View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

SCALING SDI SYSTEMS VIA QUERY CLUSTERING AND AGGREGATION

ZHANG XI

(B.Eng., Shanghai Jiao Tong University, China)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

I wish to thank Professor Lee Mong Li, my supervisor, for her excellent and patient guidance throughout my MSc study and research. Her valuable comments and constant encouragement not only helped me to complete my work smoothly, but also inspired my interest in research to a great extent.

I would like to take this opportunity to thank Professor Wynne Hsu, whose constructive suggestions and keen attention helped to shape and enrich this work greatly.

I would also like to express my gratitude to Dr. Yang Lianghuai. His insightful and detailed comments contributed significantly to the success of this work.

Moveover, I am grateful to my friends and lab mates. Their friendship and company made the process of doing my MSc thesis a great pleasure.

Special gratitude goes to my parents. It was their support, understanding and love that made me able to overcome all the difficulties. I would like to dedicate this thesis to them. Dad and Mum, I love you so much!

Contents

Acknowledgements **i**

Summary **viii**

1 Introduction **1**

1.1 Motivation 1

1.2 Major Contributions 2

1.3 Organization of the Thesis 3

2 Background and Related Work **4**

2.1 XML-based SDI Systems 4

2.1.1 XFilter 5

2.1.2 XTrie 8

2.1.3 YFilter 13

2.2 Query Pattern Trees 16

2.2.1 Tree Edit Distance 17

2.2.2 Tree Aggregation 18

3 Scalable XML-based SDI System **24**

3.1 Framework 24

3.2 Query Clustering 27

3.2.1 Distance Function 28

3.3 Combining Query Clustering and Aggregation 37

3.3.1 $C \rightarrow A$ 38

3.3.2 $C + A$ 39

3.4 YFilter* 41

3.4.1 Motivating Example 41

3.4.2 Overview of YFilter* 44

3.4.3 NFA Construction — Information Maintained for Shredded Rooted Path 48

3.4.4 NFA Execution — Associate Matching Instance to NFA Runtime Stack Entry 53

4 Performance Study **62**

4.1 Experiment Setup 63

4.2 Scalability 65

4.3 Sensitivity Experiments 67

4.3.1 Clustering Granularity 67

4.3.2 Diversity of User Preference 70

4.3.3 Distribution of QPT 71

4.4 YFilter* versus YFilter 72

5 Conclusions and Future Work **73**

5.1 Conclusions 73

5.2 Future Work 74

List of Figures

2.1 Architecture of an XML-based SDI System 5

2.2 XFilter 6

2.3 Substring Decomposition 9

2.4 XTrie Index Structure 11

2.5 NFA Construction 14

2.6 An Example of NFA Execution 16

2.7 Examples of QPT and Tree Patterns 19

2.8 Tree Pattern Aggregation 19

3.1 XML-based SDI System with Query Clustering and Aggregation 26

3.2 Algorithm to Compute Aggregation Similarity 29

3.3 BMP Algorithm 30

3.4 Matching Relative Paths 32

3.5 Example for Aggregation Similarity Calculation 35

3.6 $C \to A$ 38

3.7 $C + A$ 39

3.8 Path Shred in YFilter 42

3.9 Two XML Document Trees 42

3.10 Valid Context 47

3.11 A Sample Tree Pattern for Decomposition in YFilter* 49

3.12 Tree Patterns 52

3.13 An NFA for Tree Patterns in Figure 3.12 52

3.14 The XML Document Used in YFilter* NFA Execution 54

3.15 Runtime Stack in YFilter* NFA Execution 55

4.1 Scalability 65

4.2 Precision vs Cluster Granularity 67

4.3 Time vs Cluster Granularity 68

4.4 Query Diversity 70

4.5 QPT Distribution 71

4.6 YFilter* vs YFilter 72

List of Tables

3.1 Information Maintained for Rooted Paths of QPT_0 51

3.2 Information Maintained for Shredded Rooted Paths of QPT_1, QPT_2, QPT_3 53

4.1 Parameters 63

Summary

The rapid growth of XML data on the Internet has necessitated the development of XML-based Selective Dissemination of Information (SDI) systems to quickly deliver useful information to the users based on their profiles or user subscriptions.

In this work, we primarily investigate how clustering and aggregation of user queries can help to increase the scalability of SDI systems. The subscriptions in XML-based SDI systems are typically specified in the form of XML queries. A key insight is that, the bottleneck of such systems lies in the large number of document-subscription matchings required. These matchings are very costly. To reduce the number of matchings required, we propose to cluster and aggregate user queries. A new distance function, called *aggregation similarity*, is designed to measure the similarity of query patterns. Based on this similarity measure, we cluster the query patterns into groups. By aggregating the query patterns within each group, we are able to reduce the number of document-subscription matchings required. This is achieved by mapping each original user query to a representative query obtained from aggregation, and the document-subscription matchings are only carried out against those representative queries, which are significantly smaller in number compared to original user queries.

The XML filtering technique used in our system is named YFilter*, which is developed based on YFilter. YFilter* enhances YFilter's ability to handle tree-structured XML queries. Experiment result shows that YFilter* is much more efficient than YFilter in handling tree-structured queries.

We have conducted extensive experiments to show that the proposed techniques are able to achieve high precision, high recall, while reducing the runtime requirement in XML-based SDI systems. Other experiments study the influence of various factors on the performance of the system.

Chapter 1

Introduction

1.1 Motivation

Selective Dissemination of Information(SDI) systems have proliferated as a result of the massive amount of information on the Internet. SDI application systems continuously collect information from various data sources, filter the data against user preferences, or profiles, and then deliver personalized information to the relevant users. Traditional SDI systems typically express user preferences/profiles in Information Retrieval(IR) style. A user profile is represented by a single keyword or a bag of keywords whereby simple string matching can be performed to retrieve the relevant documents. Various IR techniques can be used to speed up the filtering of documents.

XML, the eXtensible Markup Language [24], has become the de facto standard for data exchange on the Internet. The growth of XML resources has fuelled research on retrieving XML information more quickly and effectively. An XML-based Selective

Dissemination of Information (SDI) system [1] aims to distribute XML data to users based on their preferences/profiles. In contrast to traditional SDI systems, user profiles in XML-based SDI systems are usually expressed in XML query languages such as XPath [22], XML-QL [21], XQuery [26] etc. The IR-based techniques used in traditional SDI systems barely exploit the path information in XML. While the path information is able to capture the context of the user interests, and therefore leads to more accurate description of user preference, the matching process becomes expensive. Moreover, the number of the users can easily grow into millions when the XML-based SDI system is deployed on the Internet. This motivates the development of scalable XML-based SDI systems.

1.2 Major Contributions

This thesis examines how two techniques, query clustering and query aggregation, play important roles in the construction of a scalable XML-based SDI system. In addition, improvements are made to the state-of-the-art XML filtering technique, YFilter [10], to better handle tree-structured XML queries.

The major contributions in this thesis are:

1. Define a new distance function between user queries, called *aggregation similarity* for clustering queries.

2. Design two approaches to integrate clustering and aggregation of queries. The first method $C \rightarrow A$ first performs Clustering of query patterns followed by

Aggregation. The second method $C + A$ carries out Clustering and Aggregation at the same time.

3. Develop an efficient filtering method called YFilter*, which is based on YFilter [10], for the matching of queries which have predicates with path expressions, also known as *nested paths*, e.g. /a/b[c/d]/e.

4. Detailed performance study on the proposed methods to scale an XML-based SDI system.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 gives the background of the research and related works, including the XML-based SDI system, various XML filtering techniques, tree edit distance and tree aggregation technique. Chapter 3 describes the architecture of the proposed XML-based SDI system. We present two methods to combine query clustering and aggregation in SDI systems and devise a technique called YFilter* for tree pattern filtering. Experiment results are given in Chapter 4, and we conclude in Chapter 5.

Chapter 2

Background and Related Work

2.1 XML-based SDI Systems

Altinel et al describe the architecture for a generic XML-based SDI system in [1]. As shown in Figure 2.1, an XML-based SDI system has two inputs: user profiles and XML documents. User profiles describe the information preferences of individual users. They can be established explicitly by the users. In some systems, however, they can be learned automatically by the system through the application of machine learning techniques to user access traces. XML documents contain the information from various data sources. The main component of the system is an XML filtering engine which matches the incoming XML documents against the user profiles and decides which users/group of users the document should be directed to.

A key feature of SDI systems is that the roles of queries and data are reversed [28]. In a database system, large numbers of data items are indexed and stored, and queries

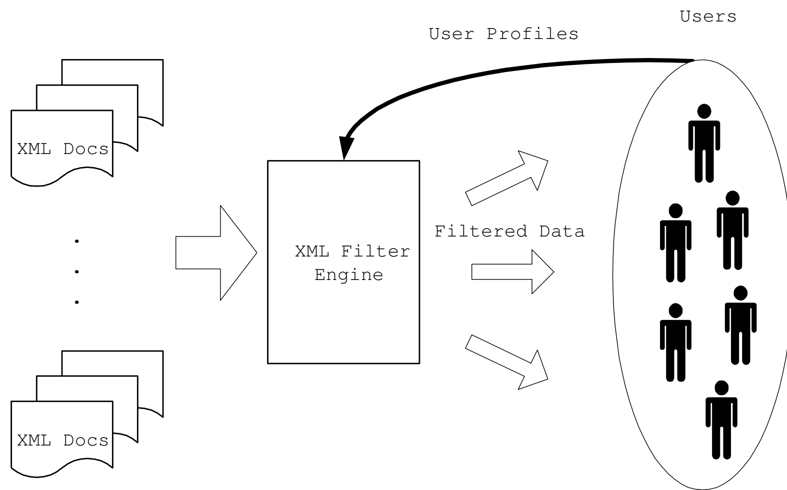
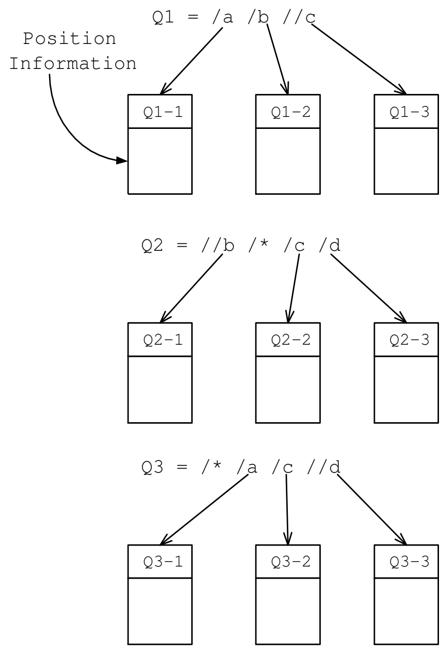


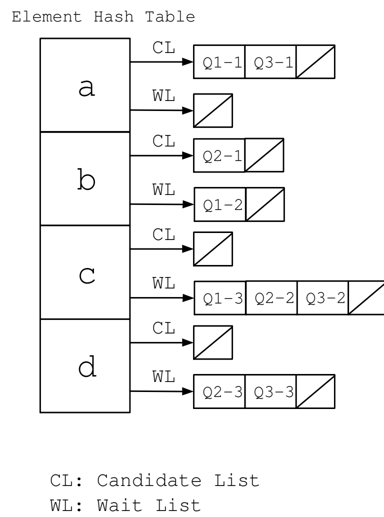
Figure 2.1: Architecture of an XML-based SDI System

are individually applied. In contrast, in an SDI system, large numbers of queries are stored, and the documents are individually matched to the queries. Therefore, advanced techniques which efficiently index the queries, such as XFilter [1], XTrie [4], and YFilter [10], have been developed to speed up the matching against documents. We will review these techniques in the following subsections.

2.1.1 XFilter

Altinel et al proposed an XML filtering technique, *XFilter*, based on *Finite State Machine(FSM)* in [1]. In XFilter, each XPath query without nested paths can be converted to an FSM containing a set of states. The embedded nest path queries are treated as independent queries. A post-processing step is needed to "glue" the nested path query and the one it is embedded in. The main structure in the filtering engine is an inverted list *Query Index*, which indexes these FSMs on the label of states in order to achieve





(a) Queries and Corresponding Path Nodes (b) Query Index

Figure 2.2: XFilter

simultaneous processing of XML data for multiple queries.

In Figure 2.2 (a), each query is decomposed to a set of path nodes, each of which represents a state in the FSM of this query. Note that no path nodes are generated for "*" node and relative path "//". In fact, each path node has some *position information* associated with it, such as the location of the path node in the order of path nodes for the query, the distance in *document levels* between this path node and its previous path node, etc. "*" and "//" is recorded in terms of that position information. The Query Index in Figure 2.2 (b) is actually a hash table indexed on the label of node. Associated with each unique node label are two lists, i.e. *Candidate List* and *Waiting List*. Each query can only have one current state in its FSM. It corresponds to a "current path node", which is stored in the Candidate List. Other path nodes of the query are stored in the Waiting List. A state transition in the FSM is represented by promoting a path node from the Wait List to the Candidate List. When executing XFilter, XML documents are parsed in an event-driven style using SAX parsing interface [14]. The encounter of a new element triggers a transition in FSMs. When a FSM reaches its accepting state, that is, the last path node of a query is promoted to the Candidate List, the corresponding query is said to be matched.

An important problem introduced by XML is the existence of wildcard "*" and relative path "//" in XPath queries. XFilter solves the problem by introducing FSM computation model and maintaining extra level information of documents. This method is nature in its idea and successful in its application. However, the drawback in XFilter is its space requirements. The space cost of XFilter is dominated by the number of tag

nodes (i.e. non-“*” nodes, “//” is not a node label but a relationship between two nodes here) in each XPath query. Another problem with XFilter is that it treats tree pattern XPath expression in a “flat” style. That is, XFilter first decomposes tree pattern to a set of root to leaf paths, and treats them as independent from each other. After filtering work for single paths is done, XFilter employs some post-processing to combine the paths shredded from the same XPath tree and judges the matching of the entire XPath tree pattern. The procedure makes XFilter keep tracks of all instances of partial matched tree patterns, which results in more processing overhead.

2.1.2 XTrie

XTrie [4] treats XPath queries as tree patterns as a whole. It decomposes tree patterns into collections of substrings and indexes them using a trie. XTrie is a sophisticated indexing technique. We introduce it here by first explaining how XPath queries are decomposed into substrings, then showing the two main components in the indexing structure of XTrie, and finally describing its matching algorithm.

Decompose tree pattern to substrings. XTrie interprets XPath queries/expressions as sets of substrings.

