Tree Pattern Aggregation for Scalable XML Data Dissemination

Chee-Yong Chan, Wenfei Fan,[*] Pascal Felber,[†] Minos Garofalakis, Rajeev Rastogi
Bell Labs, Lucent Technologies
{cychan,wenfei,minos,rastogi}@research.bell-labs.com, Pascal.Felber@eurecom.fr

Abstract

With the rapid growth of XML-document traffic on the Internet, scalable content-based dissemination of XML documents to a large, dynamic group of consumers has become an important research challenge. To indicate the type of content that they are interested in, data consumers typically specify their subscriptions using some XML pattern specification language (e.g., XPath). Given the large volume of subscribers, system scalability and efficiency mandate the ability to *aggregate* the set of consumer subscriptions to a smaller set of content specifications, so as to both reduce their storage-space requirements as well as speed up the document-subscription matching process. In this paper, we provide the first systematic study of subscription aggregation where subscriptions are specified with *tree patterns* (an important subclass of XPath expressions). The main challenge is to aggregate an input set of tree patterns into a smaller set of generalized tree patterns such that: (1) a given *space constraint* on the total size of the subscriptions is met, and (2) the *loss in precision* (due to aggregation) during document filtering is minimized. We propose an efficient tree-pattern aggregation algorithm that makes effective use of document-distribution statistics in order to compute a *precise* set of aggregate tree patterns within the allotted space budget. As part of our solution, we also develop several novel algorithms for tree-pattern containment and minimization, as well as "least-upper-bound" computation for a set of tree patterns. These results are of interest in their own right, and can prove useful in other domains, such as XML query optimization. Extensive results from a prototype implementation validate our approach.

1 Introduction

XML (eXtensible Markup Language) [16] has become the dominant standard for data encoding and exchange on the Internet, including e-Business transactions in both Business-to-Business (B2B) and Business-to-Consumer (B2C) applications. Given the rapid growth of XML traffic on the Internet, the effective and efficient delivery of XML documents has become an important issue. Consequently, there is growing interest in the area of XML *content-based filtering and routing* (e.g., [4]), which addresses the problem of effectively directing high volumes of XML-document traffic to interested consumers based on document *contents*. Unlike conventional routing, where packets are routed based on a limited, fixed set of attributes (e.g., source/destination IP addresses and port numbers), content-based routing is based on general patterns of the document contents, which is significantly more flexible and demanding. Consumers typically specify their *subscriptions*, indicating the type of XML content that they are interested in, using some XML pattern specification language (e.g., XPath [15]). For each incoming XML document, a *content-based router* matches the document contents against the set of subscriptions to identify the (sub)set of interested consumers, and then routes the document to them. Thus, in content-based routing, the "destination" of an XML document is generally unknown to the data producer, and is computed *dynamically* based on the document contents and the active set of subscriptions.

Effective support for scalable, content-based XML routing is crucial to enabling efficient and timely delivery of relevant XML documents to a large, dynamic group of consumers. Given the large volume of potential consumers, system scalability and efficiency madate the ability to judiciously *aggregate* the set of consumer subscriptions to a smaller set of content specifications. The goal, of course, is to both reduce the subscriptions' storage space requirements (e.g., so that the routing table fits in main memory), as well as speed up the filtering of incoming XML traffic. For instance, a core router in a B2B application may choose to aggregate subscriptions based on geographical location, affiliation, or domain-specific information (e.g., telecommunications). Subscription 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. However, since there is typically a *"loss of precision"* associated with such aggregation, the documents matched by the aggregated set A is, in general, a superset of those matched by the original set S. As a result, a document may be routed to consumers who have not subscribed to it, thus resulting in an increase in the amount of unwanted

[*]Currently on leave from Temple University and supported in part by NSF Career Award IIS-0093168.

[†]Current affiliation: Institut EURECOM, Sophia Antipolis, France

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

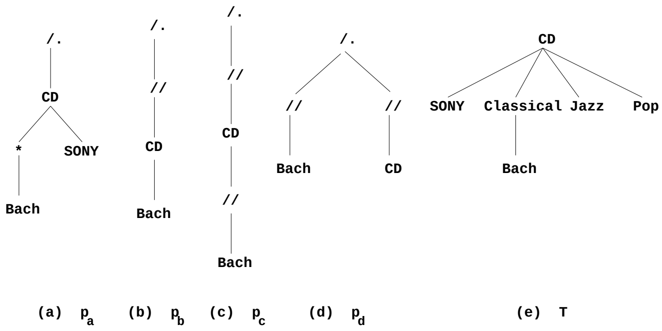


Figure 1: Example Tree Patterns and XML Document Tree.

document traffic. In order to avoid such spurious forwarding of documents, it is desirable to minimize the number of such "false matches" (i.e., minimize the loss in precision) with respect to the given space constraint for the aggregated subscriptions.

So far, there has only been limited work on subscription aggregation, mainly for very simple subscription models. For example, in [12], each subscription is a set of attribute-predicate pairs (e.g., $\{issue =$ "GE", $price < 120, volume > 1000\}$), and an aggregated subscription is allowed to contain wildcard values, indicating the entire set of domain values for certain attributes. [1] In this paper, we provide the first systematic study of the subscription aggregation problem where subscriptions are specified using the much more expressive model of *tree patterns*. Tree patterns represent an important subclass of XPath expressions that offers a natural means for specifying tree-structured constraints in XML and LDAP applications [3]. Compared to earlier work based on attribute/predicate-based subscriptions, effectively aggregating tree-patterns poses a much more challenging problem since subscriptions involve both content information (node labels) as well as structure information (parent-child and ancestor-descendant relationships). Briefly, our *tree pattern aggregation problem* can be stated as follows: Given an input set of tree patterns S and a space constraint, aggregate S into a smaller set of generalized tree patterns that meets the space constraint, and for which the loss in precision due to aggregation is minimized.

Example 1.1 *Consider the two similar tree-pattern-based subscriptions p_a and p_b shown in Figure 1, where p_a matches any document with a root element labeled "CD" that has both a sub-element labeled "SONY" as well as a sub-element (with an arbitrary label) that in turn has a sub-element labeled "Bach"; and p_b matches any document that has some element labeled "CD" with a sub-element labeled "Bach". Here the node labeled '$*$' (wildcard) matches any label, while the node labeled '//' (descendant) matches some (possibly empty) path. The XML document T shown in Figure 1(e) matches (or satisfies) p_a but not p_b because the sub-element labeled "Bach" in T does not have a parent element labeled "CD". For efficiency reasons, one might want to aggregate the set of tree patterns $\{p_a, p_b\}$ into a single tree pattern. Two examples of aggregate tree patterns for $\{p_a, p_b\}$ are p_c and p_d (in Figure 1) since any document that satisfies p_a or p_b also satisfies both p_c and p_d. Although both p_c and p_d have the same number of nodes, p_c is intuitively "more precise" than p_d with respect to $\{p_a, p_b\}$ since p_c preserves the ancestor-descendant relationship between the "CD" and "Bach" elements as required by p_a and p_b. Indeed, any XML document that satisfies p_c also satisfies p_d (and thus we say that p_d "contains" p_c).* $\square$

To the best of our knowledge, our work is the first to address this timely subscription aggregation problem for XML data dissemination. Our main contributions can be summarized as follows.

- We study the properties of tree patterns and develop efficient algorithms for deciding tree pattern containment, minimizing a tree pattern, and computing the most precise aggregate (i.e., the "least upper bound") for a set of patterns. Our results are not only interesting in their own right, but also provide solutions for special cases of our tree pattern aggregation problem.
- We propose a novel, efficient method that exploits coarse statistics on the underlying distribution of XML documents to compute a "precise" set of aggregate patterns within the allotted space budget. Specifically, our scheme employs the document statistics to estimate the *selectivity* of a tree pattern, which is also used as a measure of the pattern's preciseness. Thus, our aggregation problem reduces to that of finding a compact set of aggregate patterns with minimal loss in selectivity, for which we present a greedy heuristic.
- We demonstrate experimentally the effectiveness of our approach in computing a space-efficient and precise set of aggregate tree patterns.

The usefulness of our results on tree patterns and their aggregation is not limited to content-based routing, but also extends to other application domains such as the optimization of XML queries involving tree patterns and the processing/dissemination of subscription queries in a multicast environment [9] (where aggregation can be used to reduce server load and network traffic). Further, our work and results are complementary to recent work on efficient indexing structures for XPath expressions [2, 6]. The focus of this earlier research is to speed up document filtering with a given set of XPath subscriptions using appropriate indexing schemes. In contrast, our work focuses on *effectively reducing the volume of subscriptions* that need to be matched in order to ensure scalability given bounded storage resources for routing. Clearly, our techniques can be used as a pre-processing step for the indexes of [2, 6] when hard constraints on the size of the index must be met. Due to space limitations, the proofs of all theoretical results can be found in the full version of this paper [5].

