AxPRE Summaries: Exploring the (Semi-)Structure of XML Web Collections*

Mariano P. Consens
University of Toronto
consens@cs.toronto.edu

Flavio Rizzolo
University of Toronto
flavio@cs.toronto.edu

Alejandro A. Vaisman
Universidad de Buenos Aires
avaisman@dc.uba.ar

Abstract

The nature of semistructured data in web collections is evolving. Increasingly, XML web documents (or documents exchanged via web services) are valid with regard to a schema, yet the actual structure of such documents exhibits significant variations across collections for several reasons: the schema is very lax (e.g., RSS feeds), the schema is large and different subsets are used (e.g., industry standards like UBL), or open content models allow arbitrary schemas to be mixed (e.g., RSS extensions like those used for podcasting). Many web development tasks that incorporate XPath queries to process XML documents require an understanding of the actual structure present in the collection.

This paper introduces the unique capabilities of AxPRE summaries *for exploring the (semi-)structure of large XML collections. AxPRE summaries are implemented in a tool, DescribeX, that supports visualizing XML collections via summaries that can be interactively* refined *using a powerful and descriptive* axis path regular expression *language. Experimental results on gigabyte collections valig that flexibility does not come at the expense of efficiency.*

1 Introduction

XML continues to be widely used as a common format for web accessible data as well as for data exchanged among web applications (using web services or a simple REST style transfer). Compared to the earlier wild web days of abundant (not even well-formed) HTML, there is a clear trend toward applications that validate XML documents against schemas. However, despite schema-validity, the actual structure present in web documents exhibits significant variations across collections for several reasons.

First, the schemas used can be very lax (e.g., by extensive use of the `<xsd:choice>` construct in XML schema[1]). This is the case for RSS feeds (a format used by content distributors to deliver to subscribers frequently updated content over the Web). Second, a schema can be very large and only subsets are actually used in a given instance. This is the situation with several industry specific standards that contain hundreds of elements (such as UBL[2] or HR-XML[3]). Finally, a schema can be extended by incorporating elements from other namespaces and corresponding schemas (by using the `<xsd:any>` XML Schema construct to allow open content models). A wide variety of industry standards (like RSS, UBL and HR-XML) adopt open content models as an extensibility mechanism, enabling different user communities to pick and choose how to combine schemas.

Many web development tasks resort to XPath [23] queries to process XML documents and require an understanding of the actual structure present in the collection. Understanding the actual structure of a web collection can be a significant barrier to write meaningful XPath queries. Similar challenges are faced by applications issuing XPath queries to process a collection of feeds that incorporates RSS extensions[4] supporting additional elements for describing pictures, podcasts, and videos.

This paper addresses the need to describe the actual metadata structure of large collections of web documents (by and large encoded and processed as XML). We propose a novel approach for flexibly summarizing the structure of metadata actually present in a collection [9]. The proposed framework is implemented in DescribeX, a tool (demonstrated in [1]) for describing and visualizing XML collections via summaries that can be tailored using a powerful language: *axis path regular expressions* (*AxPRE*, for short).

XML structural summaries are graphs representing relationships between sets of XML elements with a common structure (paths, subtrees, etc.). Describing metadata in semistructured collections was a major motivation in one of the earliest summary proposals in the literature [18]. Since then, research on summaries has focused on query processing, making summaries one of the most studied techniques for query evaluation and indexing in XML (and other

---

*Technical Report posted June 2007 (revised November 2007).

[1] `http://w3.org/TR/xmlschema-1`

[2] `http://oasis-open.org/committees/ubl/`

[3] `http://hr-xml.org`

[4] `http://rss-extensions.org`

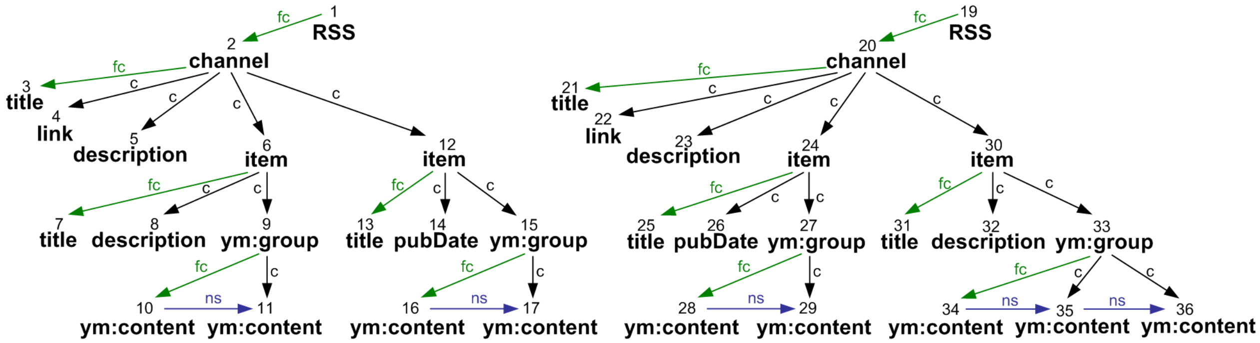

Figure 1. The axis graph of two sample RSS feeds

semistructured) data models [13, 17, 15, 6, 16, 21, 4], as well as for providing statistics useful in XML query estimation and optimization [20]. Although these are all interesting problems we can address (see [8] for XPath query evaluation using DescribeX), in this paper we focus on metadata exploration, which has become increasingly relevant over the last few years and has received considerably less attention in the summary literature. *AxPRE summaries* have a unique capability that makes them suitable for describing the (semi-)structure of XML collections: they are the first summaries capable of *refining and describing* the nature of the partitions created using a powerful language.