Definition 1 (Substring) *Given an XPath expression p, a sequence of element names $s = t_1 \cdot t_2 \ldots \cdot t_n$ is a substring of p if s is equal to the concatenation of the element names of the nodes along a path $< v_1, v_2, \ldots v_n >$ in the tree representation of p, such*

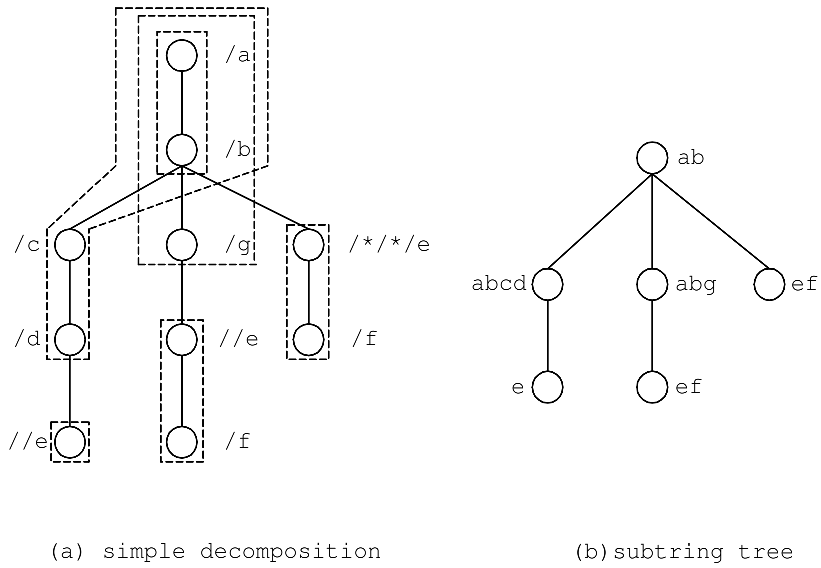


Figure 2.3: Substring Decomposition

that each v_i *is the parent node of* $v_{i+1} (1 \leq i < n)$ *and the label of each* v_i *(except for* v_1*) is prefixed only by "/".*

Definition 1 states that each pair of consecutive element names in a substring of p must be separated by parent-child ("/") operator. The nodes within the dashed box in Figure 2.3 are all substrings.

XTrie relies on a specific class of substring decompositions, referred as *simple decomposition*, for installing XPath expressions into the indexing structure. A simple decomposition of XPath expression p contains substrings from the following two sources:

1. *minimal substring decomposition* of p.

   A sequence of substrings $S =< s_1, s_2, \ldots, s_n >$ is a *substring decomposition* of

p, if each $s_i \in S$ is a substring of p and each node t_j in the tree representation of p is contained in $Path(s_i)$ for some $s_i \in S$, where $Path(s_i)$ is denoted for the path of s_i in the tree. This decomposition is *minimal*, if each $s_i \in S$ is of maximal length. In other words, no other substring contains s_i.

2. substrings *"taking notes"* of branching nodes.

   A substring "taking notes" of a *branching node* v in the tree representation of p is the maximal substring in p with v as its last node.

The substrings of the simple decomposition of p can be organized into a unique rooted tree, namely *substring tree*, as follows. Denote $S =< s_1, s_2, \ldots, s_n >$ to the simple decomposition of p, where the substrings are ordered based on the sequence in which they would be matched in an ordered matching of p and n is the number of substrings. The *root substring* is s_1 and the *parent substring* of s_j, where $j > 1$, is s_k if either

1. $Path(s_k)$ is a prefix of $Path(s_j)$, or

2. the last node of $Path(s_k)$ is the parent node of the first node of $Path(s_j)$.

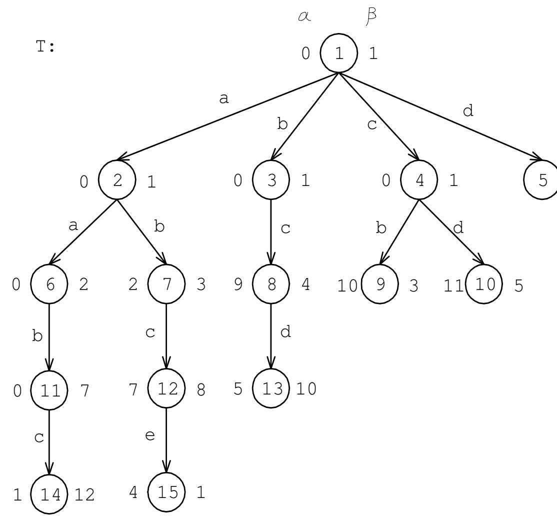
Example 1 *In Figure 2.3 (a), the simple decomposition of p is illustrated in dashed boxes, where all the substrings are from minimal decomposition except substring ab, which takes notes of branching node "$/b$". Figure 2.3 (b) is the corresponding substring tree of the simple decomposition in Figure 2.3 (a).* □

```
p1 = //a/a/b/c/*/a/b
p2 = /a/b[c/e]/*/b/c/d
p3 = /a/b[c/*/d]//b/c
p4 = //c/b//c/d/*/*/d
```

ST:

	Parent Row	Rel Level	Rank	Num Child	Next
1	0	[4,∞]	1	1	0
2	1	[3,3]	1	0	3
3	0	[2,2]	1	2	6
4	3	[2,2]	1	0	0
5	3	[4,4]	2	0	0
6	0	[2,2]	1	2	0
7	6	[1,1]	1	1	0
8	7	[2,2]	1	0	12
9	6	[2,∞]	2	0	0
10	0	[2,∞]	1	1	0
11	10	[2,∞]	1	1	0
12	11	[3,3]	1	0	0

(a)Substring Table



(b)Trie

Figure 2.4: XTrie Index Structure

There are two main components of an XTrie index structure: a *Substring-Table* and a *Trie*.

Index Structure: The Substring-Table. The substring-table(ST) contains one row for each substring of each indexed XPath expression. The rows in ST are physically clustered in order to group together the substrings belonging to the same XPath expression p, which are stored consecutively based on their order in the simple decomposition of p. To facilitate locating XPath expressions containing the same substring, the rows containing the same substring in ST are also logically linked as a list.

Index Structure: The Trie. XTrie, as implied by its name, uses a trie to index all distinct substrings obtained from XPath expressions. Denote N for a node in the trie, $label(N)$ is the string formed by concatenating the edge labels along the path from root to node N. The nodes in the Trie(T) and the rows in the Substring-Table(ST) are inter-

connected via the α value associated with each node in T, while the nodes in T are intra-connected to reflect suffix relationship via the β value associated with each node in T.

Example 2 *Figure 2.4 shows an example of XTrie structures for four XPath expressions. Notice that, in ST, the* parent row *value reflects the parent-child relationship between substring trees, while the* next *value links the same substring in different XPath expressions. In T, the α value indicates the first row of the substring with the same label and the β value indicates the number of the node in T which indexes its maximal suffix string, if any.* $\square$

Matching Algorithm. XTrie uses the event-driven SAX interface for XML document parsing. The XTrie index structure works in the following way. Trie T detects the occurrence of matching substrings as the input document is parsed. For each matching substring s detected, XTrie iterates through all the instances of s in the indexed XPath expressions by traversing the appropriate linked list of rows in the substring-table ST associated with s to check if the matched substring s corresponds to any non-redundant matching. Additional dynamical runtime information is maintained to ensure that only non-redundant matchings are checked.

XTrie relies on the decomposition of substrings to treat the XPath queries in a tree pattern style. It is space-efficient since the space cost of XTrie is dominated by the number of substrings in each tree patterns. The XTrie index structure together with

its matching algorithm makes it possible to reduce unnecessary index probes and avoid redundant matchings.

2.1.3 YFilter

YFilter [10] combines multiple tree patterns into a single *Nondeterministic Finite Automata*(NFA). In YFilter, any single path expression written using "/", "//" and nodes labelled with element name or wildcard "*" can be transformed into a regular expression, and thus there exists an FSM which accepts the language described by such an expression [12]. Like XFilter, YFilter decomposes each tree pattern query into a set of single path queries, then combines their corresponding FSMs into a single NFA, where all common prefixes of paths appear only once. YFilter employs some post-processing to decide the matching of a tree pattern from the matching of its shredded paths.

Construction of NFA. Each single path XPath expression is viewed as the concatenation of location steps. Each location step is modelled as one or more transitions in the NFA as follows.

1. "$/a$" or "$/*$"

   Modelled as a state s_1 linked to another state s_2 via a directed edge labelled a or "*".

2. "$//a$" or "$//*$"

   Modelled as a state s_1 linked to a state s_2, and s_2 further linked to a third state s_3, where s_2 is a special state with a self-loop labelled "*". The directed edge between

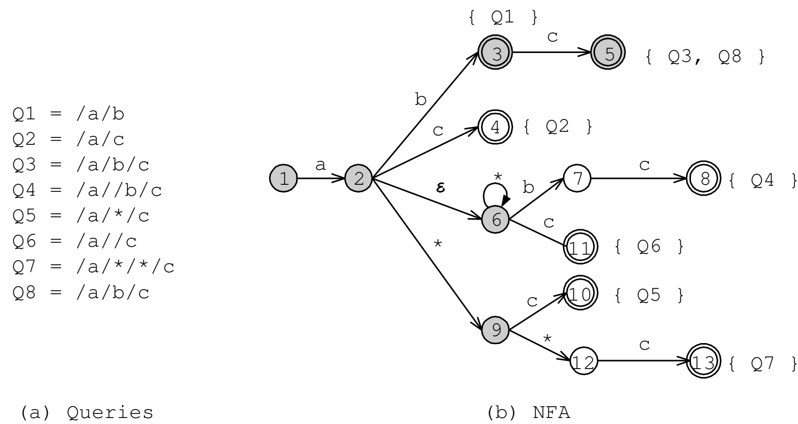


Figure 2.5: NFA Construction

s_1 and s_2 is labelled "ε", and the directed edge between s_2 and s_3 is labelled a or "*".

The special symbol "*" matches any element, and the symbol "ε" is used to mark a transition that requires no input.

In the NFA construction, XPath expressions with the same prefix share the same states and the corresponding transitions. The construction of NFA can be better illustrated by the following example.

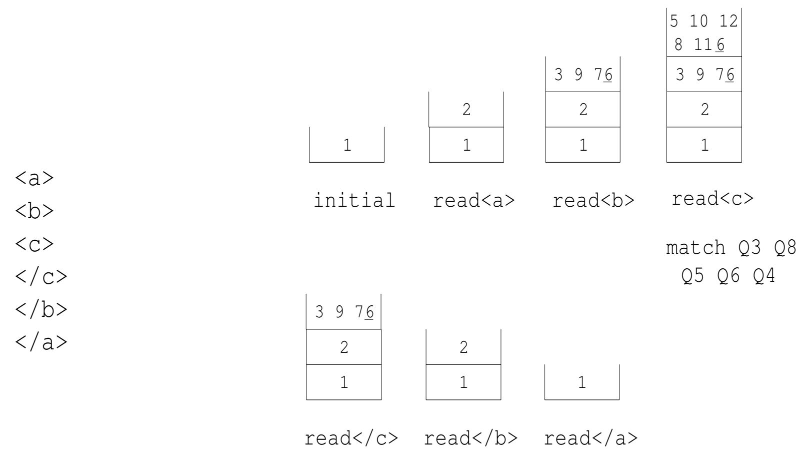
Example 3 (NFA Construction) *Figure 2.5 shows an NFA constructed for eight queries. A circle denotes a state, which is numbered for easy reference. Two concentric circles denote an accepting state; such states are marked with the IDs of the queries they represent. A directed edge represents a transition. The symbol on an edge represents the input that triggers the transition. Note that, as stated above, "*" matches any input and "ε" requires no input. In Figure 2.5, shaded circles represent states shared by queries. Common prefixes shared by all queries appear only once in the NFA.* □

NFA Execution. Similiar to XFilter and XTrie, YFilter executes the NFA in an event-driven style. YFilter uses a runtime stack in the execution, and there are multiple active states in the NFA. For each element encountered, four types of transitions are checked for each active state:

1. All target states triggered by the label of incoming element are added to a set for "target states", say S_t.
2. Check whether current state can be triggered by "*". If so, the corresponding target state is also added to S_t.
3. If the current state itself is a target state of an "ε"-transition, in other words, the current state has a self-loop, then the state itself is added to S_t.
4. Finally, if the current state can be triggered by an "ε"-transition, its corresponding target state, which is a state with self-loop, is processed recursively according to 1-3 above. The resulting target states are also added to S_t.

Example 4 *Figure 2.6 (b) shows the evolution of the contents of the runtime stack, when executing the NFA in Figure 2.5 (b) on the example XML document in Figure 2.6 (a). Each state in the NFA is represented by its ID. An underlined ID in the stack entry indicates that the state has a self-loop.* □

YFilter exploits the commonality among path queries by merging the common prefix of paths so that they are processed at most once. YFilter is able to efficiently handle



(a) XML Document Fragment (b) Runtime Stack

Figure 2.6: An Example of NFA Execution

XPath queries with no predicates, i.e. single path queries, and queries with simple value-based predicates. Similar to XFilter, it requires an expensive post-processing step for queries with nested paths.

2.2 Query Pattern Trees

The user profile model used in XML-based SDI system is usually XPath [22], as in [1, 4, 10]. XPath is a language for addressing parts of an XML document that was designed for use by both the XSLT Transformation(XSLT) [23] and XPointer [25] language. XPath provides a flexible way to specify path expressions. When an XML document is modelled as a tree, as in DOM parsing interface [20], XPath expressions are patterns that can be matched by the XML tree.

Yang defines in [29] a *query pattern tree* for XML queries.

Definition 2 (Query Pattern Tree (QPT)) *A query pattern tree (QPT) is a rooted tree* $QPT =< V, E >$*, where* V *is the vertex set, and* E *is the edge set. Each vertex has a label whose value is in* $\{$*"*", "//"*$\} \cup$ *tagSet, where tagSet is the set of all the element and attribute names in the underlying Document Type Definition (DTD) .*

It is easy to translate an XPath expression to a QPT and vice versa.

2.2.1 Tree Edit Distance

As we can see in the following chapter, we want to know to which extent two QPTs are similar to each other. Traditionally, the comparison of tree is carried out based on a pattern matching technique called *tree edit distance*. There is considerable previous work on finding edit distance between trees [5, 6, 8, 9, 17, 18, 19, 30]. Most algorithms are direct descendants of the dynamic programming techniques for finding 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 edit distance algorithms is the set of edit operations allowed. An early work in this area is by Selkow [17], which allows inserting the deleting of single nodes at the leaves, and relabelling of nodes anywhere in the tree. The work by Chawathe in [7] 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 [18, 19, 27, 30].

Expanding upon these more basic operators, Chawathe et al define in [8] a move operator that can move a subtree as a single edit operation, and in subsequent work [5] copying and its inverse, gluing, of subtrees is allowed. The approaches in [8, 5] are heuristic approaches and the algorithm in [5] operates on unordered trees, making it unsuitable for computing distances between XML documents.

Nierman and Jagadish develop in [15] a structural similarity metric for XML documents based on an "XML aware" tree edit distance. Nierman and Jagadish generalize Chawathe's approach in [7] by allowing such operations as tree insertions and deletions.

So far, there is no direct work on the comparison of XML query pattern trees. Though tree edit distance seems to be a natural choice for comparing QPTs, a close look into it reveals that, tree-edit-distance-based approaches, including Nierman and Jagadish's "XML aware" tree edit distance, is not suitable for this task. The main difficulty comes from the relative path "//", which is abundant in QPTs.

2.2.2 Tree Aggregation

Chan et al [3] develope a *tree aggregation* technique to combine multiple queries, represented in the form of *tree patterns*, into a generalized pattern to reduce the storage requirements as well as to speed up the document-subscription matching process.