[1] Due to space constraints, a more detailed overview of related work can be found in the appendix.

2 Problem Formulation

2.1 Definitions

A *tree pattern* is an unordered node-labeled tree that specifies content and structure conditions on an XML document. More specifically, a tree pattern p has a set of nodes, denoted by $Nodes(p)$, where each node v in $Nodes(p)$ has a label, denoted by $label(v)$, which can either be a tag name, a "$*$" (wildcard that matches any tag), or a "$//$" (the descendant operator). In particular, the root node has a special label "$/.$". We use $Subtree(v, p)$ to denote the subtree of p rooted at v, referred to as a *sub-pattern* of p. Some examples of tree patterns are depicted in Figure 2.

To define the semantics of a tree pattern p, we first give the semantics of a sub-pattern $Subtree(v, p)$, where v is not the root node of p. Recall that XML documents are typically represented as node-labeled trees, referred to as *XML trees*. Let T be an XML tree and t be a node in T. We say that T *satisfies* $Subtree(v, p)$ at node t, denoted by $(T, t) \models Subtree(v, p)$, if the following conditions hold: (1) if $label(v)$ is a tag, then t has a child node t' labeled $label(v)$ such that for each child node v' of v, $(T, t') \models Subtree(v', p)$; (2) if $label(v) = *$, then t has a child node t' labeled with an arbitrary tag such that for each child node v' of v, $(T, t') \models Subtree(v', p)$; and (3) if $label(v) = //$, then t has a descendant node t' (possibly $t' = t$) such that for each child v' of v, $(T, t') \models Subtree(v', p)$.

We next define the semantics of tree patterns. Let T be an XML tree with root t_{root}, and p be a tree pattern with root v_{root}. We say that T *satisfies* p, denoted by $T \models p$, iff for each child node v of v_{root}, (1) if $label(v)$ is a tag a, then t_{root} is labeled with a and for each child node v' of v, $(T, t_{root}) \models Subtree(v', p)$ (here $label(v)$ specifies the tag of t_{root}); (2) if $label(v) = *$, then t_{root} may have any label and for each child node v' of v, $(T, t_{root}) \models Subtree(v', p)$; (3) if $label(v) = //$, then t_{root} has a descendant node t' (possibly $t' = t_{root}$) such that $T' \models p'$, where T' is the subtree rooted at t', and p' is identical to $Subtree(v, p)$ except that "/." is the label for the root node v (instead of $label(v)$). Observe that v_{root} is treated differently from the rest of the nodes of p. The motivation behind this is illustrated by p_i in Figure 2, which specifies the following: for any XML tree T satisfying p_i, its root must be labeled with a and moreover, it must contain two consecutive a elements somewhere. This cannot be expressed without our special root label "/." (as tree patterns do not allow a union operator).

Example 2.1 *Consider the tree pattern p_a in Figure 2. An XML document T satisfies p_a if its root element satisfies all the following conditions: (1) its label is* `a`*; (2) it must have a child element with an arbitrary tag, which in turn has a child element with a label* `b`*; and (3) it must have a descendant element which has both a* `c`*-child element and an* `a`*-child element. Thus, p_a essentially specifies (existential) conjunctive conditions on XML documents. It should be noted that documents satisfying p_a may have tags/subtrees not mentioned in p_a. For instance, the root element of T may have a* `d`*-child element, and the* `b`*-elements of T may have* `c`*-descendant elements.* □

A tree pattern p is said to be *consistent* if and only if there exists an XML document that satisfies p. We only consider consistent tree patterns in our work. Further, the tree patterns defined above can be naturally generalized to accommodate simple conditions and predicates (e.g., $issue$ = "GE" and $price < 1000$). To simplify the discussion, we do not consider such extensions in this paper.

It is worth mentioning that a tree pattern can be easily converted to an equivalent XPath expression [15] in which each sub-pattern is expressed as a condition/qualifier [5]. Thus, our tree patterns are graph representations of a class of XPath expressions, which are similar to the tree patterns that have been studied for XML queries (e.g., [3, 17]). It is tempting to consider using a larger fragment of XPath to express subscription patterns. However, it turns out that even a mild generalization of our tree patterns (e.g., with the addition of union/disjunction operators) leads to a much higher complexity (coNP-hard or beyond) for basic operations such as containment computation (e.g., see [10]).

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 then T also satisfies p. If $q \sqsubseteq p$, we refer to p as the *container pattern* and q as the *contained pattern*. We say that p and q are *equivalent*, denoted by $p \equiv q$, if $p \sqsubseteq q$ and $q \sqsubseteq p$. This definition can be generalized to *sets* of tree patterns: a set of tree patterns S is *contained* in another set of tree patterns S', denoted by $S \sqsubseteq S'$, if for each $p \in S$, there exists $p' \in S'$ such that $p \sqsubseteq p'$. Containment for sub-patterns is defined similarly.

The *size* of a tree pattern p, denoted by $|p|$, is simply the cardinality of its node set. For example, referring to Figure 2, $|p_a| = 7$ and $|p_b| = 8$.

2.2 Problem Statement

The *tree pattern aggregation problem* that we investigate in this paper can now be stated as follows. Given a set of tree pattern subscriptions S and a space bound k on the total size of the aggregated subscriptions, compute a set of tree patterns S' that satisfies all of the following three conditions:

(C1) $S \sqsubseteq S'$ (i.e., S' is at least as general as S),

(C2) $\sum_{p' \in S'} |p'| \leq k$ (i.e., S' is "concise"), and

(C3) S' is as "precise" as possible, in the sense that there does not exist another set of tree patterns S'' that satisfies the first two conditions and $S'' \sqsubseteq S'$.

Clearly, the tree pattern aggregation problem may not necessarily have a unique solution since it is possible to have two sets S' and S'' that satisfy the first two conditions but $S' \not\sqsubseteq S''$ and $S'' \not\sqsubseteq S'$. Therefore, we need to devise some measure to quantify the goodness of candidate solutions in terms of both their conciseness as well as preciseness.

With respect to conciseness, we are interested in *minimal* tree patterns that do not contain any "redundant" nodes. More precisely, we say that a tree pattern p is *minimized* if for any tree pattern p' such that $p' \equiv p$, it is the case that $|p| \leq |p'|$. With respect to preciseness, it can be

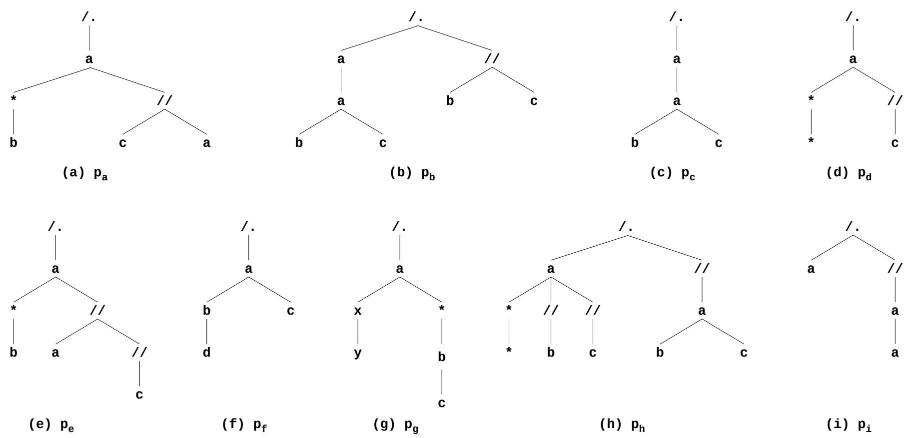


Figure 2: Examples of Tree Patterns.

shown that the containment relationship $\sqsubseteq$ on the universe of tree patterns actually defines a *lattice*. In particular, the notions of *upper bound* and *least upper bound* are of relevance to the aggregation problem and, therefore, we define them formally here.

An *upper bound* of two tree patterns p and q is a tree pattern u such that $p \sqsubseteq u$ and $q \sqsubseteq u$, i.e., for any XML tree T, if $T \models p$ or $T \models q$ then $T \models u$. 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, $u \sqsubseteq u'$. Once again, we generalize the notion of LUBs to a set S of tree patterns. An *upper bound* of S is a tree pattern U, denoted by $S \sqsubseteq U$, such that $p \sqsubseteq U$ for every $p \in S$. The LUB of S, denoted by $\sqcup S$, is an upper bound U of S such that for any upper bound U' of S, $U \sqsubseteq U'$.