1.1 Motivating Example

We motivate our work by describing the tasks of a developer, Sue, who has to code a web application that retrieves RSS feeds from several content providers to produce an aggregated meta feed.

Figure 1 shows two RSS feed instances represented as axis graphs [8]. An axis graph is an abstract representation of the XPath data model [23] extended with edges that represent binary relations between elements. Selected axes are shown in the figure: fc, ns, and c (shorthands for $firstchild$, $nextsibling$, and $child$, respectively). Note that an axis graph can include binary relations between elements and/or attributes that are not XPath axes per se, like fc and ns, *id-idrefs*, or other binary relations. The two feeds make use of the Media RSS extension[5] providing support for media syndication (the elements in this extension use the namespace prefix `media`), which we abbreviated by `ym`). Several `<ym:content>` elements appear in a `<ym:group>` within `<item>`.

Sue has access to a repository containing several months of sample feeds published by the content providers (a collection with thousands of XML files). She is planning to prototype and then refine an application using XPath patterns based on the samples. However, to write the required XPath expressions Sue has to understand the structure of the feed collection. Sue could manually open a few files to get a sense of the structure of the entire collection. At some point Sue will have to check if the patterns she has selected are indeed characteristic of the collection. She could use XPath queries for this task, but she will have to come up with large number of queries on her own.

Fortunately, Sue has access to the DescribeX tool for exploring the structure of the entire feed collection. The tool can process a collection with thousands of files in a minute to provide a first glimpse of the collection's structure using a label *summary descriptor* (SD, for short), the simplest of the AxPRE summaries created by DescribeX. The label SD in Figure 2, created from the two feeds in Figure 1, partitions the elements in the collection by label (element name in this case). For example, SD node s_6 groups all the item elements in the two documents with the actual element numbers listed below the node (this set is called the *extent* of the SD node s_6). An SD edge is labeled by the axis relation it represents (i.e. edge (s_6, s_5) is labeled by c, which means that there is a c axis relation between elements in the extent of s_6 and s_5). Figure 2 shows three kinds of edges, depending on properties of the partitions that participate in the axis relation: dashed, regular, and bold (described in Section 3).

Sue needs to characterize feed items that have different structure in order to process them differently (and to select them using different XPath expressions). For instance, she will aggregate differently items that provide a publishing date (in a pubDate element) from those that do not, and also items that contain media in different formats (i.e., in separate content elements) from those that come with a single media file. From the label SD of Figure 2 she only knows that an item may contain any combination of title, pubDate, group and description, and that a group may have several content sub-elements. To continue exploring items, Sue uses DescribeX to interactively *refine* the SD node s_6 in Figure 2, creating the three nodes s_{61}, s_{62}, s_{63} in a new SD shown in Figure 3. This creates a more refined parti-

---

[5] `http://search.yahoo.com/mrss`

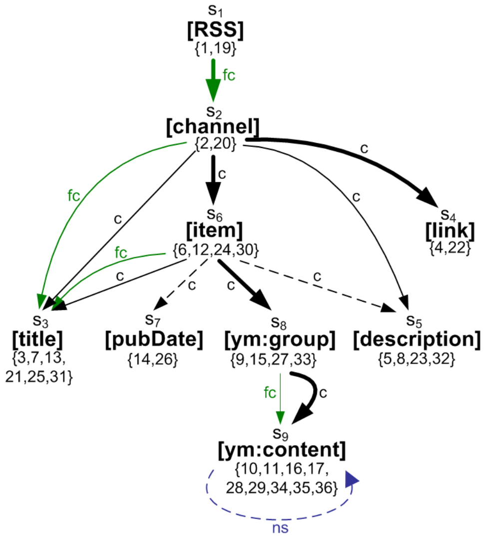

Figure 2. Label SD for the RSS samples

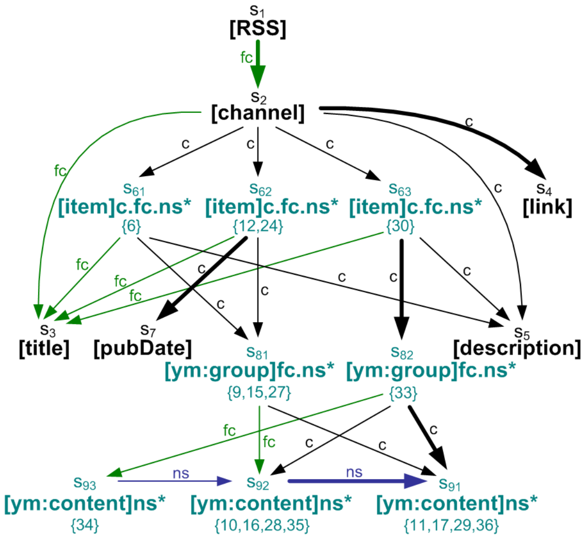

Figure 3. A refined SD for the RSS samples

tion of the item elements in the extent associated with s_6 that further describes the structure of items. The refinement is based on bisimilarity applied to neighbourhoods of nodes described by an AxPRE, a path regular expression on binary relations. AxPREs can specify more complex refinements, like ns^* in SD nodes s_{93}, s_{94} and s_{95} of Figure 3, which distinguishes `[ym:content]` elements based on the labels and number of their following siblings. Sue can either choose a refinement from a list suggested by DescribeX containing the most common ones (e.g. p^* for label paths from the root, c^* for label paths to the leaves, $fc.ns^*$ for sequence of child labels) or she can write her own AxPRE. (AxPREs and refinements are described in detail in Sections 2 and 3.)