Definition 3 (Tree Pattern (TP)) *A tree pattern is an unordered labelled tree that specifies content and structure conditions on an XML document. A tree pattern* $TP =<$

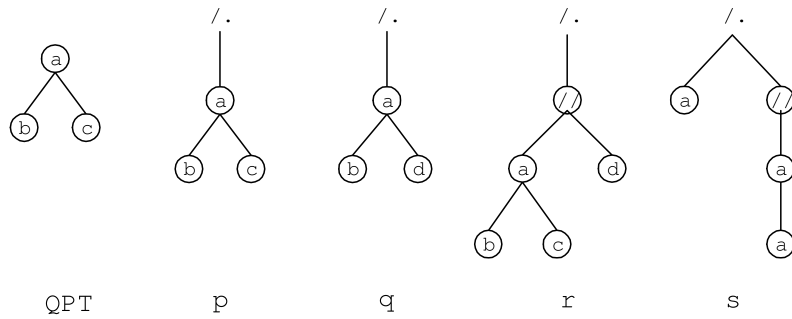


Figure 2.7: Examples of QPT and Tree Patterns

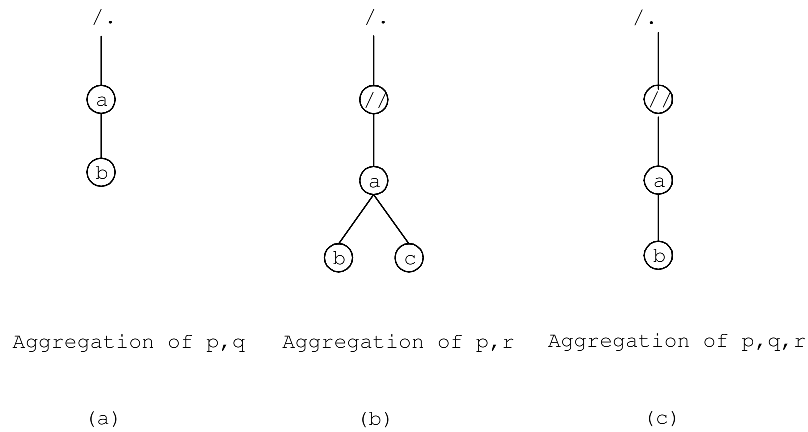


Figure 2.8: Tree Pattern Aggregation

$V, E >$*, where* V *is the vertex set, and* E *is the edge set. Each vertex, expect the root, has a label with its value in* $\{$*"*", "//"*$\} \cup$ *tagSet. The root vertex is labelled with a special symbol "/.".*

Tree Pattern is actually a generalization of *QPT*. A QPT can be converted to a tree pattern by adding a special root node labelled "/.". Figure 2.7 shows four tree patterns, p, q, r, s. Since a QPT can be easily converted to a tree pattern. These two terms are used interchangeably when the exact reference is clear with the context.

Similar to other work on subscription aggregation [16], the tree pattern aggregation essentially involves aggregating an initial set of subscriptions S into a smaller set A such that any document that matches some subscription in S also matches some subscription in A. The subscriptions here are modelled as tree patterns. It is guaranteed in [3] that, though there is typically a "loss in precision" associated with such aggregation, the documents matched by the aggregated set A is a superset of those matched by the original set S. In other words, all the documents matched by S are matched by A.

The essence of Chan's tree aggregation method is an algorithm to calculate the *Least Upper Bound*(LUB) of two tree patterns. The concept of least upper bound is similar to that in the *lattice* theory. In order to understand the LUB concept used in tree aggregation, we briefly review the related concepts.

Definition 4 (Contained) *A tree pattern q is said to be contained in another tree pattern p, denoted by $q \sqsubseteq p$, if and only if for any XML tree T, if T satisfies q, T also satisfies p.*

Definition 5 (Equivalent) *Two tree patterns p and q are said to be equivalent, denoted by $p \equiv q$, if and only if $p \sqsubseteq q$ and $q \sqsubseteq p$.*

Definition 6 (Upper Bound) *An upper bound of two tree patterns p and q is a tree pattern u such that $p \sqsubseteq u$ and $q \sqsubseteq u$. An upper bound of a set S is a tree pattern U, denoted by $S \sqsubseteq U$, such that $p \sqsubseteq U$, $\forall p \in S$.*

Definition 7 (Least Upper Bound (LUB)) *The least upper bound(LUB) of p and q , denoted by $p \sqcup q$, is an upper bound u of p and q such that for any upper bound $u^{'}$ of p and q, $u \sqsubseteq u^{'}$. The LUB of a set S, denoted by $\sqcup S$, is an upper bound $U^{'}$ of S such that for any upper bound $U^{'}$ of S, $U \sqsubseteq U^{'}$.*

In the calculation of LUB, or the *most precise* aggregated tree pattern, for two tree patterns p and q, two types of generalization are considered, namely, *position-preserving* generalization and *off-position* generalization. We illustrate these two kinds of generalization by the following two examples.

Example 5 (Position-Preserving Aggregation.) *Consider the aggregation of the tree patterns p and q in Figure 2.7. The two tree patterns contain a common sub-pattern, namely, a node labelled a with a child node labelled b. This pattern occurs in the same position with respect to the root nodes of p and q. The* position-preserving *generalization captures this class of common sub-patterns in the aggregated tree pattern. Figure 2.8(a) shows the aggregated tree pattern for p and q.* □

Example 6 (Off-Position Aggregation.) *Next, we consider the aggregation of the tree patterns p and r in Figure 2.7. Both tree patterns have in common the sub-pattern: a node labelled a with child nodes labelled b and c. However, this sub-pattern is located in different positions with respect to the root nodes in the two tree patterns. The* off-position *generalization captures this type of common sub-patterns in the aggregated tree pattern. Figure 2.8(b) shows the resulting aggregated tree pattern.* □

Denote $Subtree(u, p)$ to the subtree of tree pattern p rooted at node u, referred to as a sub-pattern of p. Let $label(u)$ and $Child(u, p)$ be the label of node u and the set of child nodes of u respectively. Suppose u_{root} and v_{root} are root nodes of tree patterns p and q respectively. Then $p \sqcup q$ is computed from the LUB of $Subtree(u, p)$ and $Subtree(v, q)$, where $u \in Child(u_{root}, p)$ and $v \in Child(v_{root}, q)$. More specifically, let sub-pattern $p^{'} = Subtree(u, p)$ and sub-pattern $q^{'} = Subtree(v, q)$. If $q^{'} \sqsubseteq p^{'}(p^{'} \sqsubseteq q^{'})$, then the LUB of $p^{'}$ and $q^{'}$ is $p^{'}(q^{'})$. Otherwise, the LUB is constructed by a set of sub-patterns $\{x, x^{'}, x^{''}\}$ where

- x represents the *position-preserving* generalization of $p^{'}$ and $q^{'}$, which captures common sub-patterns located in the same position of $p^{'}$ and $q^{'}$. The root node of x is labelled by $MaxLabel(u, v)$ which is defined as

$$MaxLabel(u, v) = \begin{cases} label(u) & \text{if } label(u) = label(v), \\ // & \text{if } label(u) = \text{“//”} \text{ or } label(v) = \text{“//”}, \\ * & \text{otherwise} \end{cases} \tag{2.1}$$

  The subtrees of x are LUBs of each child subtree of $p^{'}$ and each child subtree of $q^{'}$.

- $x^{'}$ and $x^{''}$ represent the *off-position* generalization of $p^{'}$ and $q^{'}$ respectively, which captures the common sub-patterns located in different positions of $p^{'}$ and $q^{'}$. The root node of $x^{'}(x^{''})$ is labelled “//”, and the subtrees of $x^{'}(x^{''})$ are LUBs of $q^{'}(p^{'})$ itself and each child subtree of $p^{'}(q^{'})$.

While the LUB algorithm in [3] only gives out the method to compute LUB for two tree patterns, an important property of LUB proven in [3] makes it the foundation for computing LUB for a set of tree patterns.

Property 1 (Sequence Independent) *Given any set S of tree patterns, $\sqcup S$ always exists and is unique up to equivalence.*

Property 1 states that given a set of tree patterns, the aggregation result is the same regardless of the sequence of aggregation. That is, the final aggregated pattern obtained is influenced only by the set of tree patterns involved. Thus the LUB algorithm for two tree patterns can be used to calculate the LUB of a set of tree patterns.

Example 7 *Figure 2.8(c) shows the LUB of p, q and r regardless of the sequence of aggregation.* $\Box$

Chapter 3

Scalable XML-based SDI System

3.1 Framework

It is crucial that a scalable XML-based SDI system should provide support for the efficient and timely delivery of relevant XML documents to a large, dynamic group of users. Given the large number of user subscriptions and the growing number of XML documents, the goal of a scalable XML-based SDI system is to reduce the user subscriptions judiciously as well as speed up the filtering of incoming XML documents.

Previous work mainly focuses on speeding up the filtering of XML documents [1, 4, 10]. In this work, we investigate a different approach to improve the efficiency of XML-based SDI system. The basic idea is to reduce the number of matchings required by grouping similar queries together and aggregating the queries in each group into a representative query. Matching is performed only on these representative queries which are substantially smaller than the number of original queries.

It is intuitive that when N_q original queries are clustered and aggregated to N_r representative queries ($N_r << N_q$), the processing time t of the latter is much less than that of the former. Therefore, the average response time of each original query, i.e. $t_{avg} = \frac{t}{N_q}$, is significantly reduced.

When mapping each original query to a representative query, we ensure that the XML documents fetched by the former are covered by those fetched by the latter. On the other hand, the number of documents fetched by the representative query but not by the original query, i.e. the *loss of precision*, should not be too large. Otherwise, the user will not be satisfied. As a result, we need to cluster similar queries and aggregate only those similar queries to generate the representative query.

As shown in Figure 3.1, after the original queries, $Q_1, Q_2, \ldots, Q_n$, are issued by users, they are clustered and aggregated into representative queries, Q_{rep1}, Q_{rep2}, $\ldots$, Q_{repm}. Admittedly, these representative queries are more general than the original queries. Since $m << n$, the number of queries in the systems are significantly reduced. Matching and retrieval are performed only against these representative queries.

In the filtering stage of the system, we develop YFilter*, which is a XML filtering technique based on YFilter. YFilter* inherits the merits of YFilter, including exploiting the commonality between queries and the simultaneous processing of queries. Meanwhile, it enhances YFilter's ability to handle queries with nested paths.

The tradeoff in the above scalable XML-based SDI system with query clustering and aggregation is the loss of precision due to the matching to generalized representative

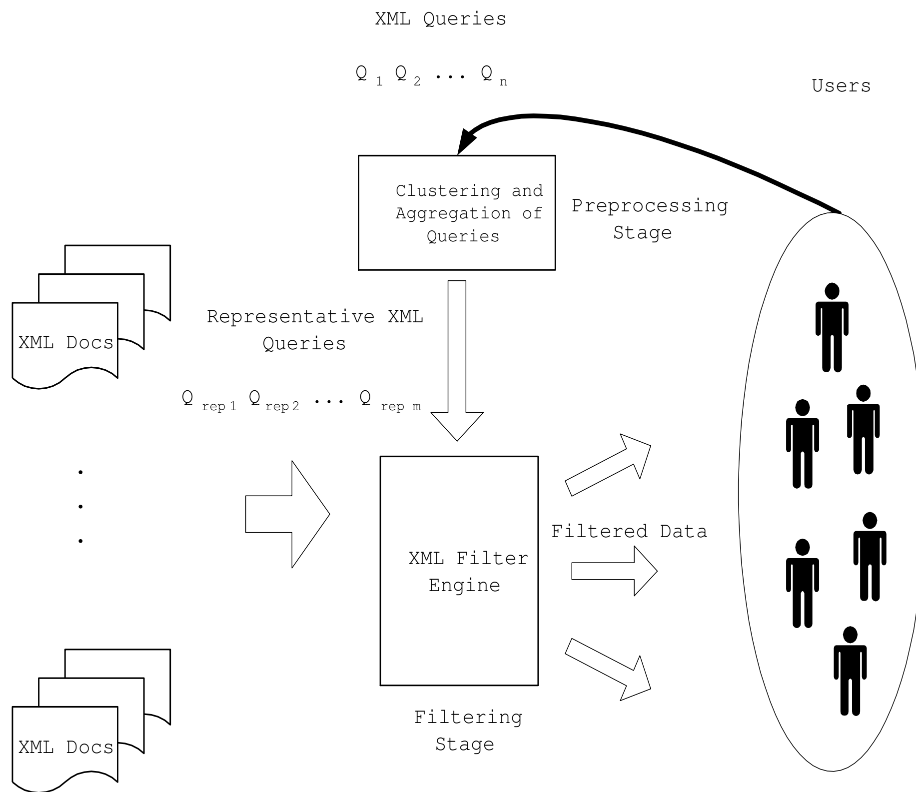


Figure 3.1: XML-based SDI System with Query Clustering and Aggregation

queries and the savings obtained through reducing the number of matchings needed. The experiment results presented in the next chapter show that the loss in precision is around 20% to 30% with the average response time improved by an order of 1 magnitude.

3.2 Query Clustering

The bottleneck of SDI systems lies in the large number of document-subscription matchings required. These matchings are also very costly. To reduce the number of such matchings, we propose to cluster and aggregate user queries. Many clustering techniques exist and are applicable here. In this work, we employ the hierarchical clustering method.

The clustering is aimed at generating clusters for aggregation. Similar to other clustering tasks, it involves a distance function to determine how similar two tree patterns are. As we have already stated in Section 2.2.1, the traditional comparison of tree patterns is carried out based on tree edit distance. More recently, Lee et al develop in [13] a novel and non-tree-edit-distance-based algorithm to measure the structural and semantical similarity between DTDs. [13] focuses on the structural similarity and the cardinality constraints("?", "+" and "*") in DTDs. Note that the cardinality constraint "*" in DTDs has different meaning from the wildcard "*" in QPTs.

However, none of the above techniques are suitable for our application because of the existence of relative path "//" in XML query pattern trees. A "//" node in a QPT can be matched to zero or more nodes in another QPT. The similarity between QPTs with

one or more “//” nodes should be carefully defined and computed. Although the tree-edit-distance-based algorithms work well for nodes with normal tags or even wildcard, they do not take the special property of “//” into account. Lee’s method for clustering DTD, where there is no “//” node, does not consider the property of “//” either.

Therefore, we design a new distance function which handles relative paths, and identifies tree patterns that will result in minimal information loss after aggregation. We call it the *aggregation similarity* function.

3.2.1 Distance Function

The main idea behind the *aggregation similarity* function is to determine the proportion of the nodes of a tree pattern that can be matched by the nodes of another tree pattern. That is, the aggregation similarity function calculates the maximal number of matching nodes between two tree patterns first, and then normalize this maximal matching number by the square root of the product of their sizes. $Size(p)$ and $Size(q)$ denote the number of nodes contained in the tree pattern p and q respectively. Then the normalization factor used for calculating the *aggregation similarity* of p and q is given by $\sqrt{Size(p) \times Size(q)}$. Note that the artificial root node “/.” is excluded from the count of size.