Clearly, if p is *an aggregate tree pattern* for a set of tree patterns S (i.e., $S \sqsubseteq p$), then p is an upper bound of S. Observe that, if p is the LUB of S, then p is the *most precise* aggregate tree pattern for S. In fact, it can be shown that $\sqcup S$ exists and is unique up to equivalence for any set S of tree patterns [5]; thus, it is meaningful to talk about $\sqcup S$ as the most precise aggregate tree pattern.

Example 2.2 *Consider again the tree patterns in Figure 2. Observe that $p_b \equiv p_c$; and since $|p_b| > |p_c|$, p_b is not a minimized pattern. In fact, except for p_b, all the tree patterns in Figure 2 are minimized patterns. Note that $p_a \not\sqsubseteq p_c$ because the root node of p_a does not have a tag-a child node; and $p_c \not\sqsubseteq p_a$ because there exists no node in p_c that is a parent node of both a tag-a-node and a tag-c-node. Observe that $p_a \sqsubseteq p_d$ and $p_c \sqsubseteq p_d$; i.e., p_d is an upper bound of p_a and p_c. However, $p_d \not\equiv p_a \sqcup p_c$ since we have another tree pattern, p_e, which is an upper bound of p_a and p_c such that $p_e \sqsubseteq p_d$. Indeed, $p_e = p_a \sqcup p_c$ with $|p_e| < |p_a| + |p_c|$. Note, however, that the size of an LUB is not necessarily always smaller than the size of its constituent patterns. For example, $p_h = p_c \sqcup p_f$ but $|p_h| > |p_c| + |p_f|$. Note that p_d is an upper bound of $\{p_a, p_b, p_c, p_e, p_f, p_g, p_h\}$.* $\square$

We conclude this section by presenting some additional notation used in this paper. For a node v in a tree pattern p, we denote the set of child nodes of v in p by $Child(v, p)$. We also define a partial ordering $\preceq$ on node labels such that if x and x' are tag names, then (1) $x \preceq * \preceq //$ and (2) $x \preceq x'$ iff $x = x'$. Given two nodes v and w, $MaxLabel(v, w)$ is defined to be the "least upper bound" of their labels $label(v)$ and $label(w)$ as follows:

$$MaxLabel(v,w) = \begin{cases} label(v) & \text{if } label(v) = label(w), \\ // & \text{if } (label(v) = //) \\ & \text{or } (label(w) = //), \\ * & \text{otherwise.} \end{cases}$$

For example, $MaxLabel(a, b) = *$ and $MaxLabel(*, //) = //$. For notational convenience, we refer to a node v in a tree pattern as an *ℓ-node* if $label(v) = \ell$, and refer to v as a *tag-node* if $label(v) \notin \{/., *, //\}$.

3 Computing the Most Precise Aggregate

In this section, we consider a special case of our tree pattern aggregation problem, namely, when the aggregate set S' consists of a single tree pattern and there is no space constraint. For this case, we provide an algorithm to compute the *most precise* aggregate tree pattern (i.e., LUB) for a set of tree patterns. Some of the algorithms given in this section are also key components of our solution for the general problem, which is presented in the next section.

Given two input tree patterns p and q, Algorithm LUB in Figure 3 computes the most precise aggregate tree pattern for $\{p, q\}$ (i.e., the LUB of p and q). It traverses p and q top-down and computes the *tightest container sub-patterns* for each pair of sub-patterns $p' = Subtree(v, p)$ and $q' = Subtree(w, q)$ encountered, where v and w are nodes in p and q, respectively. The tightest container sub-patterns of p' and q' are a set R of sub-patterns such that:

(1) R consists of container sub-patterns[2] of p' and q', i.e., for any XML document T and any element t in T, if $(T, t) \models p'$ or $(T, t) \models q'$ then $(T, t) \models r$ for each $r \in R$; and,

[2]Note that a sub-pattern of tree patterns p and q is an upper-bound of p and q, and we use these two terms interchangeably.

Algorithm LUB (p, q)
Input: p and q are tree patterns.
Output: A tree pattern representing the LUB of p and q.
1) **if** $(q \sqsubseteq p)$ **then** return p;
2) **if** $(p \sqsubseteq q)$ **then** return q;
3) Initialize $TCSubPat[v, w] = \emptyset$,
$\forall\ v \in Nodes(p),\ \forall\ w \in Nodes(q)$;
4) Let v_{root} and w_{root} denote the root nodes of p and q, resp.;
5) **for** each $v \in Child(v_{root}, p)$ **do**
6) **for** each $w \in Child(w_{root}, q)$ **do**
7) $TCSubPat[v, w] =$ LUB_SUB $(v, w, TCSubPat)$;
8) Create a tree pattern x with root node label /. and
the set of child sub-patterns
$\bigcup_{v \in Child(v_{root}, p), w \in Child(w_{root}, q)} TCSubPat[v, w]$;
9) **return** MINIMIZE (x);

Algorithm LUB_SUB $(v, w, TCSubPat)$
Input: v, w are nodes in tree patterns p, q (respectively),
$TCSubPat$ is a 2-dimensional array such that
$TCSubPat[v, w]$ is the set of tightest container
sub-patterns of $Subtree(v, p)$ and $Subtree(w, q)$.
Output: $TCSubPat[v, w]$.
1) **if** $(TCSubPat[v, w] \neq \emptyset)$ **then**
2) **return** $TCSubPat[v, w]$;
3) **else if** $(Subtree(w, q) \sqsubseteq Subtree(v, p))$ **then**
4) **return** $\{Subtree(v, p)\}$;
5) **else if** $(Subtree(v, p) \sqsubseteq Subtree(w, q))$ **then**
6) **return** $\{Subtree(w, q)\}$;
7) **else**
8) Initialize $R = \emptyset$; $R' = \emptyset$; $R'' = \emptyset$;
9) **for** each $v' \in Child(v, p)$ **do**
10) **for** each $w' \in Child(w, q)$ **do**
11) $R = R \cup$ LUB_SUB $(v', w', TCSubPat)$;
12) **for** each $v' \in Child(v, p)$ **do**
13) $R' = R' \cup$ LUB_SUB $(v', w, TCSubPat)$;
14) **for** each $w' \in Child(w, q)$ **do**
15) $R'' = R'' \cup$ LUB_SUB $(v, w', TCSubPat)$;
16) Let x be the pattern with root node label $MaxLabel(v, w)$
and set of child subtree patterns R;
17) Let x' be the pattern with root node label //
and set of child subtree patterns R';
18) Let x'' be the pattern with root node label //
and set of child subtree patterns R'';
19) **return** $TCSubPat[v, w] = \{x, x', x''\}$;

Figure 3: Least-Upper-Bound Computation Algorithm.

(2) R is tightest in the sense that for any other set of container sub-patterns R' of p' and q' that satisfies condition (1), any XML document T and any element t in T, if $(T, t) \models r$ for each $r \in R$ then $(T, t) \models r'$ for all $r' \in R'$.

Intuitively, R is a collection of conditions imposed by both p' and q' such that if T satisfies p' or q' at t, then T also satisfies the conjunction of these conditions at t. We now show how the LUB for p and q can be computed from the tightest container sub-patterns. Let v_{root} and w_{root} be the roots of patterns p and q, respectively. Note that a document T that satisfies p also satisfies, for each $v \in Child(v_{root}, p)$, the restriction of p to the root node and only $Subtree(v, p)$. Consequently, a document T that satisfies p or q must also satisfy the pattern x consisting of a root node (with label /.) whose children are the tightest container sub-patterns for each pair $Subtree(v, p)$ and $Subtree(w, q)$, where $v \in Child(v_{root}, p)$ and $w \in Child(w_{root}, q)$. This pattern x is thus an LUB of p and q.

The main subroutine in our LUB computation (Algorithm LUB_SUB) computes the tightest container sub-patterns of p' and q' as follows. If $q' \sqsubseteq p'$ (resp. $p' \sqsubseteq q'$), then p' (resp. q') is the tightest container sub-pattern; otherwise, the tightest container sub-patterns are a set $\{x, x', x''\}$ of sub-patterns, which are defined in the following manner. The root node of x is labeled with $MaxLabel(v, w)$ and the child subtrees of x are the tightest container sub-patterns of each child subtree of p' and each child subtree of q'. Intuitively, the root of x corresponds to the roots of p' and q' (with a label equal to the least upper bound of that of p' and q'). In other words, x preserves the positions of the corresponding nodes in p' and q'. However, this "position-preserving" generalization is not sufficient since p' and q' may have common sub-patterns at different positions relative to their roots. For example, p_c and p_f in Figure 2 have a common sub-pattern rooted at an a-node that has both a b-child and a c-child, but this pattern is located at different positions relative to the roots of p_c and p_f. To capture these "off-position" common sub-patterns, we need to compute x' and x''. The child subtrees of x' are the tightest container sub-patterns of q' itself and each child subtree of p'; and the label of the root node of x' is // to accommodate common sub-patterns at different positions relative to the roots of p' and q'. Similarly, the root node of x'' has label //, and the child subtrees of x'' are the tightest container sub-patterns of p' itself and each child subtree of q'.