Sue can distinguish three kinds of items with different structure beyond the elements directly contained by item, a capability not available using DTD's (unless item elements are renamed, which is not a possibility when the original DTD or the instances cannot be modified). In particular, proposals to infer a DTD from an instance (such as [3, 12]) by suggesting (general, but succinct) regular expression from the strings of child elements, do not help to identify the three kinds of items as done above. For instance, the DTD expression *tile,(description|pubDate),ym:group* can be inferred for the item elements occurring in the instances shown in Figure 1. However, a DTD can only give a rule for the children of items, there is no mechanism for giving rules relating items to their grandchildren (or any other elements farther away). In contrast, the AxPRE summary in Figure 3 can represent that the items with three ym:content grandchildren (node s_{63}) are also the items with a description (but not a pubDate).

1.2 Contributions and Organization

This paper identifies the growing need for describing the structure of web collections (encoded in XML) using mechanisms that go beyond providing one or more schemas. We advocate the use of highly customizable summaries that represent the actual structure of metadata labels as used in a given collection. The summary labels can mix data and metadata (e.g., an XML element with a given attribute value).

The next section presents the background definitions for AxPRE summaries, which are specified by a partition created using the novel notion of bisimilarity applied to subgraphs described by an AxPRE (a path regular expression on binary relations, XPath axis in particular). Section 3 presents a key contribution: a rich framework for *refining AxPRE summaries* that are capable of describing the criteria used to create refined partitions.

Another major contribution appears in Section 4: the implementation of DescribeX, a tool for interactively creating and refining an AxPRE summaries given large collections of XML documents. Two refinement strategies are considered; one based on materialized partitions, and the other based on a virtual approach to compute extents using XPath expressions derived from the AxPRE summary. Section 5 provides experimental results, using gigabyte size XML collections, that validate the performance of the techniques employed by DescribeX. Before concluding, Section 6 provides a comprehensive description of how AxPRE Summaries relate to (and significantly extend) the extensive literature on summaries.

2 AxPRE Summaries Background

This section provides an overview of the DescribeX framework (introduced in [8]). The framework includes a powerful language based on *axis path regular expressions* (AxPREs) for describing the extents in an SD. Extents are defined using a novel notion: bisimilarity applied to neighbourhoods of nodes described by an AxPRE (a path regular expression on binary relations). For representing an XML instance, DescribeX uses a labeled graph model called *axis graph*.

Definition 2.1 (Axis Graph) *An axis graph* $\mathcal{A} = (Inst, Axes, Label, \lambda)$ *is a structure where* $Inst$ *is a set of nodes,* $Axes$ *is a set of binary relations* $\{E_1^{\mathcal{A}}, \ldots, E_n^{\mathcal{A}}\}$ *in* $Inst \times Inst$ *and their inverses,* $Label$ *is a finite set of node names, and* λ *is a function that assigns labels in* $Label$ *to nodes in* $Inst$. *Edges are labeled by axis names and nodes are labeled by element or attribute names (including namespaces), or by new labels defined using XPath.*

An axis graph is an abstract representation of the XPath data model [23] extended with edges that represent XPath relations between elements.

The axis graph can also include additional axes, such as $id\text{-}idrefs$, fc, ns, and other binary relations between elements or attibutes that can be expressed in XPath (e.g., $fc := child :: *[1]$ and $ns := following\text{-}sibling :: *[1]$).

Example 2.1 *DescribeX provides significant flexibility for summarizing combinations of XML data and metadata. A new node label [ym:quicktime] can be defined in an axis graph by the XPath expression* `ym:content[type="video/quicktime"]`*, which represents ym:content elements with different types of media as separate nodes.*

DescribeX uses AxPREs for characterizing the sets in the partition of elements. An AxPRE gives a precise description of the elements in the extent of an SD node, something not provided by any other proposal in the literature.

Definition 2.2 (Axis Path Regular Expressions) *An* axis path regular expression *is an expression generated by the grammar*

$$E \longleftarrow axis \mid axis[B(l)] \mid (E \mid E) \mid (E)^* \mid E.E \mid \epsilon$$

where $axis \in Axes$ *and* ϵ *is the symbol representing the empty expression.*

Definition 2.2 describes the syntax of path regular expressions on the binary relations (labeled edges) of the axis graph including node label tests ($B(l)$ is a boolean function on a label $l \in Label$ that supports more elaborate tests on labels, beyond just matching it). AxPREs could also be written using a syntax closer to XPath syntax.

AxPREs are used in DescribeX for defining neighbourhoods of nodes computed by intersecting the automaton of the AxPRE and the axis graph starting from a given node.