Figure 3.2 gives the details of algorithm *AggrSim*, where $Nodes(p)$ and $Nodes(q)$ denote the set of nodes of tree pattern p and q respectively. *AggrSim* calls a subroutine *MaxMatch* to compute the maximal number of matching nodes between two

Algorithm AggrSim(p, q)

Input: Tree patterns p and q

Output: Aggregation similarity of p and q

for each $u_i \in Nodes(p)$ and $v_j \in Nodes(q)$ **do** $M[u_i, v_j] = null$;

$M[u_{root}, v_{root}] = MaxMatch(u_{root}, v_{root})$;

$sim = \frac{M[u_{root}, v_{root}]}{\sqrt{Size(p) \times Size(q)}}$;

return sim;

Algorithm MaxMatch(u, v)

Input: u and v are nodes of p, q respectively

Output: maximal number of matching nodes in $Subtree(u, p)$ and $Subtree(v, q)$

if $(M[u, v] \neq null)$ **then return** $M[u, v]$;

else $BMP = BestMatchedPairs(u, v)$;

if(($label(u)$ = "//" **and** $label(v)$ = "//") **or** ($label(u) \neq$ "//" **and** $label(v) \neq$ "//"))

then $M[u, v] = \sum_{(u_i, v_j) \in BMP} M[u_i, v_j] + IsMatch(u, v)$;

else

  if ($label(u) \neq$ "//" **and** $label(v)$ = "//") **then**

    $N_{cand1} = \max\{MaxMatch(u, v_i) | \forall v_i \in Child(v, q)\}$;

    if $(\sum_{(u_i, v_j) \in BMP} M[u_i, v_j] = 0)$ **then** $N_{cand2} = 0$;

    else $N_{cand2} = \sum_{(u_i, v_j) \in BMP} M[u_i, v_j] + 1$;

    $N_{cand3} = \max\{MaxMatch(u_j, v) | \forall u_j \in Child(u, p)\}$;

  else ($label(u)$ = "//" **and** $label(v) \neq$ "//") **then**

    $N_{cand1} = \max\{MaxMatch(u_i, v) | \forall u_i \in Child(u, p)\}$;

    if $(\sum_{(u_i, v_j) \in BMP} M[u_i, v_j] = 0)$ **then** $N_{cand2} = 0$;

    else $N_{cand2} = \sum_{(u_i, v_j) \in BMP} M[u_i, v_j] + 1$;

    $N_{cand3} = \max\{MaxMatch(u, v_j) | \forall v_j \in Child(v, q)\}$;

  $M[u, v] = \max(N_{cand1}, N_{cand2}, N_{cand3})$;

return $M[u, v]$;

Figure 3.2: Algorithm to Compute Aggregation Similarity

tree patterns. The subroutine *MaxMatch* is the core function of aggregation similarity calculation. It recursively computes the maximal number of matching nodes between $Subtree(u, p)$ and $Subtree(v, q)$ in a bottom-up manner. Note that this maximal matching number between each pair of subtrees of p and q is computed only once. It is stored in a matrix M. Successive references to this value can be retrieved from M directly.

Algorithm BestMatchedPairs(u, v)

Input: u and v are nodes of tree patterns p, q respectively

Output: BMP list of the child nodes of u and v

If either u or v is a leaf node **then return** null;

for each $u_i \in Child(u, p)$

    for each $v_j \in Child(v, q)$

        $sim = \frac{MaxMatch(u_i, v_j)}{\sqrt{Size(Subtree(u_i, p)) \times Size(Subtree(v_j, q))}}$;

        Add triple (u_i, v_j, sim) to list l;

Sort l in descending order by sim;

do

    Get first triple (u_i, v_j, sim) from l, add pair (u_i, v_j) to list bmp;

    Delete form l all triples involving u_i and v_j;

until $l = null$;

return bmp;

Figure 3.3: BMP Algorithm

We use a list called BMP to store the *Best Matched Pairs* of subtrees rooted at the child node of u and v with regard to tree pattern p and q respectively. Best matched pairs are pairs with maximal number of matching nodes. Figure 3.3 shows *BestMatchedPairs* algorithm, which calculates the BMP list for the child nodes of node u and v. *BestMatchedPairs* calls *MaxMatch* to determine the number of matching nodes between two

subtrees, and then picks up the matching pattern of u and v's child nodes, which gives priority to pairs with larger value in normalized number of matching nodes. The length of the BMP list is $\min(|Child(u,p)|, |Child(v,q)|)$.

The calculation of aggregation similarity lays special emphasis on effectively handling the relative path "//". In the core function *MaxMatch*, cases are categorized by whether the label of the node is "//" or not. Before discussing the details of each case, we introduce the following functions, which are used as routines in the algorithm. We have

$$MaxMatch = \sum_{(u_i,v_j)\in BMP} MaxMatch(u_i, v_j) + IsMatch(u, v) \tag{3.1}$$

where

$$IsMatch(u,v) = \begin{cases} 1 & if\ label(u) = label(v), label(u), label(v) \in \{\text{"*"}, \text{"//"}\} \cup tagSet \\ & or\ label(u) = \text{"*"} \mid \text{"//"} \\ & or\ label(v) = \text{"*"} \mid \text{"//"}, \\ 0 & otherwise \end{cases} \tag{3.2}$$

Equation 3.1 can be used to find the maximal number of matching nodes between $Subtree(u,p)$ and $Subtree(v,q)$ when both u and v are labelled "//", or both u and v are not labelled "//". However, when only one of the two nodes is labelled "//", we need to consider whether "//" is matched to zero, one or more nodes, as illustrated in *Case 3*.

Case 1: $label(u)$ and $label(v)$ are in *tagSet* or wildcard "*".

Use the BMP list to calculate the number of matching nodes of the child nodes of u and v. Then check whether the labels of u and v match each other or not.

Case 2: Both $label(u)$ and $label(v)$ are "//".

The final number of matching nodes is the maximal number of matching nodes among the child nodes of u and v, which can be obtained via the BMP list, plus 1, which indicates that the labels of u and v are considered to be matched. In fact, *Case 2* is a special case of *Case 1*, and therefore be combined in the algorithm, as shown in *MaxMatch*. Here, we list *Case 2* as a separate case in order to state explicitly on how aggregation similarity deals with "//" in different circumstances.

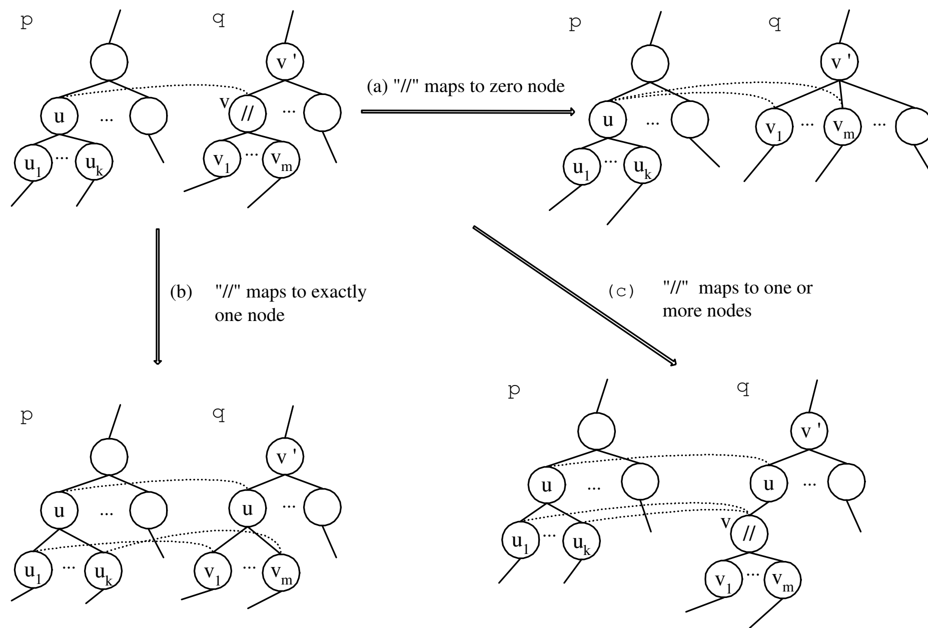


Figure 3.4: Matching Relative Paths

Case 3: One and only one of $label(u)$ and $label(v)$ is "//".

Without loss of generality, let $label(u)$ $\neq$"//" and $label(v)$ $=$"//", as shown in

Figure 3.4.

(a) "//" maps to an empty chain.

As shown in Figure 3.4 (a), all the child nodes of "//", $v_1, v_2, \ldots, v_m$ are treated as the child nodes of the parent node of "//",i.e. node $v^{'}$. Compute the maximal number of matching nodes between $Subtee(u, p)$ and $Subtree(v_i, q)$, $i = 1, 2, \ldots, m$. Suppose v_j is the one whose rooted subtree has the largest number of matching nodes with $Subtree(u, p)$. This matching pattern is the most profitable when "//" maps to zero nodes, but we can not tell whether it will outperform other possibilities. In other words, when "//" maps to one or more nodes, more number of matching nodes might be obtained. As a result, we use a variable N_{cand1} to record the maximal number of matching node when "//" maps to zero nodes, and that subtree $Subtree(u, p)$ matches to subtree $Subtree(v_j, q)$ becomes a candidate matching pattern. The algorithm goes on exploiting other possibilities of the matching of "//" first, and delays the decision until later.

(b) "//" maps to exactly one node.

Node u is matched to node v("//") in Figure 3.4 (b), which can also be interpreted as that "//" is materialized by $label(u)$. In this case, the total number of nodes matched is given by the number of matching nodes among the child nodes of u and v plus 1. This number is stored in variable N_{cand2}. The BMP list computed for u and v can be used to determine the matching nodes among child nodes of u and v. The matching pattern associated with N_{cand2}

is a candidate matching pattern.

Note that there is special case here. When the sum of matching nodes in the BMP list of u and v is 0, that is, there is no matching between the child nodes of u and v, it is meaningless to materialize "//" node with $label(u)$. Under this circumstance, N_{cand2} is set to 0 other than 1.

(c) "//" maps to one or more nodes.

In this case, "//" is matched to multiple nodes, as shown in Figure 3.4 (c). "//" is materialized by $label(u)$ first, and then the matching should go on along the path which will yield the most number of matching nodes. Therefore, the maximal numbers of matching nodes between each subtree rooted at the child node of u, $Subtree(u_1, p)$,$Subtree(u_2, p)$,. . .,$Subtree(u_k, p)$ and subtree $Subtree(v, q)$ are computed. The number corresponding to the subtree $Subtree(u_j, p)$ with the largest matching number to $Subtree(v, q)$ is recorded in N_{cand3}. The pattern that, "//" is materialized by $label(u)$ and $Subtree(v, q)$ is matched to $Subtree(u_j, p)$, becomes a candidate matching pattern.

After exploiting all possible matching patterns of "//", the final number of maximal matching nodes is determined by

$$MaxMatch = \max(N_{cand1}, N_{cand2}, N_{cand3}) \tag{3.3}$$

Note that, in *Case 3* above, when only one of u or v is labelled "//", *MaxMatch* tries

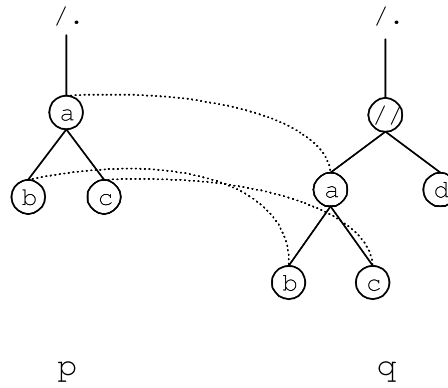


Figure 3.5: Example for Aggregation Similarity Calculation

to compute the maximal number of matching nodes in a "position-preserving" manner by matching "//" to exactly one node, as in *Case 3 (b)*. The best matching method in this case yields N_{cand2} number of matching nodes. At the same time, *MaxMatch* will try to determine the number of matching nodes in an "off-position" manner when "//" is mapped to zero or multiple nodes, as in *Case 3 (a)* and *(c)*. The best matching method under each circumstance results in N_{cand1} and N_{cand3} number of matching nodes respectively. Finally, *MaxMatch* selects the maximum of N_{cand1}, N_{cand2} and N_{cand3} to determine the matching approach.

The following example illustrates the various cases.

Example 8 (Aggregation Similarity Calculation) *Consider tree patterns p and q in Figure 3.5. Both root nodes of p and q have only one child. Thus, the only pair in the BMP list of two root nodes is $(a, \text{"//"})$, which stands for $(Subtree(a,p), Subtree(\text{"//"}, q))$. The matching of these two sub-patterns falls into* Case 3.

If "//"-node in q is not matched by any node in $Subtree(a,p)$, i.e. *"//" is an*

empty chain, then the algorithm will try to match $Subtree(a, p)$ *to subtrees rooted at "//"-node's child, i.e.* $Subtree(a, q)$ *and* $Subtree(d, q)$. *Matching* $Subtree(a, p)$ *with* $Subtree(a, q)$ *will result in 3 pairs of matchings: (a, a), (b, b) and (c, c). On the other hand, if the algorithm matches* $Subtree(a, p)$ *with* $Subtree(d, q)$, *then no matchings will be obtained. Therefore, the former matching pattern is adopted, which makes* N_{cand1} *of* $Subtree(a, p)$ *matching* $Subtree(\text{"//"}, q)$ *equals to 3.*

N_{cand2} *of* $Subtree(a, p)$ *matching* $Subtree(\text{"//"}, q)$ *is 0, because if "//"-node is matched to exactly one node in* p, *the only matching can be found is "//"-node in* q *matched to "*a*"-node in* p *and no matchings between "//"-node's children and "*a*"-node's children can be found, thus* N_{cand2} *is set to zero.*

N_{cand3} *of* $Subtree(a, p)$ *matching* $Subtree(\text{"//"}, q)$ *is determined by matching* $Subtree(\text{"//"}, q)$ *to the "*a*"-node's child, which has the largest number of matching nodes. Either matching* $Subtree(\text{"//"}, q)$ *to* $Subtree(b, p)$ *or to* $Subtree(c, p)$ *will result in zero matching node, so* N_{cand3} *of* $Subtree(a, p)$ *matching* $Subtree(\text{"//"}, q)$ *is 0. Now the algorithm finds out that the maximal number of matching nodes for* $Subtree(a, p)$ *and* $Subtree(\text{"//"}, q)$ *is 3, when the "//"-node is matched to zero node.*

Given that the size of the tree patterns p,q *are 3 and 5 respectively, we have* $AggrSim = \frac{3}{\sqrt{3 \times 5}} = 0.775.$ □

3.3 Combining Query Clustering and Aggregation

Our XML-based SDI systems aims at achieving the scalability in the presence of large amounts of user subscriptions. We propose query clustering and aggregation to reduce the number of subscriptions that will be filtered against by the XML documents in the filtering engine. The clustering and aggregation of queries can be viewed as a pre-processing stage in the SDI system.

Having defined the similarity distance between two pattern trees in the previous section, we are able to cluster the queries based on this similarity measure, namely *aggregation similarity*. In addition, in Section 2.2.2, we have already described Chan's tree aggregation method [3] in details. Chan's approach will be adopted in our system. The remaining task is how to combine these two pre-processing techniques. In this section, we propose two ways to achieve this objective.