By computing the tightest container sub-patterns recursively, the algorithm computes the LUB of the input tree patterns p and q. By induction on the structures of p and q, we can show the following result [5].

Proposition 3.1: *Given two tree patterns p and q, Algorithm* LUB (p, q) *computes $p \sqcup q$.* ☐

Example 3.1 *Given p_c and p_f in Figure 2, Algorithm* LUB *returns p_h, which is indeed $p_c \sqcup p_f$. To help explain the computation of p_h, we use the notation x_n to refer the n^{th} node (in some tree pattern) that is labeled "x", where each collection of nodes sharing the same label are ordered based on their pre-order sequence; for example, in p_h, we use $//_1$ and $//_3$ to refer to the leftmost and rightmost //-nodes, respectively. Algorithm* LUB_SUB *(invoked by Algorithm* LUB*) first extracts the "position preserving" tightest container sub-patterns for $Subtree(a_1, p_c)$ and $Subtree(a, p_f)$, which yields the sub-pattern $Subtree(a_1, p_h)$ (in Steps 9–11). Note that the root node of $Subtree(a_1, p_h)$ is labeled* a *because both the root nodes of $Subtree(a_1, p_c)$ and $Subtree(a, p_f)$ are labeled* a*. The sub-patterns $Subtree(a_2, p_c)$ and $Subtree(b, p_f)$, however, have quite different structures and thus a "position-preserving" attempt to extract their common sub-patterns only yields*

$Subtree(*_1, p_h)$. *In particular, the common sub-pattern consisting of an* a*-node with both a* b*-child-node and* c*-child-node is not captured by the above process because they occur at different positions relative to the root nodes of* $Subtree(a_2, p_c)$ *and* $Subtree(b, p_f)$. *To extract such "off-position" common sub-patterns, Algorithm* LUB_SUB *compares* $Subtree(a_1, p_c)$ *with* $Subtree(b, p_f)$ *and* $Subtree(c, p_f)$*, as well as compares* $Subtree(a, p_f)$ *with* $Subtree(a_2, p_c)$ *(in Steps 12–15). Indeed, this yields* $Subtree(//_3, p_h)$ *which has a* $//$*-root since this common sub-pattern occurs at different positions relative to the root nodes of* $Subtree(a_1, p_c)$ *and* $Subtree(a, p_f)$*. It should be mentioned that both* $Subtree(//_1, p_h)$ *and* $Subtree(//_2, p_h)$ *are also produced by the "off-position" processing, as Algorithm* LUB_SUB *recursively processes the sub-pattern* $Subtree(a_2, p_c)$ *with* $Subtree(b, p_f)$ *and* $Subtree(c, p_f)$*, respectively. Finally, the algorithm removes the redundant nodes in the result tree pattern by using a minimization algorithm (which will be explained shortly) to generate the LUB* p_h. □

It is straightforward to show that our LUB operator "$\sqcup$", considered as a binary operator, is *commutative* and *associative*, i.e., $p_1 \sqcup p_2 = p_2 \sqcup p_1$ and $p_1 \sqcup (p_2 \sqcup p_3) = (p_1 \sqcup p_2) \sqcup p_3$. As a result, Algorithm LUB can be naturally extended to compute the LUB of any set of tree patterns. We next explain the details of the two auxiliary algorithms used in Algorithm LUB.

Algorithm LUB needs to check the containment of tree patterns, which is implemented by Algorithm CONTAINS in Figure 4. Given two input tree patterns p and q, the algorithm determines if $q \sqsubseteq p$. It maintains a two-dimensional array $Status$, which is initialized with $Status[v, w] = null$ to indicate that $v \in Nodes(p)$ and $w \in Nodes(q)$ have not been compared; otherwise, $Status[v, w] \in \{true, false\}$ such that $Status[v, w] = true$ if and only if $Subtree(w, q) \sqsubseteq Subtree(v, p)$. Clearly, $q \sqsubseteq p$ if and only if $Status[v_{root}, w_{root}] = true$, where v_{root} and w_{root} denote the root nodes of p and q, respectively.

The main subroutine in our containment algorithm is Algorithm CONTAINS_SUB. Abstractly, CONTAINS_SUB traverses p and q top-down and updates $Status[v, w]$ for each pair of nodes $v \in Nodes(p)$ and $w \in Nodes(q)$ visited as follows. Let p' and q' denote $Subtree(v, p)$ and $Subtree(w, q)$, respectively. If $Status[v, w]$ has already been computed (i.e., $Status[v, w] \neq null$), then its value is returned. Otherwise, our algorithm determines whether $q' \sqsubseteq p'$, as follows. If $label(v) \neq //$, then $Status[v, w] = true$ iff $label(w) \preceq label(v)$ and each child subtree of v contains some child subtree of w. Otherwise, if $label(v) = //$, two additional conditions need to be taken into account. This is because unlike a $*$-node or a tag-name-node, a $//$-node in a container tree pattern can also be "mapped" to a (possibly empty) chain of nodes in a contained tree pattern. For example, consider the tree patterns p_d and p_f in Figure 2. Note that $p_f \sqsubseteq p_d$, and the $//$-node in p_d is not mapped to any node in p_f in the sense that p_f would still be contained in p_d if the $//$-node in p_d is deleted. On the other hand, for the tree patterns p_d and p_g in Figure 2, $p_g \sqsubseteq p_d$ and the $//$-node in p_d is mapped to both the $*$- and b-nodes in p_g in the sense that $Subtree(*, p_g) \sqsubseteq Subtree(//, p_d)$ and $Subtree(b, p_g) \sqsubseteq Subtree(//, p_d)$. These two additional scenarios are handled by Steps 10 and 12 in Algorithm CONTAINS_SUB: Step 10 accounts for the case where a $//$-node (v itself) is mapped to an empty chain of nodes, and Step 12 for the case where a $//$-node (v itself) is mapped to a non-empty chain. Note that in Steps 8 and 12, the expression $\bigvee_{w' in Child(w,q)}$ CONTAINS_SUB $(x, w', Status)$ returns $false$ if $Child(w, q) = \emptyset$.

Algorithm CONTAINS (p, q)
Input: p and q are two tree patterns.
Output: Returns $true$ if $q \sqsubseteq p$; $false$ otherwise.
1) Initialize $Status[v, w] = null$,
   $\forall\, v \in Nodes(p)$, $\forall\, w \in Nodes(q)$;
2) Let v_{root} and w_{root} denote the root nodes of p and q, resp.;
3) **if** $(Child(v_{root}, p) = \emptyset)$ **then**
4) **return** $true$;
5) **else**
6) **return** CONTAINS_SUB $(v_{root}, w_{root}, Status)$;

Algorithm CONTAINS_SUB $(v, w, Status)$
Input: v, w are nodes in tree patterns p, q (respectively),
$Status$ is a 2-dimensional array such that each
$Status[v, w] \in \{null, false, true\}$.
Output: $Status[v, w]$.
1) **if** $(Status[v, w] \neq null)$ **then**
2) **return** $Status[v, w]$;
3) **if** (v is a leaf node in p) **then**
4) $Status[v, w] = (label(w) \preceq label(v))$;
5) **else if** $(label(w) \not\preceq label(v))$ **then**
6) $Status[v, w] = false$;
7) **else**
8) $Status[v, w] =$
$$\bigwedge_{v' \in Child(v,p)} \left(\bigvee_{w' \in Child(w,q)} \texttt{CONTAINS\_SUB}\ (v', w', Status) \right);$$
9) **if** $(Status[v, w] = false)$ **and** $(label(v) = //)$ **then**
10) $Status[v, w] =$
$\bigwedge_{v' \in Child(v,p)}$ CONTAINS_SUB $(v', w, Status)$;
11) **if** $(Status[v, w] = false)$ **and** $(label(v) = //)$ **then**
12) $Status[v, w] = \bigvee_{w' \in Child(w,q)}$ CONTAINS_SUB $(v, w', Status)$;
13) **return** $Status[v, w]$;

Figure 4: Tree-Pattern Containment Algorithm.

By induction on the structures of p and q, we can show the following result.

