008.

8. Chengfei Liu and Jianxin Li, "Designing Quality XML Schemas from E-R Diagrams", The 7th International Conference on Web-Age Information Management (WAIM), pp.508-519, Hong Kong, China, 2006.

9. Nan Tang, Jeffrey Xu Yu, Kam-Fai Wong, Kevin Lu and Jianxin Li, "Accelerating XML Structural Join by Partitioning", the 16th International Conference on Database and Expert Systems Applications (DEXA), pp.280-289, Copenhagen, Denmark, 2005.