1. $C \rightarrow A$. Clustering followed by Aggregation.

2. $C + A$. Clustering and Aggregation are carried out at the same time.

Despite the different approaches adopted by $C \rightarrow A$ and $C+A$, both of them involves hierarchical clustering, which should have a stop criterion. There are two alternatives here. One is the *number of representative queries*, the other is *minimal aggregation similarity*.

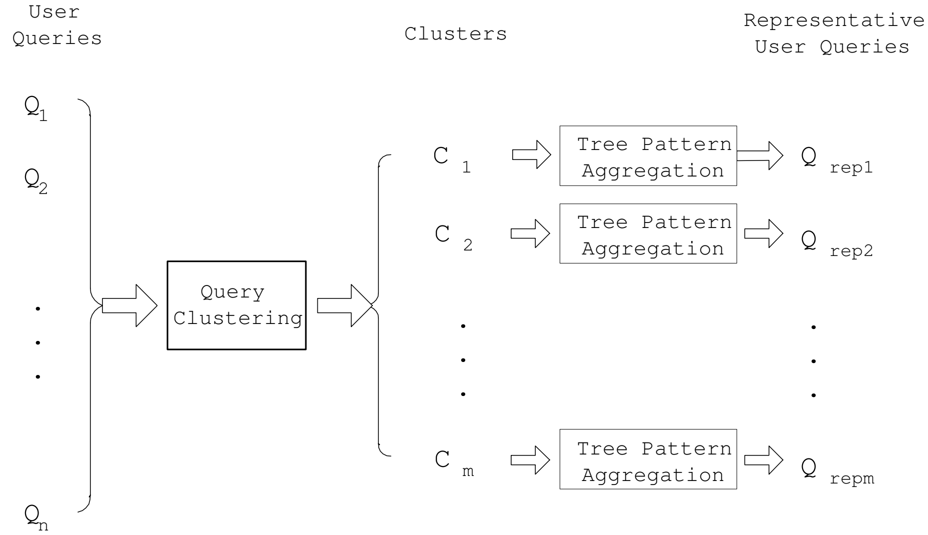


Figure 3.6: $C \rightarrow A$

3.3.1 $C \rightarrow A$

Figure 3.6 shows the $C \rightarrow A$ approach.

$C \rightarrow A$ finds clusters of similar queries before aggregating the queries within each cluster. In Figure 3.6, user queries Q_1, Q_2, ..., Q_n are directed to the clustering component first. There, queries are clustered based on aggregation similarity. The output of the clustering component are several clusters $C_1, C_2, \ldots, C_m$, where $m << n$. These clusters are further directed to the aggregation component. The aggregation component performs tree pattern aggregation and generates one representative query for each input cluster. Hence, the number of representative queries obtained is the same as the number of clusters. In Figure 3.6, Q_{repi} is the representative query obtained from cluster C_i. All the representative queries $Q_{rep1}, Q_{rep2}, \ldots, Q_{repm}$ become the input for the filtering engine in the SDI system.

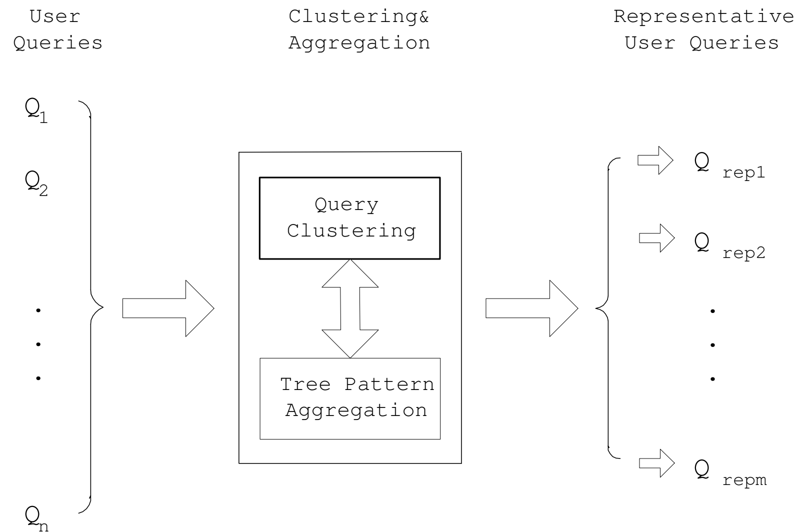


Figure 3.7: $C + A$

3.3.2 $C + A$

The second approach, $C + A$, however, does not separate aggregation from clustering. On the other hand, $C + A$ integrates aggregation to the hierarchical clustering performed on user queries. Figure 3.7 shows the process of $C + A$.

Clustering and aggregation in $C + A$ are carried out together and can be viewed as a "black-box" as drawn in Figure 3.7. To better understand how $C + A$ works, we now disassemble the "black-box" and go into the details. Given a set S containing user queries $Q_1, Q_2, \ldots, Q_n$, $C + A$ does the follows.

1. Compute the pairwise aggregation similarity of queries with in set S.

2. Select the most similar pair of query patterns, say (Q_i, Q_j).

3. Aggregate Q_i and Q_j. Denote $Q_i \cup Q_j$ to the result.

4. Update set S by inserting $Q_i \cup Q_j$ and deleting Q_i and Q_j.

$$S = S - Q_i - Q_j + Q_i \cup Q_j \tag{3.4}$$

5. Check whether stop criterion is satisfied or not. If yes, stop. Otherwises, go to *1*.

The choice of the most similar pair of queries can be viewed as a step in the hierarchical clustering, which is followed by query aggregation. Clustering and aggregation is carried out alternately in the above process. The above process continues until the stop criterion of clustering is satisfied. Since query aggregation is carried out at the same time as clustering, we will finally obtain the representative queries $Q_{rep1}, Q_{rep2}, \ldots, Q_{repm}$ for the clusters that have been generated implicitly in this process. Again, $Q_{rep1}, Q_{rep2}, \ldots, Q_{repm}$ are the input to the filtering engine.

Recall the important *sequence independent* property of the LUB-based tree aggregation in Section 2.2.2, which states that the final result of aggregation depends only on the set of tree patterns involved and is independent of the sequence. In $C + A$, clusters are formed implicitly during the process. The clusters formed in $C \rightarrow A$ and $C + A$ might be different even for the same stop criterion. Therefore, we expect the quality of aggregation is different in the two approaches, which is largely determined by the quality of clustering. We will study the quality of clustering and aggregation of $C \rightarrow A$ and $C + A$ in the experiment part.

3.4 YFilter*

3.4.1 Motivating Example

After the query clustering and aggregation stage in the SDI system, each original user query maps to an representative query, which is among the output of the pre-processing stage. These representative queries are the input of the filtering engine.

In the filtering engine, we develop a filtering technique called YFilter*, which is based on YFilter [10], to do the filtering of XML documents.

YFilter adopts an NFA-based approach to carry out the XML filtering in the SDI system in order to achieve scalability [10]. The commonality of path expressions are merged in the construction of NFA, thus reducing the storage requirement of NFA . However, YFilter focuses on a subset of XPath queries, namely queries with no predicates and queries with simple value-based predicates.

For *nested path* queries, or queries with predicates containing path expressions, e.g. $/a/b[c/d]/e$, YFilter does the follows.

1. Decompose nested path queries into separate *rooted paths*.
2. Construct the NFA as if all rooted paths are independent from each other.
3. Execution NFA against XML documents.
4. For each query with nested paths, perform post-processing to ensure that all the rooted paths of it are satisfied.

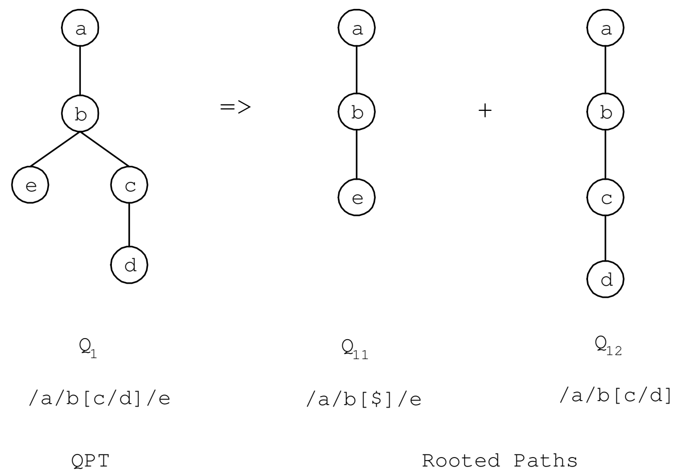


Figure 3.8: Path Shred in YFilter

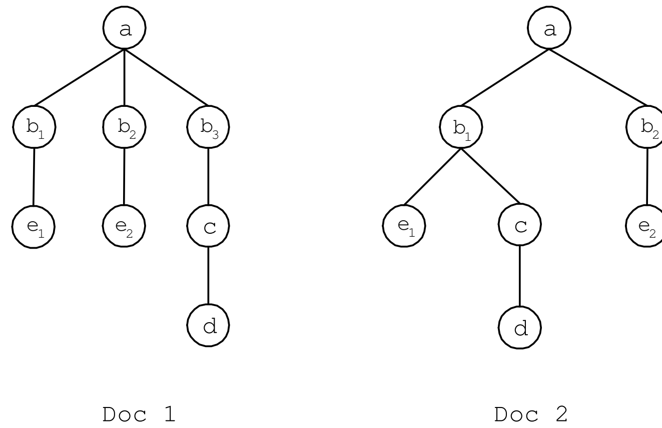


Figure 3.9: Two XML Document Trees

where

Rooted path is defined as a root to leave path in a query pattern tree.

We illustrate this procedure and highlight the cost of post-processing with an example.

Example 9 *Figure 3.8 shows the tree pattern representation of query* $Q_1 : /a/b[c/d]/e$*. YFilter shreds this tree pattern into the paths* $Q_{1_1} : /a/b[\$]/e$ *and* $Q_{1_2} : /a/b[c/d]$*, where*

$ *is to mark that a predicate should be evaluated here after the XML document parsing is finished. This is the post-processing described in* Step 4 *above.*

Figure 3.9 depicts two XML document trees, Doc 1 *and* Doc 2*, where element nodes with the same tag are numbered in a pre-order traversal of the tree.*

As Doc 1 *is being parsed, YFilter maintains the following information:*

(a) Q_{1_1} *is matched at* e_1 *via* b_1*;*

(b) Q_{1_1} *is matched at* e_2 *via* b_2*;*

(c) Q_{1_2} *is matched at* d *via* b_3.

However, one cannot tell whether Q_1 *is matched at this point when* Step 3 *is finished. In the post-processing* Step 4*, YFilter finds that:*

(a) The Q_{1_1} *matching instance containing* b_1 *does not share the same* b *with the* Q_{1_2} *matching instance containing* b_3*;*

(b) The Q_{1_1} *matching instance containing* b_2 *does not share the same* b *with the* Q_{1_2} *matching instance containing* b_3 *either.*

Since there are no matching instances of Q_{1_1} *and* Q_{1_2} *sharing the same* b*,* Q_1*, where these two rooted paths are shredded, is not matched by* Doc 1.

Similarly, for Doc 2*, we have:*

(a) Q_{1_1} *is matched at* e_1 *via* b_1*;*

(b) Q_{1_2} *is matched at* d *via* b_1*;*

(c) Q_{1_1} *is matched at* e_2 *via* b_2*.*

In the post-processing step, we find out that:

The Q_{1_1} *matching instance containing* b_1 *shares the same* b *with the* Q_{1_2} *matching instance.*

Hence, Q_1 *is matched by* Doc 2. □

We can see from the above example that the size of the information maintained by YFilter for post-processing is proportional to the number of matching instances in the document. Useless matching instances of paths cannot be discarded and actual matching of tree patterns can not be told until the end of parsing. This motivated us to design YFilter*, which is based on YFilter, to efficiently handle *nested path* queries without post-processing.

3.4.2 Overview of YFilter*

Each nested path query is actually a tree pattern. YFilter* use the same techniques as in YFilter to decompose a tree pattern into a set of rooted paths and to construct the NFA by viewing each rooted path as being independent from each other. In the NFA execution, YFilter* also uses YFilter's runtime stack approach.

YFilter* differs from YFilter in that, it maintains additional information for each shredded rooted path to reflect its relationship to other rooted paths in its corresponding

tree pattern. Moreover, during the NFA execution for each document, YFilter* associates matching instances of shredded paths with runtime stack entry. YFilter* discards useless matching instances of shredded paths and finds the matching of tree patterns as early as possible.

The following example illustrates the basic idea of YFilter*.

Example 10 *Consider again the query in Figure 3.8, and the XML documents in Figure 3.9.*

For Doc 1*, when YFilter* is about to finish processing* b_1*, all its descendent nodes would have already been parsed. It only finds:*

The matching instance of Q_{1_1} *at* e_1

and

NO matching instance of Q_{1_2} *so far.*

Since there is no matching instance of Q_{1_2} *containing* b_1*, the* Q_{1_1} *matching instance containing* b_1 *cannot be part of a matching instance of* Q_1 *in* Doc 1*, and hence it can be discarded.*

In contrast, for Doc 2*, before YFilter* finishes processing* b_1*, it finds:*

The matching instance of Q_{1_1} *at* e_1

and

The matching instance of Q_{1_2} *at* d*.*

YFilter determines that these two matching instances share the same* b *and concludes that tree pattern* Q_1 *has been matched by* Doc 2*.*

Additionally, when YFilter parses* Doc 2 *further, it ignores the Q_{1_1} matching instance at e_2 because the tree pattern Q_1, from which Q_{1_1} is shredded, has already been matched by this document.* □

As we can see from *Example 10*, the *branch point node* b of the QPT in Figure 3.8 is very important in the tree pattern matching. In fact, it can be considered as a *context* node.

Figure 3.10 illustrates the XML documents corresponding to the XML document trees in Figure 3.9.

When processing *Doc 1*, a matching instance of Q_{1_1} is found when the first "$< e >$" tag in *Doc 1* is encountered. Thereafter, no matching instance of Q_{1_2} is found until after the first "$< /b >$" tag is encountered. b_1, the branch point node of Q_{1_1} and Q_{1_2} in Q_1, is materialized by the first b element in *Doc 1*. No matching instance of Q_{1_2} is found within the *valid context* of this branch point node, which is the shadowed area of *Doc 1* in Figure 3.10. Hence, in the above example, this matching instance of Q_{1_1} is discarded.

In contrast, when processing *Doc 2*, a matching instance of Q_{1_2} is found within the *valid context* of the matching instance of Q_{1_1} containing the first b element in *Doc 2*. Therefore, Q_1 is matched. In Figure 3.10, the materialized branch point node is the first b element in *Doc 2* and the corresponding *valid context* is shadowed.