Definition 2.3 (AxPRE Neighbourhood of v) *Let* $\mathcal{A}$ *be an axis graph,* v *a node in* $\mathcal{A}$, α *an AxPRE, and* $NFA(\alpha)$ *the Thompson construction finite automaton of* α *accepting all prefixes. The* AxPRE neighbourhood *of* v *for* α, *denoted* $\mathcal{N}_\alpha(v)$, *is the subgraph of* $\mathcal{A}$ *product of the intersection between* $\mathcal{A}$ *and* $NFA(\alpha)$ *such that* v *intersects with the initial state of* $NFA(\alpha)$.

This approach for defining summaries is based on the intuition that nodes that have similar neighbourhoods should be grouped together in the same extent. DescribeX uses the similarity notion of *labeled bisimulation*, which provides a way of computing a double homomorphism between graphs.

Definition 2.4 (Labeled Bisimulation) *Let* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *be two subgraphs of an axis graph* $\mathcal{A}$, *such that* $Axes_{\mathcal{G}_1} \subseteq Axes$ *and* $Axes_{\mathcal{G}_2} \subseteq Axes$. *A labeled bisimulation between* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *is a symmetric relation* $\approx$ *such that for all* $v \in \mathcal{G}_1$, $w \in \mathcal{G}_2$, $E_i^{\mathcal{G}_1} \in Axes_{\mathcal{G}_1}$, *and* $E_i^{\mathcal{G}_2} \in Axes_{\mathcal{G}_2}$: *if* $v \approx w$, *then* $\lambda(v) = \lambda(w)$; *if* $v \approx w$, *and* $\langle v, v' \rangle \in E_i^{\mathcal{G}_1}$, *then* $\langle w, w' \rangle \in E_i^{\mathcal{G}_2}$ *and* $v' \approx w'$.

Example 2.2 *Consider elements* 12 *and* 24 *in the axis graph of Figure 1. They have bisimilar* $[item]c.fc.ns^*$ *neighbourhoods and therefore belong to the same set in the partition (the one that corresponds to* s_{62} *in Figure 3).*

The widespread use of bisimulation in summaries is motivated by its relatively low computational complexity properties. The bisimulation reduction of a labelled graph can be done in time $O(m \log m)$ (where m is the number of edges in a labelled graph) as shown in [19], or even linearly for acyclic graphs, as shown in [11]. Using bisimulation also allows us to capture all the existing bisimulation-based proposals in the literature (Section 6).

Definition 2.5 (AxPRE Partition) *Let* $\mathcal{A}$ *be an axis graph,* $N \subseteq Inst$, *and* α *an AxPRE. An* AxPRE partition *of* N *for* α, *denoted* $\mathcal{P}_\alpha(N)$, *is a partition of the nodes in* N *defined as follows: two nodes* $v, w \in N$ *belong to the same class* $P_i \in \mathcal{P}_\alpha(N)$ *iff there exists a* labeled bisimulation $\approx$ *between* $\mathcal{N}_\alpha(v)$ *and* $\mathcal{N}_\alpha(w)$ *such that* $v \approx w$.

Definition 2.6 (Summary Descriptor) *A* summary descriptor *(SD for short) of an axis graph* $\mathcal{A}$ *consists of a partition* $\{N_i\}_i$ *of* $\mathcal{A}$ *and a set of AxPRE partitions* $\{\mathcal{P}_{\alpha_i}(N_i)\}_i$, $1 \leq i \leq m$, *together with a labeled graph* G, *called* SD graph, *representing axis relationships between* *elements in the equivalence classes of the AxPRE partitions. Each node s in the SD graph has associated one set in some $\mathcal{P}_{\alpha_i}(N_i)$ called the* extent *of s and is labeled by α_i. Edges are labeled by the $axis$ relation they represent.*

When the extents of all nodes in a SD $\mathcal{D}$ are defined with the same AxPRE α we have an *homogeneous* SD. In this case we say that $\mathcal{D}$ is an α SD. In contrast, if at least two different nodes are defined with different AxPREs we have an *heterogeneous* SD.

3 Summary Refinements

The description provided by a node in the SD can be changed by an operation that modifies its AxPRE and thus its AxPRE neighbourhood. This operation is called a *refinement* of an SD node.

Previous proposals perform global refinements on the entire SD graph [15, 16] or local refinements based on statistics and/or workload [21, 14, 20], without the ability to refine a clearly defined neighbourhood. In contrast, we can precisely characterize the neighbourhood considered for the refinement with an AxPRE.

DescribeX refinements are based on the notion of *summary axis stability*.

Definition 3.1 (Summary Axis Stability) *Let $e = \langle s_i, s_j \rangle$ be an SD graph edge with label $axis$. We say that e is either an* existential *edge iff $\exists x \in extent(s_i), \exists y \in extent(s_j) \wedge \langle x, y\rangle \in axis$, or a* forward-stable *edge iff $\forall x \in extent(s_i), \exists y \in extent(s_j) \wedge \langle x, y\rangle \in axis$.*

Definition 3.1 captures the relationship between edges in the SD graph and the axis graph, and generalizes to several axis the edge stability representation in XSketch [20]. Note that all forward-stable edges are also existential. In Figures 2 and 3, existential edges are represented by dashed lines and forward-stable edges by solid lines. A dashed line does not necessarily mean that an edge is not forward-stable, it might be that stability has not been checked on that edge (existential edges in Figures 2 and 3 have been checked and are not forward-stable). When an edge e and its inverse are both forward-stable, e is shown in bold lines.