Proposition 3.2: *Given two tree patterns* p *and* q*, Algorithm* CONTAINS(p, q) *determines if* $q \sqsubseteq p$ *in* $O(|p| \cdot |q|)$ *time.* □

The quadratic time complexity of our tree-pattern containment algorithm is due to, among other things, the fact that each pair of sub-patterns in p and q is checked at most once, because of the use of the $Status$ array. To simplify the discussion, we have omitted from Algorithm CONTAINS certain subtle details that involve tree patterns with chains of $//$- and $*$-nodes. Such cases require some additional pre-processing to convert the tree pattern to some canonical form, but this does not increase our algorithm's time complexity.

To ensure that our tree patterns are concise, we need to identify and eliminate "redundant" nodes in them. Given a tree pattern p, a minimized tree pattern p' equivalent to p can be computed using a recursive algorithm MINIMIZE. Starting with the root of p, our minimization algorithm performs the following two steps to minimize the sub-pattern $Subtree(v,p)$ rooted at node v in p: (1) For any $v', v'' \in Child(v,p)$, if $Subtree(v',p) \sqsubseteq Subtree(v'',p)$, then delete $Subtree(v',p)$ from $Subtree(v,p)$; and, (2) For each $v' \in Child(v,p)$ (that was not deleted in the first step), recursively minimize $Subtree(v',p)$. The complete details can be found in [5].

Proposition 3.3: *Algorithm* MINIMIZE *minimizes any tree pattern p in $O(|p|^2)$ time.* □

Proposition 3.4: *For any minimized tree patterns p and p', $p \equiv p'$ iff $p = p'$ (i.e., they are syntactically equal).* □

Given the low computational complexities of CONTAINS and MINIMIZE, one might expect that this would also be the case for Algorithm LUB. Unfortunately, in the worst case, the size of the (minimized) LUB of two tree patterns can be exponentially large (see [5] for a detailed analysis). Our implementation results, however, demonstrate that our LUB algorithm exhibits reasonably low average-case complexity in practice.

4 Selectivity-based Aggregation Algorithm

While the LUB algorithm presented in the previous section can be used to compute a single, most precise aggregate tree pattern for a given set S of patterns, the size of the LUB may be too large and, therefore, may violate the specified space constraint k on the total size of the aggregated subscriptions (Section 2.2). Thus, in order to fit our aggregates within the allotted space budget, we relax the requirement of a single precise aggregate by permitting our solution to be a *set* $S' = \{p_1, p_2, \ldots, p_m\}$ (instead of a single pattern), such that each pattern $q \in S$ is contained in some pattern $p_i \in S'$. Of course, we also require that S' provide the "tightest" containment for patterns in S for the given space constraint (Section 2.2); that is, the number of XML documents that satisfy some tree pattern in S' but not S, is small.

A simple measure of the preciseness of S' is its *selectivity*, which is essentially the fraction of filtered XML documents that satisfy some pattern in S'. Thus, our objective is to compute a set S' of aggregate patterns whose selectivity is very close to that of S. Clearly, the selectivity of our tree patterns is highly dependent on the distribution of the underlying collection of XML documents (denoted by D). It is, however, infeasible to maintain the detailed distribution D of streaming XML documents for our aggregation—the space requirements would be enormous! Instead, our approach is based on building a *concise synopsis* of D on-line (i.e., as documents are streaming by), and using that synopsis to estimate (approximate) tree-pattern selectivities. At a high level, our aggregation algorithm iteratively computes a set S' that is both selective and satisfies the space constraint, starting with $S' = S$ (i.e., the original set S of patterns), and performing the following sequence of steps in each iteration:

1. Generate a candidate set of aggregate tree patterns C consisting of patterns in S' and LUBs of similar pattern pairs in S'.
2. Prune each pattern p in C by deleting/merging nodes in p in order to reduce its size.
3. Choose a candidate $p \in C$ to replace all patterns in S' that are contained in p. Our candidate-selection strategy is based on *marginal gains* [14]: The selected candidate p is the one that results in the minimum loss in selectivity per unit reduction in the size of S' (due to the replacement of patterns in S' by p).

Note that our pruning step (Step 2) above makes candidate aggregate patterns less selective (in addition to decreasing their size). Thus, by replacing patterns in S' by patterns in C, we are effectively trying to reduce the size of S' by giving up some of its selectivity.

In the following subsections, we describe in more detail our algorithm for computing S'. We begin by presenting our approach for estimating the selectivity of tree patterns over the underlying document distribution, which is critical to choosing a good replacement candidate in Step 3 above.

4.1 Selectivity Estimation for Tree Patterns

The Document Tree Synopsis. As mentioned above, it is simply impossible to maintain the accurate document distribution D (i.e., the full set of streaming documents) in order to obtain accurate selectivity estimates for our tree patterns. Instead, our approach is to approximate D by a concise synopsis structure, which we refer to as the *document tree*. Our document tree synopsis for D, denoted by DT, captures path statistics for documents in D, and is built *on-line* as XML documents stream by. The document tree essentially has the same structure as an XML tree, except for two differences. First, the root node of DT has the special label "/.". Second, each non-root node t in DT has a frequency associated with it, which we denote by $freq(t)$. Intuitively, if $l_1/l_2/\cdots/l_n$ is the sequence of tag names on nodes along the path from the root to t (excluding the label for the root), then $freq(t)$ represents the number of documents T in D that contain a path with tag sequence $l_1/l_2/\cdots/l_n$ originating at the root of T. The frequency for the root node of DT is set to N, the number of documents in D.

As XML documents stream by, DT is incrementally maintained as follows. For each arriving document T, we first construct the *skeleton tree* T_s for document T. In the skeleton tree T_s, each node has at most one child with a given tag. T_s is built from T by simply coalescing two children of a node in T if they share a common tag. Clearly, by traversing nodes in T in a top-down fashion, and coalescing

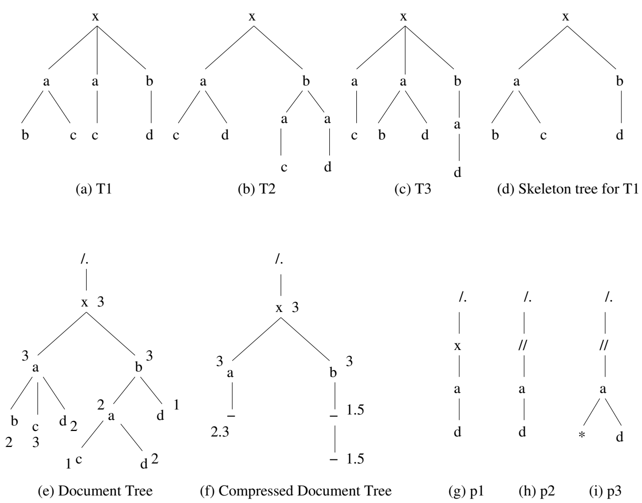


Figure 5: Example Documents, Skeleton Tree, Document Tree, and Patterns.

child nodes with common tags, we can construct T_s from T in a single pass (using an event-based XML parser). As an example, Figure 5(d) depicts the skeleton tree for the XML-document tree in Figure 5(a).

Next, we use T_s to update the statistics maintained in our document tree synopsis DT as follows. For each path in T_s, with tag sequence say $l_1/l_2/\cdots/l_n$, let t be the last node on the corresponding (unique) path in DT. We increment $freq(t)$ by 1. Figure 5(e) shows the document tree (with node frequencies) for the XML trees T_1, T_2, and T_3 in Figure 5(a) to (c). Note that it is possible to further compress DT by using techniques similar in spirit to the methods employed by Aboulnaga et al. [1] for summarizing *path* trees. The key idea is to merge nodes with the lowest frequencies and store, with each merged node, the average of the original frequencies for nodes in DT that were merged. This is illustrated in Figure 5(f) for the document tree in Figure 5(e), and with the label "–" used to indicate merged nodes. Due to space constraints, in the remainder of this subsection, we only present solutions to the selectivity estimation problem using the uncompressed tree DT. However, our proposed methods can be easily extended to work even when DT is compressed [5].

We should note here that our selectivity estimation problem for tree patterns differs from the work of Aboulnaga et al. [1] in two important respects. First, in [1], the authors consider the problem of estimating selectivity for only simple paths that consist of a //-node followed by tag nodes. In contrast, we estimate selectivities of general tree patterns with branches, and *- or //-nodes arbitrarily distributed in the tree. Second, we are interested in selectivity at the granularity of *documents*, so our goal is to estimate the number of XML documents that match a tree pattern; instead, [1] addresses the selectivity problem at the granularity of individual *document elements* that are discovered by a path. It is easy to see that these are two very different estimation problems.