In tree pattern matching, it is a key issue to determine the *valid context* information for each matching instance of shredded paths at runtime. This leads to two novel features in YFilter*:

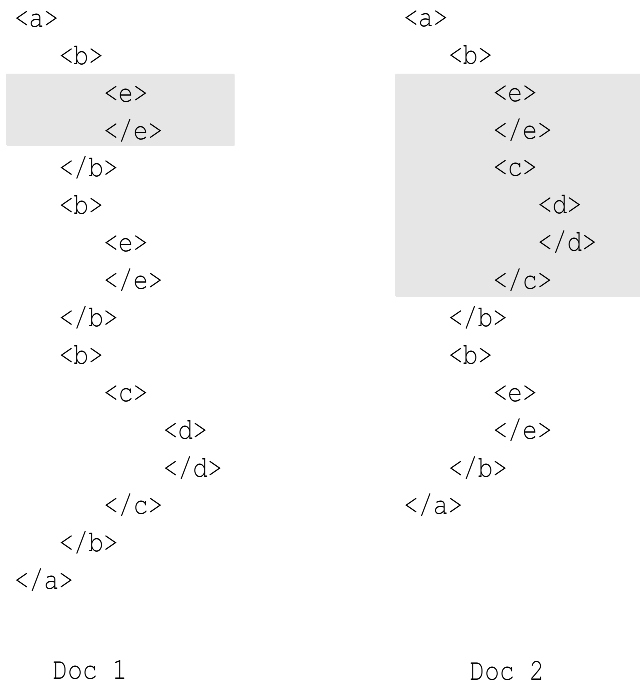


Figure 3.10: Valid Context

1. Maintain information of each shredded path related to branch point node.

   The information is used to capture the relationship between the shredded rooted paths of the same tree pattern.

2. Associate matching instances with runtime stack entries in NFA execution.

   In the NFA execution, YFilter* associates each matching instance of a shredded path with a runtime stack entry, whose popup indicates the invalidation of context of a certain branch point node. The actual stack entry where the a certain matching instance should be associated can be calculated by backtracking the runtime stack, together with the information we maintained for its corresponding shredded path.

The following subsections describe the details of these two features in YFilter*.

3.4.3 NFA Construction — Information Maintained for Shredded Rooted Path

Similar to the approach in YFilter, YFilter* is an NFA-based approach. It decomposes each QPT to a set of rooted paths before constructing the NFA. The NFA is constructed by assuming that each shredded rooted path is independent from each other.

YFilter* collects additional information while shredding rooted paths from QPTs. The aim of the information is to facilitate tree pattern checking in addition to path checking in the execution of NFA. As a result, such information is mainly about the branch point nodes in tree patterns and of two consecutive rooted paths. To facilitate evaluation, we impose an arbitrary order on the rooted paths.

Suppose a QPT Q is decomposed into an ordered list $\{Q_1, Q_2, \ldots, Q_k\}$. The predecessor of Q_j, denoted by $prev(Q_j)$, is given by Q_{j-1}, for $j = 2, 3, \ldots, k$. The successor of Q_j, denoted by $succ(Q_j)$, is given by Q_{j+1}, for $j = 1, 2, \ldots, k-1$. Note that there is no $prev(Q_1)$ or $succ(Q_k)$.

Definition 8 (Branch Point(BP)) *Given a QPT Q_i which is decomposed into an ordered list of rooted paths, $\{Q_{i_1}, Q_{i_2}, \ldots, Q_{i_k}\}$. The branch point of any two consecutive paths, Q_{i_j} and $Q_{i_{j+1}}$, for $j = 1, 2, \ldots, k-1$, is the common node of these two paths that is closest to their leaf nodes, denoted by $BP(Q_j, Q_{j+1})$.*

Example 11 *In Figure 3.11, we have*

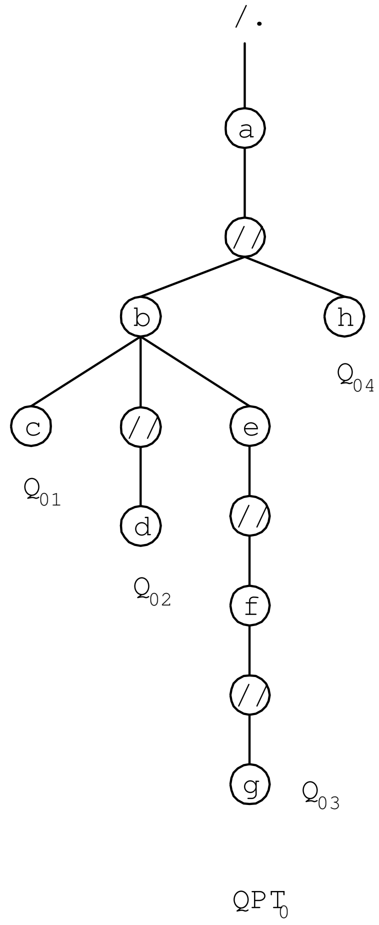


Figure 3.11: A Sample Tree Pattern for Decomposition in YFilter*

p_j	Q_{0_1}	Q_{0_2}	Q_{0_3}	Q_{0_4}
$succ$	Q_{0_2}	Q_{0_3}	Q_{0_4}	-
$prev$	-	Q_{0_1}	Q_{0_2}	Q_{0_3}

and $BP(Q_{0_1}, Q_{0_2}) = b$, $BP(Q_{0_2}, Q_{0_3}) = b$ *and* $BP(Q_{0_3}, Q_{0_4}) =$ *"//".*$\square$

Next, YFilter* collects the following information for each path p with respect to its preceding and succeeding paths in the ordered list of rooted paths.

1. $Count_{//}(p, prev(p))$ - number of "//" nodes from $BP(prev(p), p)$ to the leaf node of p.

2. $Count_{non-//}(p, prev(p))$ - number of non-"//" nodes from $BP(prev(p), p)$ to the first "//" node if it exists, or to the leaf node of p if such "//" node does not exist.

3. $Count_{//}(p, succ(p))$ - number of "//" nodes from $BP(p, succ(p))$ to the leaf node of p.

4. $Count_{non-//}(p, succ(p))$ - number of non-"//" nodes from $BP(p, succ(p))$ to the first "//" node if it exists, or to the leaf node of p if such "//" node does not exist.

To better under the above notations, we illustrate them by the following example.

Example 12 *Consider the rooted paths of QPT_0 in Figure 3.11. We have $BP(prev(Q_{0_3}), Q_{0_3}) = BP(Q_{0_2}, Q_{0_3}) = b$. Since there are two "//" nodes from the branch point b to*

p	$Count_{//}(p, prev)$	$Count_{non-//}(p, prev)$	$Count_{//}(p, succ)$	$Count_{non-//}(p, succ)$
Q_{0_1}	-1	-1	0	2
Q_{0_2}	1	1	1	1
Q_{0_3}	2	2	3	0
Q_{0_4}	1	0	-1	-1

Table 3.1: Information Maintained for Rooted Paths of QPT_0

the leaf node g of Q_{0_3}, hence $Count_{//}(Q_{0_3}, prev) = 2$. There are two non-"//" nodes from the branch point b to the first "//" node on this path, namely, the node b itself and node e. Thus, $Count_{non-//}(Q_{0_3}, prev) = 2$.

Similarly, $BP(Q_{0_3}, succ(Q_{0_3})) = BP(Q_{0_3}, Q_{0_4}) =$ "//", $Count_{//}(Q_{0_3}, succ) = 3$ and $Count_{non-//}(Q_{0_3}, succ) = 0$. Note that the branch point is a "//" node itself.

When calculating $Count_{//}(Q_{0_1}, succ)$ and $Count_{non-//}(Q_{0_1}, succ)$, the corresponding branch point is also b. However, there is no "//" node from the branch point b to the leaf node c, therefore $Count_{//}(Q_{0_1}, succ) = 0$ and $Count_{non-//}(Q_{0_1}, succ)$ is the number of non-"//" nodes from the branch point to the leaf, which equals to 2.

Table 3.1 lists the $Count_{//}$ and $Count_{non-//}$ values for paths shredded from QPT_0. Note that -1 is used to indicate that such a value does not exist, as in the cases for $(Q_{0_1}, prev)$ and $(Q_{0_4}, succ)$. □

When all QPTs have been decomposed into paths, YFilter* use YFilter's approach to construct the NFA. We have introduced the details of YFilter NFA construction in Section 2.1.3. The same techniques are adopted in YFilter*.

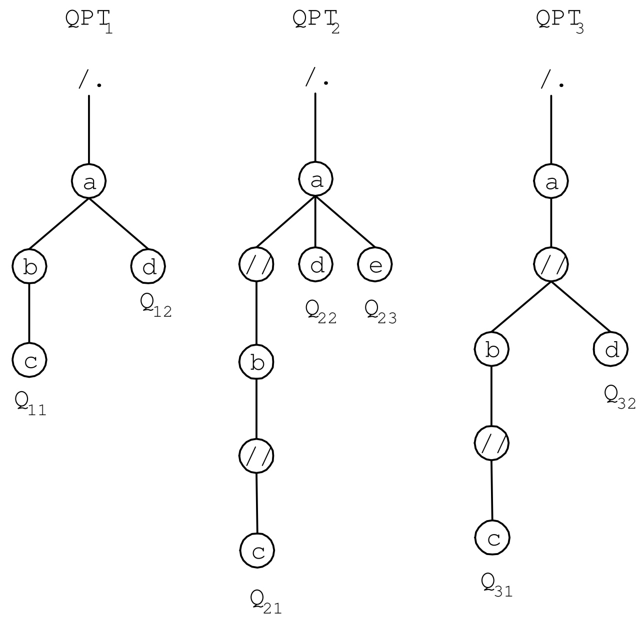


Figure 3.12: Tree Patterns

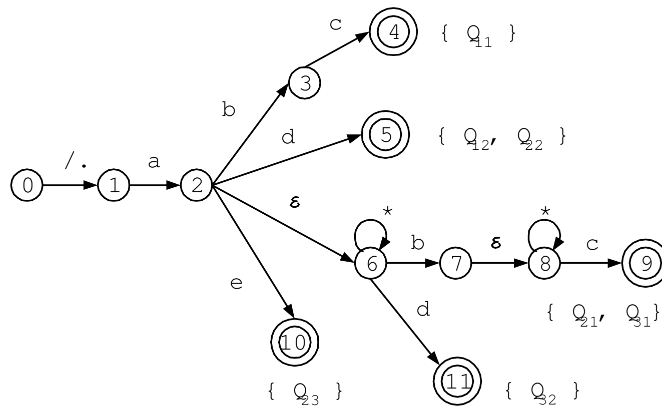


Figure 3.13: An NFA for Tree Patterns in Figure 3.12

p	$Count_{//}(p, prev)$	$Count_{non-//}(p, prev)$	$Count_{//}(p, succ)$	$Count_{non-//}(p, succ)$
Q_{1_1}	-1	-1	0	3
Q_{1_2}	0	2	-1	-1
Q_{2_1}	-1	-1	2	1
Q_{2_2}	0	2	0	2
Q_{2_3}	0	2	-1	-1
Q_{3_1}	-1	-1	2	0
Q_{3_2}	1	0	-1	-1

Table 3.2: Information Maintained for Shredded Rooted Paths of QPT_1, QPT_2, QPT_3

Example 13 *Figure 3.13 shows the NFA constructed for QPT_1, QPT_2, QPT_3 in Figure 3.12. Table 3.2 lists the information maintained by YFilter* for each shredded path.* □

3.4.4 NFA Execution — Associate Matching Instance to NFA Runtime Stack Entry

We use the standard SAX [14] interface parser to parse the XML documents. The main component of the NFA execution is a runtime stack. When parsing an XML document, a stack entry is pushed into the runtime stack when the parser encounters a begin-of-element event. The top stack entry is popped when the parser encounters an end-of-element event.

Similar to YFilter, the execution of NFA in YFilter* follows an event-driven fashion and uses a runtime stack to allow the backtracking of multiple active paths dynamically.

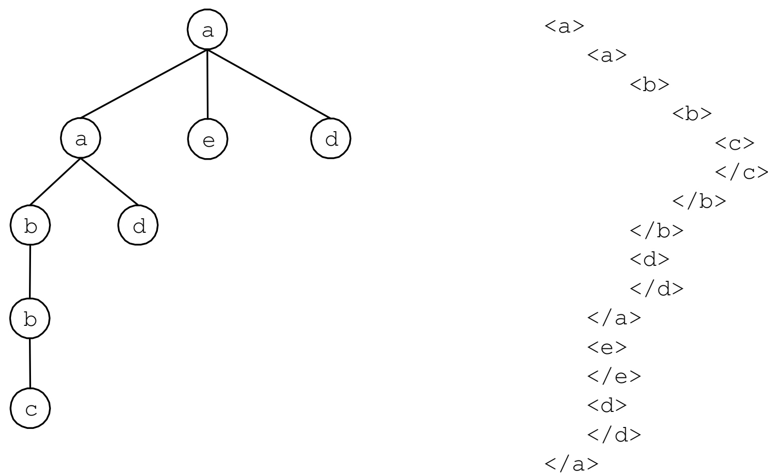


Figure 3.14: The XML Document Used in YFilter* NFA Execution

YFilter* differs from YFilter in that it associates matching instances found to the runtime stack entry during the execution.

When an accepting state of the NFA is encountered, YFilter* does the following:

1. Find all paths p matched at this accepting state.

2. Backtrack runtime stack to find actual matching instances of p.

3. Count the number of "//" nodes materialized in backtracking.

4. Associate matching instance of p to runtime stack entry.

   (a) When $Count_{//}(p, succ)$ number of "//" nodes are counted, decide the stack entry r_1, which is created when $BP(p, succ(p))$ is encountered, by taking the value of $Count_{non-//}(p, succ)$ into account. Associate current matching instance to r_1.