The notion of refinement [19] is well-known in the XML summary literature. The goal of our refinement operation is to make all edges of a neighbourhood, given by an AxPRE in the SD graph, forward-stable. For that, we need the notion of an AxPRE neighbourhood defined for an SD graph rather than an axis graph. This notion is called *summary neighbourhood*.

Definition 3.2 (Summary Neighbourhood) *Let $\mathcal{D}$ be an SD, s a node in $\mathcal{D}$, α an AxPRE, and NFA(α) the Thompson construction finite automaton of α accepting all prefixes. The* summary neighbourhood *of s for α, denoted $\mathcal{N}_\alpha(s)$, is the subgraph of $\mathcal{D}$ product of the intersection between $\mathcal{D}$ and NFA(α) such that s intersects with the initial state of NFA(α).*

If all edges in the α neighbourhood of SD node s are forward-stable, then α describes in fact the extent of s. Similarly, by following forward-stable edges we can construct an AxPRE that provides a more detailed description of the extent of s.

Example 3.1 *Consider node s_6 in Figure 2. Although its current AxPRE is $[item]$, which means that its extent contains only item elements, it is possible to infer from the SD graph a more "detailed" AxPRE. Since edges $\langle s_6, s_3\rangle$, $\langle s_6, s_8\rangle$, and $\langle s_8, s_9\rangle$ are forward-stable, we could write an AxPRE that expresses those relations, which is $[item].(c[title]|c[ym:group].c[ym:content])$. Such an AxPRE tells us that not only the extent of s_6 contains item elements, but more precisely they also have title and ym:group elements with nested ym:content elements.*

Given an SD node s and an AxPRE α, Algorithm 3.3 computes an AxPRE partition of the extent of s for α that is a refinement of the extent of s. This is achieved by stabilizing the α neighbourhood of s.

In order to stabilize a single edge, Algorithm 3.3 invokes Algorithm 3.1, for different nodes, and Algorithm 3.2, for the same node (loop).

Algorithm 3.1 (Edge Stabilization)

```
stabilizeEdge($sd, s_i, s_j$)
Input: An SD sd containing a non forward-stable edge $e = \langle s_i, s_j\rangle$ with label axis
Output: An SD sd where e has been replaced by forward-stable $e' = \langle s'_i, s_j\rangle$.
1: create new nodes $s'_i$ and $s''_i$
2: $extent(s'_i) := \{x \in extent(s_i) \mid \exists y \in extent(s_j) \wedge \langle x, y\rangle \in axis\}$
3: $extent(s''_i) := extent(s_i) - extent(s'_i)$
4: $axpre(s'_i) := axpre(s_i)|axis[\lambda(s_j)].axpre(s_j)$
5: $axpre(s''_i) := axpre(s_i)|axis[\neg\lambda(s_j)]$.
6: create an edge $e' = \langle s'_i, s_j\rangle$
7: $addEdges(s_i, \{s'_i, s''_i\})$
8: delete node $s_i$, and all its incoming and outgoing edges
```

While the definitions of the extents of s'_i and s''_i are similar to split [19] and B_Stabilize [20], the main novelty here is the ability to maintain an AxPRE characterizing the extents (lines 4 and 5).

Function $addEdge(s_i, S)$ in Algorithm 3.1 simply checks whether there are axis relations between nodes in S and all SD nodes related to s_i by an edge, adding edges when they are either existential or forward-stable.

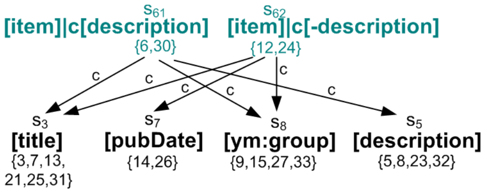

Figure 4. The c neighbourhood of s_6 from Figure 2 after stabilizing edge $\langle s_6, s_5\rangle$

Example 3.2 *Consider edge $\langle s_6, s_5\rangle$ from Figure 2. This edge is not forward-stable because elements 12 and 24 are not related to any node in $extent(s_5)$ via the c axis (i.e. there is no c edge from either 12 or 24 to a description element in Figure 1). Edge stabilization will create two nodes, s_{61} and s_{62}, such that $extent(s_{61}) = \{6, 30\}$ and $extent(s_{62}) = \{12, 24\}$. The new AxPRE of s_{61} is $[item]|c[description]$ and of s_{62} $[item]|c[\neg description]$. The new edge $\langle s_{61}, s_5\rangle$ is forward-stable. The result of stabilizing edge $\langle s_6, s_5\rangle$ is shown in Figure 4.*

Algorithm 3.2 (Edge Unfolding)

unfoldEdge$(sd, s, axis)$

Input: *An SD sd, a node s such that there exists $e = \langle s, s\rangle$ with label $axis$ and e is not forward-stable*
Output: *The SD sd where there is no edge $e = \langle s, s\rangle$ with label $axis$*

```
$k = 1$
while there is an $axis$ relationship in $extent(s)$ - i.e., $e = \langle s, s\rangle$ with label $axis$ is valid do
    create a new node $s_k$
    $extent(s_k) := \{x \in extent(s) \mid \nexists y \in extent(s) : \langle x, y\rangle \in axis\}$
    $extent(s) := extent(s) - extent(s_k)$
    create an edge $e_k = \langle s, s_k\rangle$ with label $axis$
    if $k = 1$ then
        $axpre(s_k) := \epsilon$
    else
        $axpre(s_k) := axpre(s).axpre(s_{k-1})$
        create an edge $e'_k = \langle s_k, s_{k-1}\rangle$ with label $axis$
        delete the edge $\langle s, s_{k-1}\rangle$
    $k := k + 1$
rename $s$ as $s_k$
$addEdges(s, \{s_1, \ldots, s_n\})$
```