Selectivity Estimation Procedure. Recall that the selectivity of a tree pattern p is the fraction of documents T in D that satisfy p. By construction, our DT synopsis gives accurate selectivity estimates for tree patterns comprising a single chain of tag-nodes (i.e., with no * or //). However, obtaining accurate selectivity estimates for arbitrary tree patterns with branches, *, and // is, in general, not possible with DT summaries. This is because, while DT captures the number of documents containing a single path, it does not store document identities. As a result, for a pair of arbitrary paths in a tree pattern, it is impossible to determine the exact number of documents that contain both paths or documents that contain one path, but not the other.

Our estimation procedure solves this problem, by making the following simplifying assumption: *The distribution of each path in a tree pattern is independent of other paths.* Thus, we estimate the selectivity of a tree pattern containing no $//$ or $*$ labels, simply as the *product* of the selectivities of each root to leaf path in the pattern. For patterns containing $//$ or $*$, we consider all possible instantiations for $//$ and $*$ with element tags, and then choose as our pattern selectivity the maximum selectivity value over all instantiations. (This is similar to the definition of a fuzzy `OR` operator in fuzzy logic [13].) We illustrate our selectivity estimation methodology in the following example.

Example 4.1 *Consider the problem of estimating the selectivities of the tree patterns shown in Figures 5(g) to (i) using the document tree shown in Figure 5(e). The total number of documents, N, is* 3. *Clearly, the number of documents satisfying pattern p_1 which consists of a single path, can be estimated accurately by following the path in DT and returning the frequency for the* d*-node (at the end of the path) in DT. Thus, the selectivity of p_1 is $2/3$ which is accurate since only documents T_2 and T_3 satisfy p_1. Estimating the number of documents containing pattern p_2, however, is somewhat more tricky. This is because there are two paths with tag sequences $x/a/d$ and $x/b/a/d$ in DT that match p_2 (corresponding to instantiating // with x and x/a). Summing the frequencies for the two* d*-nodes at the end of these paths gives us an answer of 4 which over-estimates the number of documents satisfying p_2 (only documents T_2 and T_3 satisfy p_2). To avoid double-counting frequencies, we estimate the number of documents satisfying p_2 to be the maximum (and not the sum) of frequencies over all paths in DT that match p_2. Thus, the selectivity of p_2 is estimated as $2/3$.*

*Finally, the selectivity of p_3 is computed by considering all possible instantiations for // and *, and choosing the one with the maximum selectivity. The two possible instantiations for // that result in non-zero selectivities are x and x/b, and $*$ can be instantiated with either* b, c *or* d *for $// = x$, and* c *or* d *for $// = x/b$. Choosing $// = x$ and $* =$ c results in the maximum selectivity since the product of the selectivities of paths $x/a/c$ and $x/a/d$ is maximum, and is equal to $(3/3) \cdot (2/3) = 2/3$.* □

Algorithm `SEL` (depicted in Figure 6), invoked with input parameters $v = v_{root}$ (root of pattern p) and $t = t_{root}$ (root of DT), computes the selectivity for an arbitrary tree

Algorithm SEL(v, t)
Input: v is a node in tree pattern p, t is a node in DT.
Output: $SelSubPat[v, t]$.
1) **if** ($SelSubPat[v, t]$ is already computed) **then**
2) **return** $SelSubPat[v, t]$;
3) **else if** ($label(t) \not\preceq label(v)$) **then**
4) **return** $SelSubPat[v, t] = 0$;
5) **else if** (v is a leaf) **then**
6) **return** $freq(t)/N$;
7) **for** each child $v_c \in Child(v, p)$ **do**
8) $Sel_{v_c} = \max_{t_c \in Child(t,DT)}\{\texttt{SEL}\ (v_c, t_c)\}$;
9) $Sel = \prod_{v_c \in Child(v,p)} Sel_{v_c}$;
10) **if** ($label(v) = //$) **then**
11) $Sel_v = \prod_{v_c \in Child(v,p)} \texttt{SEL}(v_c, t)$;
12) $Sel = \max\{Sel, Sel_v\}$;
13) $Sel_v = \max_{t_c \in Child(t,DT)}\{\texttt{SEL}(v, t_c)\}$;
14) $Sel = \max\{Sel, Sel_v\}$;
15) **return** $SelSubPat[v, t] = Sel$

Figure 6: Tree Pattern Selectivity Estimation Algorithm.

pattern p in $O(|DT| \cdot |p|)$ time. In the algorithm, for nodes $v \in p$ and $t \in DT$, $SelSubPat[v, t]$ stores the selectivity of the sub-pattern $Subtree(v, p)$ with respect to the subtree of DT rooted at node t. This selectivity is estimated similar to the selectivity for pattern p, except that we now consider all instantiations of $Subtree(v, p)$ (obtained by instantiating $//$ and $*$ with element tags), and the selectivity of each instantiation is computed with respect to t as the root instead of the root of DT. For instance, suppose that v is the a-node in p_3 (in Figure 5(i)), and t is the child a-node of the x-node in DT (in Figure 5(e)). Then, the selectivity of $Subtree(v, p_3)$ with respect to t is essentially the product of the selectivity of paths $a/*$ and a/d with respect to node t, which is $1 \cdot (2/3)$. Thus, $SelSubPat[v, t] = 2/3$.

Our goal is to compute $SelSubPat[v_{root}, t_{root}]$. For a pair of nodes v and t, Algorithm SEL computes $SelSubPat[v, t]$ from $SelSubPat[\ ]$ values for the children of v and t. Clearly, if $label(t) \not\preceq label(v)$ (Steps 3-4 of the algorithm), then every path in $Subtree(v, p)$ begins with a label different from $label(t)$ and thus the selectivity of each of the paths is 0. If $label(t) \preceq label(v)$ and v is a leaf (Steps 5-6), then we simply instantiate $label(v)$ (if $label(v) = //$ or *) with $label(t)$, giving a selectivity of $freq(t)/N$. On the other hand, if v is an internal node of p, then in addition to instantiating $label(v)$ with $label(t)$, we also need to compute, for every child v_c of v, the instantiation for $Subtree(v_c, p)$ that has the maximum selectivity with respect to some child t_c of t. Since $SelSubPat[v_c, t_c]$ is the selectivity of $Subtree(v_c, p)$ with respect to t_c, the product of $\max_{t_c \in Child(t,DT)} SelSubPat[v_c, t_c]$ for the children v_c of v gives the selectivity of $Subtree(v, p)$ with respect to t. Finally, if $label(v) = //$, then $//$ can be simply $null$, in which case the selectivity of $Subtree(v, p)$ with respect to t is computed as described in Step 11, or $//$ is instantiated to a sequence consisting of $label(t)$ followed by $label(t_c)$, where t_c is the child of t such that the selectivity of $Subtree(v, p)$ with respect to t_c is maximized (Step 13). Observe that, in Steps 8 and 13, if t has no children, then $\max_{t_c \in Child(t,DT)}\{\ldots\}$ evaluates to 0.

4.2 Tree Pattern Aggregation Algorithm

We are now ready to present our greedy heuristic algorithm for the tree pattern aggregation problem defined in Section 2.2 (which is, in general, an $\mathcal{NP}$-hard clustering problem [5]). As described earlier, to aggregate an input set of tree patterns S into a space-efficient and precise set, our algorithm (Algorithm AGGREGATE in Figure 7) iteratively prunes the tree patterns in S by replacing a small subset of tree patterns with a more concise upper-bound *aggregate pattern*, until S satisfies the given space constraint. During each iteration, our algorithm first generates a small set of potential candidate aggregate patterns C, and selects from these the (locally) "best" candidate pattern, i.e., the candidate that maximizes the gain in space while minimizing the expected loss in selectivity.

Algorithm AGGREGATE (S, k)
Input: S is a set of tree patterns, k is a space constraint.
Output: A set of tree patterns S' such that $S \sqsubseteq S'$ and $\sum_{p \in S'} |p| \leq k$.
1) Initialize $S' = S$;
2) **while** ($\sum_{p \in S'} |p| > k$) **do**
3) $C_1 = \{x \mid x = \texttt{PRUNE}(p, |p| - 1),\ p \in S'\}$;
4) $C_2 = \{x \mid x = \texttt{PRUNE}(p \sqcup q, |p| + |q| - 1),\ p, q \in S'\}$;
5) $C = C_1 \cup C_2$;
6) Select $x \in C$ such that $Benefit(x)$ is maximum;
7) $S' = S' - \{p \mid p \sqsubseteq x, p \in S'\} \cup \{x\}$;
8) **return** S';

Figure 7: Tree Pattern Aggregation Algorithm.