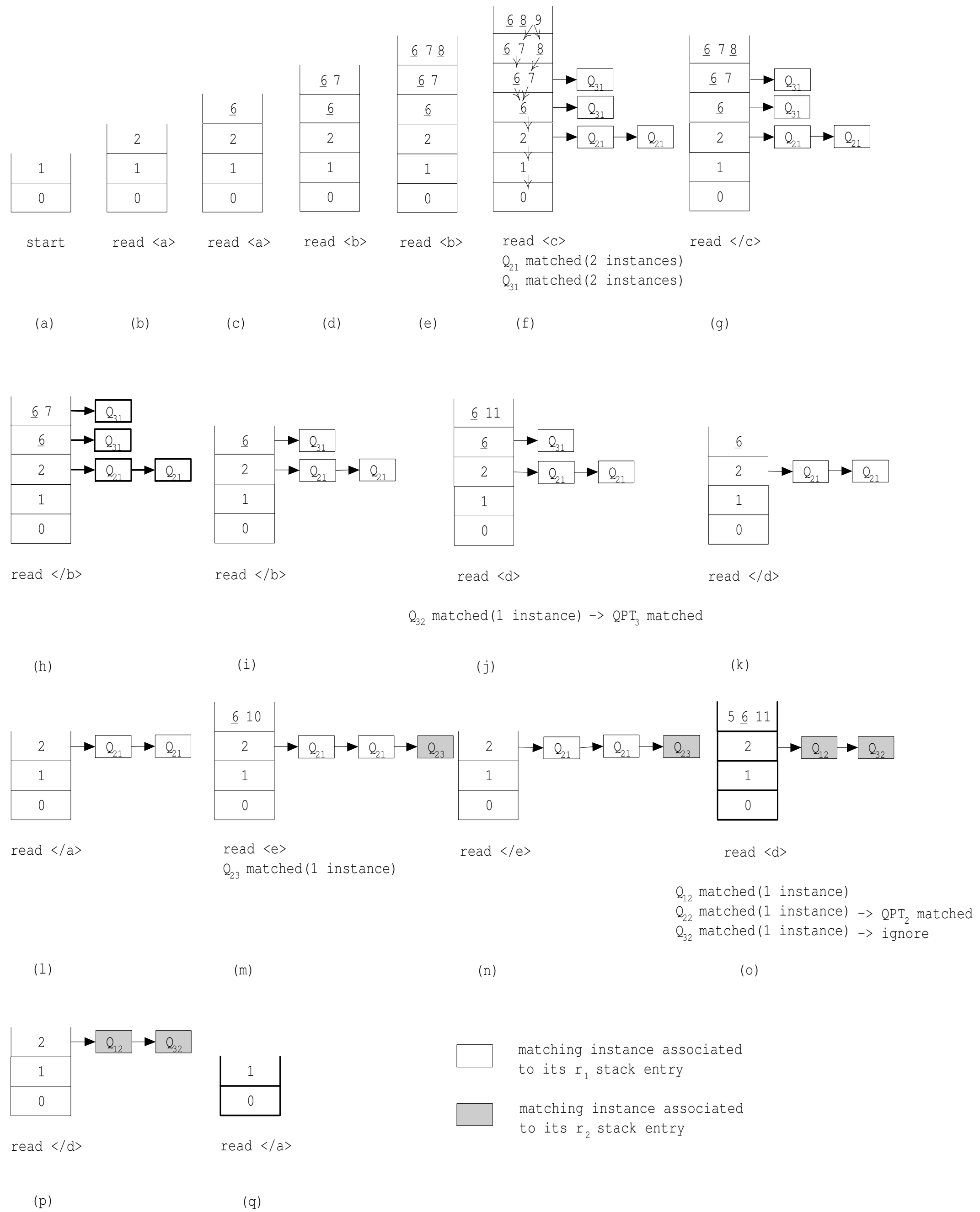


Figure 3.15: Runtime Stack in YFilter* NFA Execution

(b) When $Count_{//}(p, prev)$ number of "//" nodes are counted, decide the stack entry r_2, which is created when $BP(prev(p), p)$ is encountered, by taking the value of $Count_{non-//}(p, prev)$ into account. Associate current matching instance to r_2.

5. Check whether this matching instance of p can be used to update the matching status of its corresponding tree pattern.

A tree pattern is said to be matched when its matching status is updated by its last shredded path.

We explain the execution of NFA in YFilter* through a running example. In Figure 3.14, there is an XML document with its tree pattern format and text format. Figure 3.15 shows the evolution of the runtime stack when YFilter* executes the NFA in Figure 3.13 on the XML document in Figure 3.14.

Backtrack Runtime Stack

When tracking an NFA state in a stack entry backward, we are able to know which element triggers the transition. As a result, we can interpret each path obtained from backtracking to a matching instance.

Example 14 *Consider the runtime stack in Figure 3.15 (f). Q_{2_1} and Q_{3_1} are matched because accepting state 9 is in the top stack entry. Two matching instances are found via backtracking the runtime stack from state 9. One is 9-7-6-6-2-1-0, the other is 9-8-*

7-6-2-1-0. The corresponding matching instances are $/a/*/*/b/c$ *and* $/a/*/b/*/c$ *respectively.* □

Count "//" nodes

In backtracking, a "//" node is observed under the following circumstances:

1. When the transition happens between two states, which have other states with self-loop between them in the NFA;

2. When the transition is from a state without self-loop to another state with self-loop.

Example 15 *In Example 14, the transition 7→9 (reverse of 9-7) is an example of case 1 above, while the transition 2→6 (reverse of 6-2) is an example of case 2. Both of them corresponds to the element next to the end of a "*"-chain. Note that in the case of 7-9, the chain is empty.* □

Associate Matching Instances to Stack Entries

By using the values stored in $Count_{//}(p, succ)$ and $Count_{non-//}(p, succ)$, we are able to locate the stack entry r_1 in the runtime stack, which corresponds to the start of the branch point node with regard to $succ(p)$.

More specifically, while backtracking the matching instance of p in the runtime stack, we count the number of state transitions caused by "//". When the number is up to $Count_{//}(p, succ)$, we stop to locate stack entry r_1 as follows:

1. $Count_{non-//}(p, succ) > 0$, further step back $Count_{non-//}(p, succ) - 1$ entries;

2. $Count_{non-//}(p, succ) = 0$, in this case, the branch point itself is a "//" node. We need to locate, in backtracking, the stack entry of the last node of the materialized branch point in the matching instance. If the materialized branch point is empty, we locate the stack entry just before the empty chain. This stack entry becomes r_1.

Example 16 *In Figure 3.15 (f), firstly, we consider the path 9-7-6-6-2-1-0. When backtracking from 9 to 7, a "//" node is materialized; from 6 to 2, the second "//"node is materialized.*

Since $Count_{//}(Q_{2_1}, succ) = 2$*, there are only two "//" nodes from* $BP(Q_{2_1}, succ(Q_{2_1}))$ *to* Q_{2_1}*'s leaf. In addition, we have* $Count_{non-//}(Q_{2_1}, succ) = 1$*, which suggests there is only one non-"//" node from the BP node to the first "//" node on the path towards* Q_{2_1}*'s leaf. Therefore, when the second "//" node is materialized at stack entry containing state 2, we step* $1 - 1 = 0$ *entry back to locate entry* r_1*. YFilter* associates this matching instance to* r_1*.*

As to Q_{3_1} *in (f), since* $Count_{//}(Q_{3_1}, succ) = 2$*, again, the second "//" node is materialized at stack entry containing state 2. This time we have* $Count_{non-//}(Q_{3_1}, succ) = 0$*, which indicates the the BP node is a "//" node itself. The stack entry containing state 6 and 7 is the last node materialized for the second "//", and it becomes stack entry* r_1*, where an instance of* Q_{3_1} *is associated.*

When we consider the path 9-8-7-6-2-1-0, another matching instance of Q_{2_1} *and* Q_{3_1} *are associated to their corresponding* r_1 *entry.* □

The stack entry r_2 for each matching instance is located using the same techniques above except that, $Count_{//}(p, prev)$ and $Count_{non-//}(p, prev)$ values are used instead of $Count_{//}(p, succ)$ and $Count_{non-//}(p, succ)$.

During the parsing of a document, matching instances found are associated to their corresponding r_1,r_2 stack entries in the runtime stack. In Figure 3.15, those matching instances associated to r_2 stack entries are shadowed.

Both r_1 and r_2 are important to the matching of the whole tree pattern based on the matchings of its shredded paths, which we are going to illustrate in the next section.

Update the Matching State of QPT

If a matching instance of shredded path p has already been found, a matching instance of $succ(p)$ is expected within the valid context of $BP(p, succ(p))$, in other words, before popping out the stack entry r_1. If a matching instance of $succ(p)$ is found before r_1 is popped, then the matching status of the QPT, from which p is shredded, is updated by $succ(p)$. If no matching instance of $succ(p)$ is found before r_1 is popped, then this matching instance of p should be discarded since it cannot be part of a matching instance of p's corresponding QPT in this document.

Example 17 *Consider the* Q_{3_1} *matching instance in Figure 3.15 (f) whose* r_1 *is the stack entry containing only state 6. In (j), when its* r_1 *is still in the stack, a matching instance of its successive path* Q_{3_2} *is found. Thus, the matching state of* QPT_3*, their corresponding QPT, is advanced. Moreover, since* Q_{3_2} *is the last shredded path of* QPT_3*,* QPT_3 *is matched.*

In contrast, consider the Q_{3_1} *matching instance in (f) whose* r_1 *is the stack entry containing state 6 and 7. In (i), its* r_1 *is popped. No matching instance of* Q_{3_2} *will branch at the their BP node in this matching instance of* Q_{3_1}*. Therefore, this matching instance of* Q_{3_1} *is discarded.* □

When parsing an XML document, a matching instance of p might be found before a matching instance of $prev(p)$ is found. In this case, when a matching instance of p is found in the document, by $Count_{//}(p, prev)$ and $Count_{non-//}(p, prev)$, we are able to locate the stack entry r_2 corresponding to the beginning of the branch point relative to $prev(p)$ in the runtime stack. The procedure is the same as that used to locate r_1.

A matching instance of the $prev(p)$ is expected within the valid context of BP $(prev(p), p)$, that is, before popping out the stack entry r_2. If a matching instance of $prev(p)$ is found before r_2 is popped, then the matching status of the QPT, where p is shredded, is updated by p. If no matching instance of $prev(p)$ is found before r_2 is popped, then the this matching instance of p should also be discarded for the same reason given above.

Example 18 *In Figure 3.15 (m), a matching instance of* Q_{2_3} *is found before any matching instance of* Q_{2_2} *is found. So it is associated to its corresponding* r_2 *stack entry, which is the one containing only state 2. In (o), a matching instance of* Q_{2_2} *is found before this* r_2 *is popped, so the matching state of* QPT_2 *jumps to* Q_{2_3}*, which is also the final matching state of* QPT_2*. Thus,* QPT_2 *is matched by this document.* □

YFilter* detects the matching instances of QPTs and discards useless matching instances of shredded paths as early as possible. After a QPT is matched, none of its shredded paths will be considered any more.

Example 19 *In Figure 3.15 (o), while there is a matching instance of* Q_{3_2}*, no effect is made on this matching instance, because its corresponding QPT,* QPT_3 *has already been matched. YFilter* will just ignore this matching instance of* Q_{3_2}*.* □

Chapter 4

Performance Study

We build an XML-based SDI system which uses YFilter* in the filtering stage. We implement the two methods $C \rightarrow A$ and $C + A$ in Java. In addition, we also implement a baseline method called A, which randomly chooses QPTs to aggregate. By comparing the performance of A with that of $C \rightarrow A$ or $C + A$, we show that clustering can help to improve the quality of aggregation.

We carry out experiments to show the effectiveness and scalability of an XML-based SDI system that is augmented with query clustering and aggregation. Various factors that will have influence on the performance of the system, including clustering granularity, diversity of user preferences and QPT distribution, are investigated.

At the end of the section, we also report an experiment studying the performance of YFilter* and YFilter in handling nested-path queries. Our experiment shows that YFilter* outperforms YFilter by a factor of 2.

Pr	Description	Default	Range
N_d	Number of XML documents	100	$100 \sim 1000$
N_q	Number of QPTs	1000	$1000 \sim 4000$
C	Number of subtrees	5	$1 \sim 8$
Z	Parameter of the Zipf distribution of QPTs	0.8	$0.0 \sim 1.0$
S_q	Minimal similarity of QPTs from the same subtree	0.4	-
S_c	Minimal similarity of each result cluster	0.8	$0.0 \sim 1.0$

Table 4.1: Parameters

All of our experiments are conducted on a Pentium IV 1.6 GHz processor with 256MB memory running JVM 1.4.0 on Windows 2000 Professional.

4.1 Experiment Setup

We need two kinds of datasets for our experiments, namely, XML documents and user subscriptions (or QPTs). We use the IBM XML Generator tool [11] to generate XML documents using the *auction* DTD from the XMark benchmark [2]. The *auction* DTD contains a recursive structure that can be nested to produce XML documents with arbitrary number of levels.

When generating QPTs, each rooted subtree of the DTD is a QPT candidate. After

all possible QPTs are enumerated from the DTD, we select QPTs from this pool and repeat them according to certain distribution.

The main advantage of clustering and aggregating user subscriptions is that the common interest shared by a group of users is captured, thus allowing the SDI system to deliver the relevant XML documents to this group of users quickly. QPTs represent the preferences of users (or groups of users), and are likely to be biased towards one aspect of the DTD for users with similar interest. Therefore, by removing the root node of the *auction* DTD, we obtain subtrees each of which represent different user group interests. QPTs are generated based on these subtrees.

The parameter C indicates the number of user group interests, which is translated to the number of subtrees used in the QPT generation. Intuitively, QPTs from users of the same interest (same subtree), would have relatively higher similarity compared to QPTs from users of different interests (different subtrees). We use a parameter S_q to denote the minimal similarity between QPTs from the same subtree.

Since the overlap between different subtrees is small, we can expect the similarity between QPTs from different subtrees to be low. Furthermore, we use the Zipf [31] distribution for the QPT dataset, in which a few QPTs have very high frequency while the rest have very low frequency.

The parameter S_c specifies the clustering granularity. It denotes the minimal similarity within a cluster. In order to measure the quality of the query results, we compare them with the set of actual query results, which can be obtained by executing the original

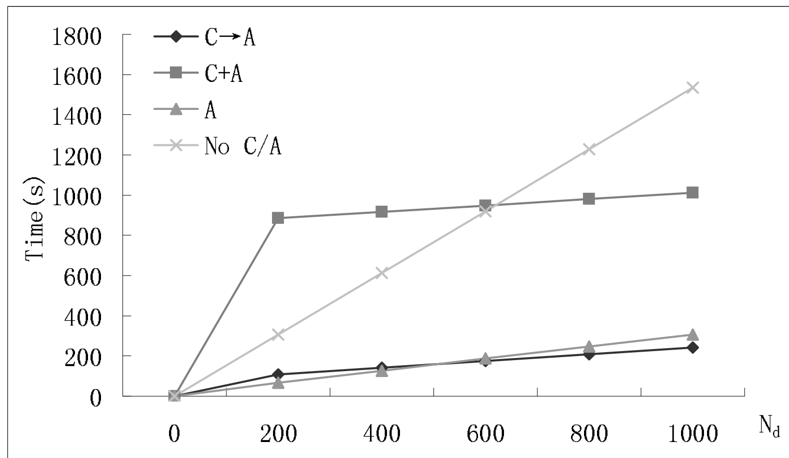


Figure 4.1: Scalability

QPTs individually on the XML dataset.

Table 4.1 summarizes the parameters used in the experiments, together with their default value and range of values tested.

4.2 Scalability

The scalability of our proposed SDI system comes from two aspects. The first is the additional query clustering and aggregation step, and the second is the filtering step. Since YFilter* does not alter the path sharing nature of YFilter, the latter's scalability [10] also applies here.

In this experiment, we examine how the additional step of clustering and aggregating user subscriptions is able to capture the common interest shared by a group of users, thus allowing the SDI system to deliver the relevant XML documents to this group of users

quickly.

Figure 4.1 shows the response time of the XML-based SDI system for 1000 QPTs. Response time here includes both the preprocessing time, i.e. clustering and aggregation time, if any, and the filtering time. Overall, the response times for all methods increases linearly with the number of XML documents. However, the increase is slower when the system uses clustering and/or aggregating techniques.

When no clustering and/or aggregation is used (no C/A), the system has to filter all the XML documents against all the QPTs. As a result, the response time increases rapidly with the increase in the number of XML documents. Although using aggregation alone (A) is able to reduce the number of QPTs against which the documents are filtered, its filtering quality is poor, as we will see from subsequent experiments.

Both $C + A$ and $C \rightarrow A$ scale well when the number of XML documents increases since the documents are filtered against a small number of representative QPTs obtained from the clustering and aggregation process. However, $C \rightarrow A$ outperforms $C + A$ because it utilizes the hierarchical clustering method which has a complexity of $O(n^2)$. In contrast, the $C+A$ approach is very time consuming due to its exhaustive computation of aggregation during the process.