Proposition 3.1 *All edges e_k and e'_k produced by Algorithm 3.2 are forward-stable.*

Example 3.3 *Consider edge $\langle s_9, s_9\rangle$ from Figure 2. The edge is not forward-stable because some element in $extent(s_9)$ is not in a ns relation with elements in the same extent (for instance, there is no element that is the next sibling of 36 in Figure 1). The first iteration of edge unfolding creates a new node, s_{91}, such that $extent(s_{91}) = \{11, 17, 29, 36\}$ and $extent(s_9) = \{10, 16, 28, 34, 35\}$, and adds a new edge $\langle s_9, s_{91}\rangle$ with label ns. Since $\langle 34, 35\rangle \in ns$, there is still a ns loop on node s_9 and the algorithm continues. The second iteration creates a new node, s_{92}, such that $extent(s_{92}) = \{10, 16, 28, 35\}$ and $extent(s_9) = \{34\}$, adds new edges $\langle s_9, s_{92}\rangle$ and $\langle s_{92}, s_{91}\rangle$ with label ns, and deletes edge $\langle s_9, s_{91}\rangle$. Since there is now no ns loop on node s_9, the algorithm renames the node to s_{93} and the corresponding edges, and ends. The new edges $\langle s_{93}, s_{92}\rangle$ and $\langle s_{92}, s_{91}\rangle$ are forward-stable. The resulting nodes and extents are shown in Figure 3.*

The AxPRE of an SD node can be "generalized" when the neighbourhood of the node satisfies certain conditions. We define this notion of *AxPRE generalization* next.

Proposition 3.2 (AxPRE Generalization) *Given an SD s and an AxPRE α for s, if α contains a subexpression $\alpha_{axis} = axis[l_1]| \ldots |axis[l_n]$ and $l_1 \ldots l_n$ are the labels of all nodes matched by $axis$ on forward-stable edges, then α_{axis} can be replaced by $\alpha'_{axis} = axis$. The new AxPRE thus obtained expresses the same neighbourhood of s as α.*

Example 3.4 *Consider node s_{61} from Figure 4. The detailed AxPRE (see Example 3.1) for s_{61}'s c neighbourhood is $[item]|c[description]|c[title]|c[ym : group]$. Since such AxPRE satisfies Proposition 3.2, it can be replaced by $[item]|c$.*

We have now all the building blocks for introducing the Neighbourhood Stabilization Algorithm, which computes a refinement of the extent of an SD node s for an AxPRE α.

Algorithm 3.3 (Neighbourhood Stabilization)

StabilizeNeighbourhood(sd, α, s)

Input: *An SD sd, an AxPRE α, and a node s*
Output: *An SD where all the edges in the α neighborhood of s are forward-stable*

```
compute the $\alpha$ neighbourhood of $s$
$S = \{s' \mid s'$ is in the $\alpha$ neighbourhood of $s\}$
while $S \neq \emptyset$ do
    pick a node $s'$ in $S$ such that $s'$ is at the end of the longest simple path from $s$
    for each edge $e = \langle s', s'\rangle$ do
        $unfoldEdge(sd, s', axis)$
    for each edge $e = \langle s'', s'\rangle$ do
        $stabilizeEdge(sd, s', s'')$
    remove $s'$ from $S$
```

Example 3.5 *Consider node s_6 in Figure 2 and suppose we want to refine it with AxPRE $[item]c.fc.ns^*$. First, Algorithm 3.3 will find the $[item]c.fc.ns^*$ neighbourhood of s_6, which consists of the c edges from s_6, the fc edge from s_8 and the ns edge from s_9 (Figure 2). Then, it unfolds edge $\langle s_9, s_9\rangle$ labeled ns, as described in Example 3.3. Finally, it stabilizes edge $\langle s_6, s_5\rangle$, as described in Example*

3.2. Figure 3 shows the resulting SD after applying AxPRE generalization (Proposition 3.2).

4 DescribeX Implementation

In this section we discuss how the framework introduced in Sections 2 and 3 is implemented in our tool, DescribeX. The tool is geared to support the interactive creation and refinement of AxPRE SDs for large collections of XML documents. We present two strategies for refining an SD: one is based on materializing the SD partitions, the other utilizes a novel virtual approach that relies on constructing XPath expressions that compute extents.

We implemented DescribeX in Java using Berkeley DB Java Edition to store and manage indexed collections (tables). The DescribeX System architecture is tailored to process XML collections one file at a time, the prevalent data processing model for the Web. Each file is parsed, processed and stored before continuing with the next file in the collection. The DescribeX System can invoke an arbitrary XPath processor for the evaluation of XPath expressions. For the experiments reported later in this paper, the Saxon [6] XPath processor was employed.