Candidate Generation. We now explain the process for generating the candidate set C in Steps 3–5 of Algorithm AGGREGATE. To reduce the size of individual candidate patterns of the form p or $p \sqcup q$, each candidate is *pruned* by invoking Algorithm PRUNE (details in [5]). Given an input pattern p and space constraint n, Algorithm PRUNE prunes p to a smaller tree pattern p' such that $p \sqsubseteq p'$ and $|p'| \leq n$. The algorithm treats tag-nodes as more selective than $*$- and $//$-nodes, and therefore tries to prune away $*$- and $//$-nodes before the tag-nodes. Specifically, the algorithm first prunes the $*$- and $//$-nodes in p by (1) replacing each adjacent pair of non-tag-nodes v, w with a single $//$-node, if w is the only child of v, and (2) eliminating subtrees that consist of only non-tag-nodes. If the tree pattern is still not small enough after the pruning of the non-tag-nodes, we start pruning the tag-nodes. There are two ways to reduce the size of a tree pattern p by one node. The first is to delete some leaf node in p, and the second is to collapse two nodes v and w into a single $//$-node, where $label(v) \neq /.$ and $Child(v, p) = \{w\}$. To help select a "good" leaf node to delete (or, pair of nodes to collapse), we make use of the selectivity of the tag names. More specifically, we use our document tree synopsis DT to estimate the total number of occurrences of a tag name in the document collection D, and then choose the tags with higher total frequencies (which are less selective) as candidates for pruning.

Candidate Selection. Once the set of candidate aggregate patterns has been generated, we need some criterion for selecting the "best" candidate to insert into S'. For this purpose, we associate a benefit value with each candidate aggregate pattern $x \in C$, denoted by $Benefit(x)$, based on its *marginal gain* [14]; that is, we define $Benefit(x)$ as the ratio of the savings in space to the loss in selectivity of using x over $\{p \mid p \sqsubseteq x, p \in S'\}$. More formally, if $v_{x_{root}}$, t_{root}, and $v_{p_{root}}$ represent the root nodes of x, DT, and $p \in S'$, then $Benefit(x)$ is equal to:

$$\frac{\left(\sum_{p \sqsubseteq x, p \in S'} |p|\right) - |x|}{\mathtt{SEL}(v_{x_{root}}, t_{root}) - \max_{p \sqsubseteq x, p \in S'} \mathtt{SEL}(v_{p_{root}}, t_{root})}$$

Note that we compute the selectivity loss by comparing the selectivity of the candidate aggregate pattern x with that of the least selective pattern contained in it. This gives a good approximation of the selectivity loss in cases when the patterns $p, q \in S'$ used to generate x are similar and overlap in the document tree DT. The candidate aggregate pattern with the highest benefit value is chosen to replace the patterns contained in it in S' (Steps 6–7).

5 Experimental Study

To verify the effectiveness of our tree pattern aggregation algorithms, we have conducted an extensive performance study using real-life DTDs and large numbers of tree patterns. Our results indicate that our proposed aggregation techniques achieve significant reductions in the number as well as total size of tree patterns with minimal loss in selectivity.

5.1 Experimental Testbed and Methodology

Our general methodology for evaluating the effectiveness of a pattern aggregation algorithm A is as follows. Given a large input set of tree patterns S and a space constraint k, we use A to compute a set of aggregate patterns S' for S, where $S \sqsubseteq S'$ and $\sum_{p \in S'} |p| \leq k$ (our space constraint is expressed in terms of number of nodes, since patterns can be arbitrarily large). We then measure the loss in precision when using S' instead of S to filter XML documents. Observe that when $k = 1$, S' contains a single container pattern ("$//$").

To measure the loss in precision of the aggregate set S', we use a subset D' of a representative set of XML documents, such that no document in D' matches any tree pattern in our initial pattern set S. The reason, of course, is that XML documents that match S are also guaranteed to match S', so they are unlikely to affect our "precision-loss" measurements. As S' becomes less precise, some documents in D' will be erroneously reported as matches. Let $Matches(D', S')$ be the number of documents in D' that match S'; the loss in precision of S' over S can be estimated as $SelLoss(S', S) = Matches(D', S')/|D'|$. An aggregation algorithm is obviously more effective if $SelLoss(S', S)$ remains small as $\sum_{p \in S'} |p|$ decreases.

XML Documents. We used two real-life DTDs to generate our XML document data set. The first one, the Extensible Hypertext Markup Language (XHTML) DTD [7], is a reformulation of HTML as an XML application and is arguably the document type most widely used over the Internet. The XHTML DTD (version 1.0) contains 77 elements with 1377 attributes. The second DTD, the News Industry Text Format (NITF) DTD[8], is supported by most of the world's major news agencies. The NITF DTD (version 2.5) contains 123 elements with 513 attributes.

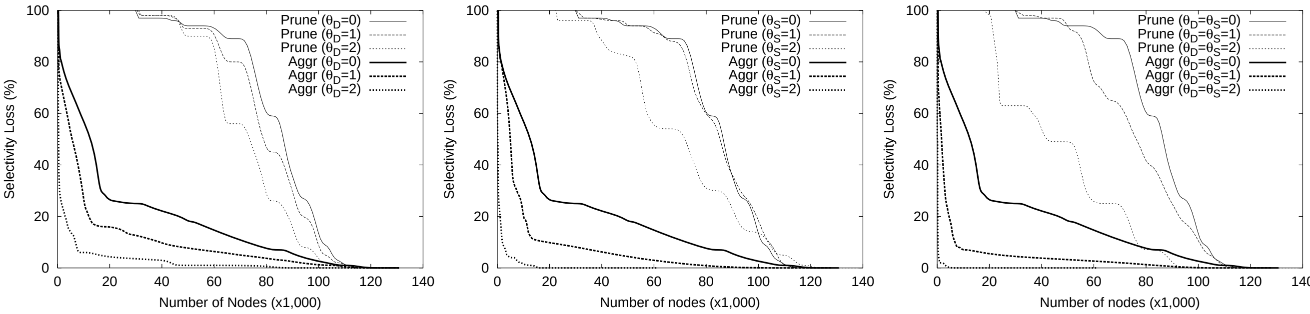
We generated our data set of XML documents using IBM's XML Generator tool [11]. Both the XHTML and NITF DTDs contain recursive structures, which can be nested to produce XML documents with arbitrary number of levels. We added the option of generating documents skewed according to a Zipf distribution [18], where some tag names appear more frequently than others, as is generally the case with real-life data.

For each each DTD and each skew value $\theta_D = \{0, 1, 2\}$, we generated two disjoint sets of 500 XML documents with approximately 100 nodes and 10 levels on average. The first set corresponds to the collection of XML documents used to construct the document tree DT for selectivity estimation; the second set is used to measure the loss in precision of the aggregation algorithms. Both sets were generated with the same parameters, and thus can be expected to have similar distributions. In each experiment, we used the combined XML documents for both the XHTML and NITF DTDs, i.e., we used a total of 1000 documents for the document tree DT, and (a different) 1000 documents for measuring the loss in precision.

XPath Expressions. To generate the set of tree patterns S, we implemented an XPath expression generator that takes a DTD as input and creates a set of valid XPath expressions based on a set of parameters that control: (1) the maximum height h of the tree patterns; (2) the probabilities p_* and $p_{//}$ of having a wildcard "$*$" or a descendant "$//$" operator at a node of a tree pattern; (3) the probability $p_{\curlywedge}$ of having more than one child at a given node; and (4) the skew θ_S of the Zipf distribution used for selecting element tag names.

For each DTD and each skew value $\theta_S = \{0, 1, 2\}$, we generated a set of 5000 tree patterns with $h = 10$ and $p_* = p_{//} = p_{\curlywedge} = 0.1$. Each experiment was run with tree patterns from both the XHTML and NITF DTDs, i.e., 10000 tree patterns which amounted to more than 100000 nodes.

Algorithms. We compared two different aggregation algorithms in our experiments. The first ("naive") algorithm, PRUNE, is based on simple node pruning and works as follows. At each iteration, it selects a tree pattern p_{max} from S with the largest number of tag-nodes, collapses multiple $*$- and $//$-nodes, and deletes a prunable node (i.e., a leaf node or a node located next to $//$-nodes) with the highest frequency (i.e., least selective) in the document tree DT. If there is already a tree pattern identical to the pruned pattern, then the duplicate is removed from S. The algorithm iterates until the space constraint is satisfied. The second algorithm, AGGR, is our greedy tree pattern aggregation algorithm (from Figure 7) with both candidate generation and selection (based on maximizing the benefit). Our experiments were conducted on a 866 MHz Intel Pentium III



(a) Varying θ_D ($\theta_S = 0$)

(b) Varying θ_S ($\theta_D = 0$)

(c) Varying θ_S and θ_D

Figure 8: Evaluation of the Aggregation Algorithms.

machine with 512 MB of main memory running Linux. Both algorithms completed the aggregation of 10000 tree patterns in approximately 10 minutes.