It is clear that the additional time incurred by clustering and/or aggregation is compensated very early. In an SDI system, user subscriptions are relatively stable. Hence, the clustering and aggregation of QPTs can be done offline.

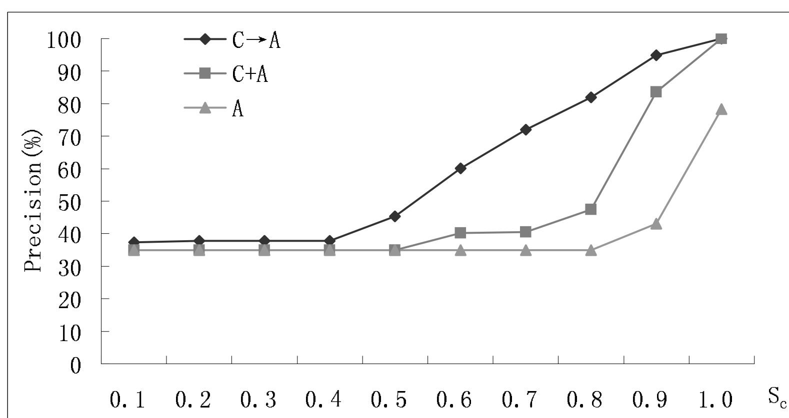


Figure 4.2: Precision vs Cluster Granularity

4.3 Sensitivity Experiments

In this set of experiments, we examine how the performance of the system is affected by the clustering granularity, the diversity of user preferences and the distribution of QPTs. The performance metric used is *precision*, which is the ratio of the documents that are retrieved by the original set of queries over the documents that are retrieved by the set of representative queries.

4.3.1 Clustering Granularity

The clustering granularity determines the number of representative QPTs obtained. It indicates the minimal similarity of each cluster. In this experiment, we examine the effect of varying S_c on the precision and response time for both $C \rightarrow A$ and $C + A$. Figure 4.2 shows the results.

For $C \rightarrow A$, when S_c increases, the QPTs in each result cluster will have higher

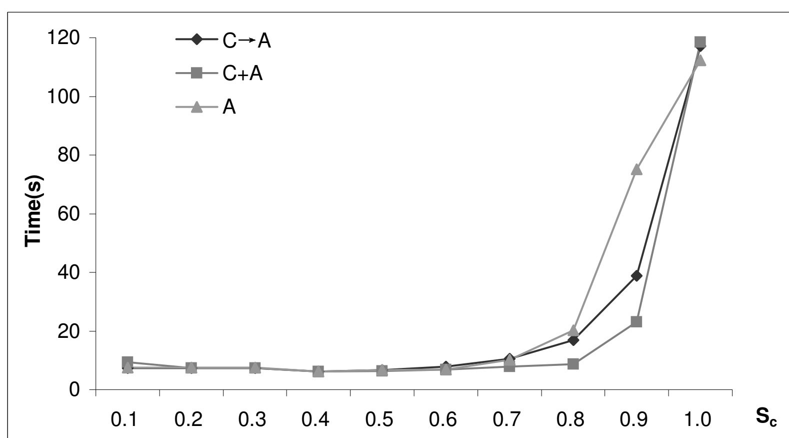


Figure 4.3: Time vs Cluster Granularity

similarity. Aggregation of more similar QPTs are likely to yield more informative aggregation results, which further implies better filtering results. Similarly for $C+A$, each iteration of $C+A$ reduces the total number of clusters by 1. With the increase of S_c for merging clusters, $C+A$ is likely to terminate although many clusters remain. With more clusters, and fewer QPTs of higher similarity within each cluster, we obtain more informative aggregation results, and hence better filtering results.

On closer examination of the results, we note that when S_c is between 0.1 and 0.3, the precision for all three methods are low. When the value of S_c is between 0.4 and 0.8, $C \rightarrow A$ outperforms both $C+A$ and A. When S_c is high ($0.9 \sim 1.0$), both $C \rightarrow A$ and $C+A$ can achieve very high precision. Further, the precision of $C \rightarrow A$ increases gradually with the increase of S_c, while the precision of $C+A$ or A has a sudden increase when S_c reaches 0.9. This may be attributed to the following reason.

When S_c is low ($0.1 \sim 0.3$), there are very few clusters. Recall that for the default setting, the minimal similarity between QPTs from the same subtree is 0.4. Therefore,

when S_c is below 0.4, QPTs from different subtrees are likely to be clustered together. Given that the overlap between different subtrees is small, the aggregation result of such cluster will be too general to be informative.

When S_c is between 0.4 to 0.8, $C \rightarrow A$ performs much better compared to $C + A$ and A because $C \rightarrow A$ is based on the QPTs' original similarity. In contrast, $C + A$ is based on the similarity of the *temporary aggregation results* (aggregation of QPTs that are already clustered), and the QPTs which have not yet been clustered. Since the aggregation result depends only on the actual QPTs involved, i.e. the cluster itself, and not on the sequence of aggregation, the temporary aggregation result is an approximation of all the QPTs already in that cluster. Hence, the similarity computation will be less accurate compared to that in $C \rightarrow A$. However, $C + A$ still performs better than A, which has no clustering at all.

It turns out that when S_c increases to 0.9, the number of result clusters in addition to the quality of aggregation starts to dominate the precision. In fact, in $C + A$, the number of result clusters increases six folds when S_c increases from 0.8 to 0.9, while the number of result clusters only doubles in $C \rightarrow A$. This also shows that $C \rightarrow A$ has a more stable performance compared to $C + A$.

Clearly, there is a trade-off between the clustering granularity and system performance. There are more clusters when the clustering granularity increases, which leads to an increase in the response time.

In the following experiments, we set S_c in the range of [0.4, 0.8] so that the quality of

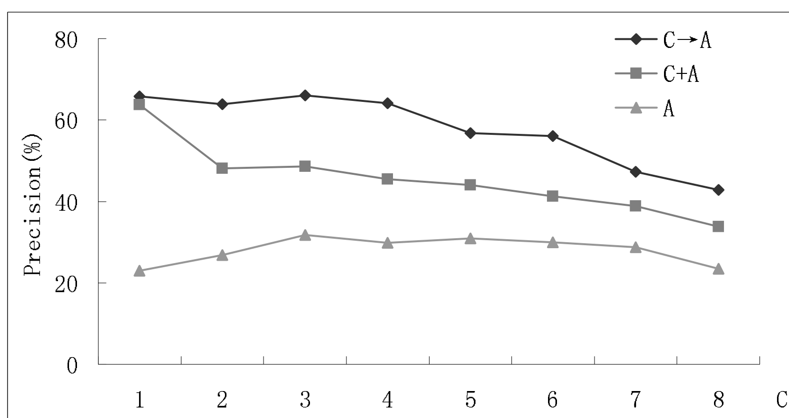


Figure 4.4: Query Diversity

aggregation dominates the performance of the system and the number of representative QPTs generated is reasonable. For example, when $S_c = 0.7$, the number of representative QPTs generated by $C \rightarrow A$ is almost one tenth of the original number of QPTs.

4.3.2 Diversity of User Preference

The number of subtrees used to generate the QPTs determines how diverse the user preferences are. In this experiment, we vary the number of subtrees (C) to study the influence of user preference on the system precision. In order to have a stable filtering time, we fix the number of result clusters at 50.

Figure 4.4 shows that the precision for all the three methods decreases when C increases. This is because when the user preference becomes increasingly diverse, the number of QPTs within each cluster is reduced, and QPTs from different subtrees may be aggregated, leading to a more general representative query, and hence lower precision.

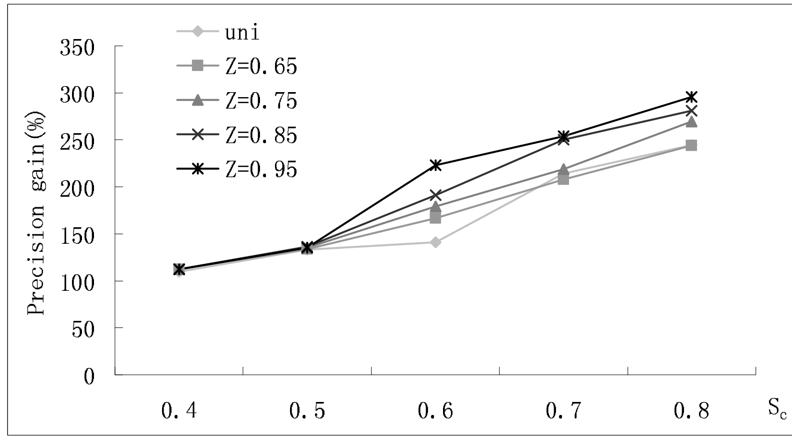


Figure 4.5: QPT Distribution

4.3.3 Distribution of QPT

Next, we examine how the distribution of QPTs affects the performance. We use both uniform and Zipf distributions to generate QPTs. The Zipf parameter Z determines the skewness of the query distribution. In order to show the improvement in the performance of methods involving clustering, we compute the *precision gain* of $C \rightarrow A$ and $C + A$ over A.

$$Precision\ Gain = \frac{(C \rightarrow A)'s\ or\ (C+A)'s\ Precision}{A's\ Precision} \times 100\% \tag{4.1}$$

We observe from Figure 4.5 that the precision gain increases when the distribution of QPT becomes more skewed. This is because there is less distinct QPTs, leading to more informative aggregation results. We also observe that there is no difference in precision gain when S_c is very low, because at the stage, the effect of skewness in QPT has been overwhelmed by the generality of aggregation.

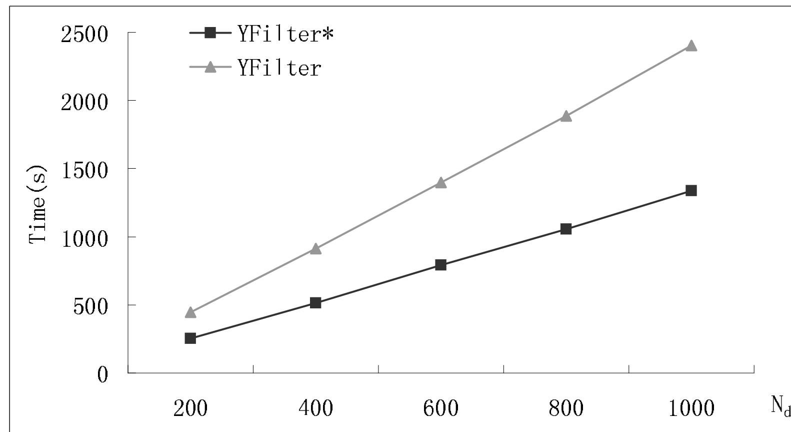


Figure 4.6: YFilter* vs YFilter

4.4 YFilter* versus YFilter

We designed YFilter* based on YFilter in order to handle nested path queries efficiently. This experiment is going to compare their performances. When implementing YFilter, we use the post-processing technique described in [10], to handle nested path queries. Figure 4.6 shows that, when processing queries with nested paths, YFilter* outperforms YFilter by a factor of 2 on average.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Motivated by the overwhelming number of document-subscription matchings required in XML-based SDI systems, in this work, we have studied how clustering and aggregating user queries can help increase the scalability of SDI systems by reducing the number of document-subscription matchings needed.

We have designed an *aggregation similarity* function for clustering tree patterns involving wildcards and relative paths.

Two methods, namely $C \rightarrow A$ and $C + A$, have been proposed to integrate the clustering and aggregation of user queries.

We have made improvements on YFilter to develop YFilter*, which enhances YFilter's ability to handle tree-structured XML queries.

Experiment results have indicated that the proposed techniques are able to decrease the response time of SDI systems and achieve 100% recall with 20% to 30% precision loss. Extensive experiments have been carried out to study the factors influencing the performance of the system.

5.2 Future Work

So far, we have made the assumption that the user subscriptions in the SDI system are static and do not change with time. One possible direction for future work is to consider the situation when there are updates, i.e. insertion and/or deletion, of user subscriptions in the system. The update of user subscriptions will lead to the update of query clusters and tree aggregation results, which has further influence on the performance of the system.

Bibliography

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of 26th International Conference on Very Large Data Bases(VLDB)*, pages 53–64, Cairo, Egypt, September 2000.

[2] R. Busse and M. Carey. Benchmark DTD for XMark, an XML Benchmark project. http://monetdb.cwi.nl/xml/downloads/downloads.html, 2002.

[3] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of 28th International Conference on Very Large Data Bases(VLDB)*, 2002.

[4] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of 18th International Conference on Data Engineering(ICDE)*, 2002.

[5] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[6] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases*, pages 90–101, Edinburgh, Scotland, U.K., Sept. 1999.

[7] S. S. Chawathe. Comparing hierarchical data in external memory. In *Proceedings of the Twenty-fifth International Conference on Very Large Data Bases (VLDB)*, pages 90–101, Edinburgh, Scotland, U.K., 1999.

[8] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montréal, Québec, June 1996.

[9] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *ICDE*, 2002.

[10] Y. Diao, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *Transactions on Database Systems(TODS)*, December 2003.

[11] A. L. Diaz and D. Lovell. XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, September 1999.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison Wesley Press, Cambridge, Massachusetts, 2 edition, 2001.

[13] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: clustering XML schemas for effective integration. In *Proceedings of the eleventh international conference on Information and knowledge management*, pages 292–299. ACM Press, 2002.

[14] D. Megginson. SAX: A Simple API for XML. http://www.megginson.com/SAX.

[15] A. Nierman and H. V. Jagadish. Evaluating structural similarity in XML documents. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, USA, 2002.

[16] L. Opyrchal, M. Astley, J. S. Auerbach, G. Banavar, R. E. Strom, and D. C. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *Proceedings of International Conference on Distributed Systems Platforms (Middleware)*, pages 185–207, 2000.

[17] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, 1977.

[18] D. Shasha and K. Zhang. Approximate pattern matching algorithms. In *Pattern Matching in Strings, Trees and Arrays, chapter 14*. 1995.

[19] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26:422–433, 1979.

[20] W3C. Document Object Model(DOM) Level 1 Specification (Second Edition), Version 1.0. http://www.w3.org/TR/REC-DOM-Level-1/, October 1998.

[21] W3C. XML-QL: A Query Language for XML. http://www.w3.org/TR/NOTE-xml-ql, August 1998.

[22] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, November 1999.

[23] W3C. XML Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, November 1999.

[24] W3C. eXtensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml, October 2000.

[25] W3C. XML Pointer Language (XPointer). http://www.w3.org/TR/xptr, August 2002.

[26] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, May 2003.

[27] J. T.-L. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(4):559–571, 1994.

[28] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the Boolean model. *Transactions on Database Systems(TODS)*, 19(2):332–334, 1994.

[29] L. H. Yang, M. L. Lee, W. Hsu, and S. Acharya. Mining frequent query patterns from XML queries. In *Eighth International Conference on Database Systems for*

Advanced Applications (DASFAA '03), March 26-28, 2003, Kyoto, Japan. IEEE Computer Society, 2003.

[30] K. Zhang and D. Shasha. Simple fast algorithm for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.

[31] G. K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison Wesley Press, Cambridge, Massachusetts, 1949.