In the DescribeX System, the extents are stored in an indexed table named `elemDB` that has schema `elemDB(`<u>`SID`</u>`,` <u>`docID`</u>`,` <u>`endPos`</u>`, startPos, SID2)`, where the underlined attributes are the key (also used for indexing). The `elemDB` table contains a tuple for each XML element in the collection. Each SD node is identified by a unique id called SID. Each element belongs to the extent of a unique SD node, whose SID is stored in the `SID` attribute. The attribute `docID` holds the identifier of the document in which the element appears. The `startPos` and `endPos` are the position, in the document, where the element starts and ends, respectively. `SID2` allows us to maintain an SID for more than one SD. The SD graph is kept in main memory in separate hash tables for each axis relation in the SD.

Alternatively, the user can decide to keep the extents virtual and thus having a `docDB` table instead of the `elemDB` table described above. The schema of `docDB` is `elemDB(`<u>`SID`</u>`,` <u>`docID`</u>`)`, which contains for each sid s the docIDs of all XML documents containing elements in the extent of s. This can be used to efficiently locate the XML documents to be evaluated by the *extent expression* (EE for short) of s in order to get the extent of s.

The SD graph is kept in main memory in separate hash tables for each axis relation in the SD, e.g. the `parentsMap` and `childrenMap` maps contain the edge definitions for the p and c SD axes respectively. In other words, each binary $axis$ relation is stored as a map between a key SID s and a set of SIDs $s_1, \ldots, s_n$ such that $\langle s, s_i \rangle \in axis$, $1 \leq i \leq n$. In addition, there is a label map, `labelMap`, that contains the label of each SD node. Finally, there is an XML file that stores the EE expressions.

<u>*buildP*(k)</u>

Input: Collection C of XML documents
Output: p^k SD

```
for each XML document $doc$ in collection $C$ do
    assign a new docID $d$ to $doc$
    create a new DOM tree $t$
    while parsing $doc$ do
        if element start tag is found in $doc$ then
            create a new $e$ in $t$ with XML attributes $sid$, $startPos$, and $endPos$ set to empty
            if the $p^k$ neighbourhood of $e$ is not in the SD graph then
                create a new SID $s'$
                update labelMap, parentsMap, and childrenMap
                store the $p^k$ XPath expression of $s'$ in the EE XML file
            get the sid $s$ of $e$ from the SD
            set $e.sid$ to $s$ and $e.startPos$ to the offset position of the start tag of the element
        if element end tag is found in $doc$ then
            set $e.endPos$ to the offset position of the end tag of the element
            append tuple $(e.s, d, e.endPos, e.startPos)$ to elemDB
```

Figure 5. p^k **SD construction**

We describe next the construction of the initial SDs performed in one-pass over the collection. After that, we show how to implement the refinements presented in Section 3 using the XPath processor and DescribeX data structures [7].

4.1 Initial SD Construction

Some SDs can be constructed in one-pass over the collection. This is possible when the parsing information collected at either the start tag or the end tag of an element v is enough to construct the AxPRE neighbourhood $\mathcal{N}_\alpha(v)$ of the element, compute the AxPRE partition and thus decide what SD extent v belongs to. For instance, the `start` tag itself is enough to classify an element v when constructing the ϵ SD (the $\mathcal{N}_\epsilon(v)$ contains just node v). For the p^k and p^* SDs, it suffices to keep the sequence of the last k open elements (for the p^k) or all of them (for the p^*) for creating $\mathcal{N}_{p^k}(v)$ and $\mathcal{N}_{p^*}(v)$. Thus, p^k and p^* SDs can be constructed in one-pass over the collection.

---

[6] `http://saxon.sourceforge.net/`

[7] Additional implementation details are given in `http://www.cs.toronto.edu/∼consens/describex/`

Algorithm buildP(k) (Figure 5) illustrates the use of the DescribeX data structures introduced in Section 4. BuildP(k) computes the ϵ, p^k and p^* SDs. The parameter k encodes the SD as follows: $k = 0$ corresponds to ϵ, $k = maxint$ to p^*, and all other values represent p^k. For each XML document in the collection, the algorithm parses the document and creates a DOM tree [8]), which will then be used for updating the SD. The DOM tree and the SD are constructed simultaneously during parsing time.

Once an SD has been constructed from scratch, the user can refine any SD node or set of nodes by changing the node's AxPRE, as described in Section 3.

4.2 Expressing Virtual Extents in XPath

In DescribeX, the extents of any SD node can be precomputed and stored in a data structure. This approach, which we call *materialized extents*, requires to have a pointer to every XML element in the collection and thus can be very space consuming. A more space-efficient approach is to keep an EE that represents all elements in the extent of a given SD node s, denoted $ee(s)$. In these *virtual extents*, the evaluation of the $ee(s)$ returns the actual extents of s. DescribeX virtual extents are a compact representation of the extents, similar to the concept of virtual view with the addition of the non-monotonic property.

We discuss next how to construct the EEs. Since an EE computes the extent of an SD node s in an axis graph regardless of any variable context, they are of the form $ee(s) = /\mathit{descendant\text{-}or\text{-}self} :: l/locpath$, where l is an element label and $locpath$ is the remainder of the EE. We call the $\mathit{self} :: l/locpath$ subexpression the *relative extent expression* (REE, for short) of s, denoted $ree(s)$. The EE of s can always be constructed from the REE of s because $ee(s) = //ree(s)$ in XPath abbreviated syntax.