5.2 Experimental Results

We first compare the performance of the two aggregation algorithms by varying the skew for element tags in the XML documents and in the XPath expressions. We ran the experiments with no skew, with skewed XML documents, with skewed XPath expressions, and with skew in both the XML documents and XPath expressions. In the last case, we skew the distribution for element names in the opposite "direction" (applying the same skew to both the XML documents and XPath expressions would yield similar results as with no skew). The experimental results are shown in Figures 8(b), 8(a), and 8(c), where the space constraint, expressed in terms of the number of nodes, is varied along the x-axis, and the y-axis indicates the observed loss in selectivity for a given space constraint, i.e., the percentage of XML documents that are erroneously reported as matches.

We also measure the benefits of aggregation in terms of filtering performance, using the XTrie matching algorithm described in [6]. Since the cost of filtering in XTrie grows linearly with the number of XPath expressions, we expect to observe a significant improvement in filtering speed as the cardinality of S decreases.

Non-skewed workload. When neither the XML data nor the tree patterns contain skew (i.e., $\theta_D = \theta_S = 0$), the AGGR algorithm can aggregate tree patterns up to 15% of their original size with only a 25% loss in precision (the results for non-skewed data are reported in all graphs of Figure 8). In contrast, the precision of PRUNE algorithm starts to degrade much sooner, and the loss in precision reaches almost 100% at 25% of the initial space. The better performance of AGGR can be attributed to three main factors: (1) the upper bound computation generates good candidates with few nodes and little loss in precision, (2) the selectivity-based heuristics help to detect and discard candidates that correspond to patterns with low selectivity (i.e., frequently occurring for a given DTD), and (3) the covering computation enables redundant tree patterns to be eliminated early.

Skewed XML documents. Real-world XML documents are generally not uniformly distributed among the valid XML data for a given DTD. When XML documents are skewed (Figure 8(a)), we observe that the effectiveness of the AGGR algorithm increases. The reason for this is that, as data becomes more skewed, the XML documents tend to form clusters with documents within a cluster being more similar than those in different clusters; this, in turn, improves the accuracy of selectivity estimation. The PRUNE algorithm also benefits from the skew (although to a lesser extent) because of its frequency-based pruning heuristic.

Skewed tree patterns. We also observe a significant improvement in our aggregation algorithm when the element names of tree patterns are skewed (Figure 8(b)). Indeed, the skew induces a clustering of patterns such that similar tree patterns are grouped into the same cluster, which consequently increases the proportion of patterns that develop containment relationships. This permits the aggregation algorithm to reduce the size of S with minimal loss of selectivity, by computing tighter upper bound patterns and discarding covered patterns.

Skewed workload. The two aggregation algorithms perform best when both the XML data and the tree patterns are skewed in different "directions" (Figure 8(c)). With high skew values, there is little overlap between the element names of the XML documents and the tree patterns, and AGGR remains highly selective with only a few hundreds nodes. The PRUNE algorithm also exhibits significant improvements and maintains 50% selectivity even after the original number of nodes are reduced to less than a third.

Filtering speed. As mentioned previously, the cost of matching tree patterns against incoming XML documents is proportional to the number of tree patterns. Since AGGR generates candidates by computing upper bounds, the candidates cover more patterns, and as result, the number of patterns in S shrinks faster with AGGR. Figure 9 shows that the average filtering time per document decreases faster (as space is increased) for AGGR than for the PRUNE algorithm. Our aggregation algorithm is therefore more effec-

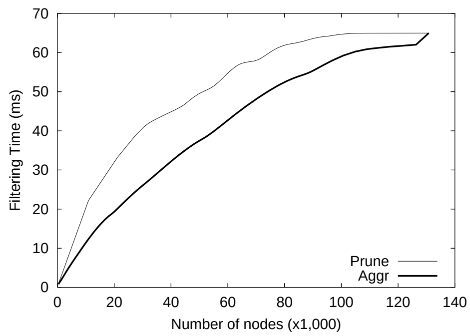


Figure 9: Filtering speed.

tive both in terms of selectivity as well as filtering speed.

6 Related Work

To the best of our knowledge, our tree pattern aggregation problem is a novel problem that has not been studied in earlier work. In contrast to the "flat patterns" previously studied in the context of aggregating attribute-predicate-based subscriptions [12], our paper focuses on hierarchical patterns, which are more complex (as tree patterns consist of both data contents and structure) and require more sophisticated aggregation techniques.

A related area is the work on query merging to reduce data dissemination costs of query subscriptions in a multicast environment [9]. The motivation for query merging is to merge multiple similar queries into a single, more general query so as to reduce the workload of the server and possibly the amount of traffic between the server and its clients. However, the problem domain considered in [9] focuses on geographical queries (represented as rectangles); furthermore, the issue of space constraint is not relevant there.

Some forms of tree patterns have been studied as queries for XML data [3, 17]. In particular, minimization algorithms for these patterns have been developed in order to optimize pattern queries. The tree patterns in [3] differ from ours in two aspects. On the one hand, the tree patterns of [3] do not allow $*$-nodes (wildcards) which, as mentioned in Section 3, give rise to subtle problems in the presence of $//$-nodes (descendants) when containment of tree patterns is considered. On the other hand, they support selection of a set of document nodes as the result of a pattern query, which we do not consider since what matters for our subscription aggregation context is whether or not a document matches a subscription; the actual set of document nodes that matches a subscription is not relevant. Because of these differences, the minimization algorithm of [3] has an $O(n^4)$ complexity in contrast to our $O(n^2)$ complexity. Similarly, the work in [17] studies a different class of tree patterns and their minimization algorithm is only known to be in polynomial time.

7 Conclusions

We have provided the first systematic study of *tree pattern aggregation*, an important problem in building next-generation, scalable XML dissemination systems. The main challenge is to aggregate an input set of tree patterns into a smaller set such that: (1) a given space constraint on the total size of the patterns is met, and (2) the loss in precision (due to aggregation) is minimized. We have proposed an efficient aggregation algorithm that makes effective use of document-distribution statistics in order to compute a precise set of aggregate tree patterns within the allotted space budget. Further, some of our algorithmic results are of interest in their own right, and can prove useful in other domains, such as XML query optimization. Extensive results from a prototype implementation have verified the effectiveness of our approach.

References

[1] A. Aboulnaga, A. Alameldeen, and J. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *Proc. 27th Intl. Conf. on Very Large Databases (VLDB 2001)*, September 2001.

[2] M. Altinel and M.J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proc. 26th Intl. Conf. on Very Large Databases (VLDB 2000)*, pages 53–64, September 2000.

[3] S. Amer-Yahia, S. Cho, L.V.S. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *Proc. of SIGMOD*, pages 497–508, Santa Barbara, California, May 2001.

[4] A. Carzaniga and A.L. Wolf. Content-based Networking: A New Communication Infrastructure. NSF Workshop on Infrastructure for Mobile and Wireless Systems, October 2001.

[5] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. Bell Labs Tech. Memorandum, February 2002.

[6] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of the 18th Intl. Conf. on Data Engineering*, San Jose, California, February 2002.

[7] World Wide Web Consortium. *The SGML/XML Web Page*. http://www.w3.org/TR/xhtml1/, January 2000.

[8] R. Cover. *The SGML/XML Web Page*. http://www.oasis.-open.org/cover/sgml-xml.html, December 1999.

[9] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query Merging: Improving Query Subscription Processing in a Multicast Environment. *IEEE Trans. on Knowledge and Data Engineering*. To appear.

[10] A. Deutsch and V. Tannen. Containment of Regular Path Expressions under Integrity Constraints. In *Proc. of Intl. Workshop on Knowledge Representation meets Databases (KRDB)*, 2001.

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

[12] L. Opyrchal et al. Exploiting IP Multicast in Content-based Publish-Subscribe Systems. In *Proc. of Intl. Conf. on Distributed Systems Platforms (Middleware)*, 2000.

[13] R. Fagin. Combining Fuzzy Information from Multiple Systems. In *Proc. of the 15th ACM Symp. on Principles of Database Systems*, Montreal, Quebec, June 1996.

[14] B. Fox. Discrete Optimization Via Marginal Analysis. *Management Science*, 13(3):211–216, November 1966.

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

[16] W3C. *Extensible Markup Language (XML) 1.0*. http://www.w3.org/XML/, 2000.

[17] P. T. Wood. Minimizing Simple XPath Expressions. In *Proc. of Intl. Workshop on the Web and Databases (WebDB)*, Santa Barbara, California, May 2001.

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