Each AxPRE α of an SD node s created by Algorithms 3.1 and 3.2 has an equivalent EE e_s that can compute the extent of s. Such EEs can be created by adding lines $4' : ree(s'_i) = ree(s_i)[axis :: * [ree(s_j)]][count(axis :: *) = count(axis :: *[ree(s_j)]]$ and $5' : ree(s''_i) = ree(s_i)$ $[count(axis :: *[ree(s_j)]) = 0]$ to Algorithm 3.1, and lines $8' : ree(s_i) = ree(s)[count(axis :: *) = 0]$ and $10' : ree(s_i) = ree(s)[axis :: *[ree(s_{i-1})]]$ $[count(axis :: *) = count(axis :: *[ree(s_{i-1})]]$ to Algorithm 3.2. The count predicates are in the EEs to make sure that all nodes in the answer of $ee(s'_i)$ satisfy *only* the $[axis :: *[ree(s_j)]]$ predicate, and all nodes in the answer $ee(s''_i)$ are not in the answer of $ee(s'_i)$. This guarantees that $ee(s'_i)$ and $ee(s''_i)$ compute a partition of the nodes in the answer of $ee(s_i)$.

Example 4.1 *Consider edge* $\langle s_6, s_5\rangle$ *from Figure 2, which is not forward-stable. Edge stabilization will create two nodes,* s_{61} *and* s_{62} *from Figure 4. Given that* $ree(s_5) = \mathit{self} :: \mathit{description}$, *and the stabilized edge corresponds to a c axis,* $ree(s_{61}) = \mathit{self} :: \mathit{item}[\mathit{child} :: *[\mathit{self} :: \mathit{description}]]$ $[\mathit{count}(\mathit{child} :: *) = \mathit{count}(\mathit{child} :: *[\mathit{self} :: \mathit{description}])]$ *and* $ree(s_{62}) = \mathit{self} :: \mathit{item}[\mathit{count}(\mathit{child} :: *[\mathit{self} :: \mathit{description}]) = 0]$.

```
refineVirtual($sd, s, r_1, \ldots, r_n, extent$)
Input: $sd$ is the SD, $s$ is the sid to be refined, $r_1 \ldots r_n$ is a family of refining XPath EEs
Output: Updated $sd$, $extent$ with the element in the extent of $s_i$
1: get the XPath EE $e_s$ of $s$
2: for each input $r_i$ do
3:     create a new sid $s_i$
4:     for each $d$ s.t. there is a tuple $t_d$ in docDB with $t_d.SID = s$ and $t_d.docID = d$ do
5:         create a DOM tree $t$ of $d$ in which each element has an $endPos$ attribute with the offset position of the end tag of the element
6:         assign to $extent$ the answer of $/e_s/r_i$
7:         update labelMap by assigning the label of $s$ to the new $s_i$
8:         store the $r_i$ XPath expression of $s_i$ in the EE XML file
9:         for each $axis$ in $sd$ do
10:            call computeEdgeByXPath($sd, axis, s_i, extent, s$) to test the existence of an $axis$ edge from $s_i$
```

Figure 6. Refine virtual extents

Since each AxPRE refinement generates several EEs, one for each new SD node to be created by the refinement, computing a refinement involves evaluating a wide range of different EEs.

4.3 Computing Refinements

Following the materialized extents approach the refinement can be evaluated with Algorithm refineMaterialized (Figure 8), whereas virtual extents can be refined by Algorithm refineVirtual (Figure 6). Both algorithms are invoked with sid s to be refined, its current EE e_s, and a family $r_1 \ldots r_n$ of refining EEs, constructed as described in Section 4.2.

Suppose that SD node s_i with EE r_i is one of the refinements of SD node s with EE e_s. The extent of s_i is computed by evaluating r_i on the set of documents that contain elements in the extent of s, which entails evaluating the expression $/e_s/r_i$ (line 6 of algorithms in Figures 6 and 8). This set of documents are obtained from `ElemDB` (if the

---

[8] `http://www.w3.org/TR/REC-DOM-Level-1`

```
computeEdgeByXPath($sd, axis, s_i, extent, s$)
Input: $sd$ is the SD, $axis$ is the axis edge to be computed, $s_i$ is the new sid, $extent$ is the extent of $s_i$, and $s$ is the sid being refined.
Output: Updated $sd$
1: assign to $candidates$ the set of sids $\{c_1, \ldots, c_n\}$ mapped to $s$ in axisMap
2: for each $c_j$ in $candidates$ do
3:     get the EE $e_j$ of $c_j$ from the EE XML file
4:     evaluate the intersection expression $e = axis :: * \cap e_j$ from $extent$
5:     if the evaluation of $e$ is not empty then
6:         add an axis edge between $s_i$ and $c_j$ to the corresponding axisMap
```

Figure 7. Compute edges with XPath

extent of s is materialized) or from `docDB` (if the extent of s is virtual). Once we have the extent of s_i, the edges in the SD graph can be constructed either from the EE when the extent is virtual (by computeEdgeByXPath, line 10 of Algorithm refineVirtual) or from `ElemDB` when the extent is materialized (by computeEdgeByMerge, line 13 of Algorithm refineMaterialized).

In order to update the edges we need to check whether there is an $axis$ edge between s_i and a set of candidate SD nodes $c_1, \ldots, c_n$ such that $\langle s, c_j \rangle \in axis$. This is performed by Algorithm computeEdgeByXPath (Figure 7) by computing the expression $e_{s_r}/axis :: * \cap e_{c_j}$, where e_{c_j} is the EE of candidate c_j (line 4). If the evaluation of the expression is not empty, then there exists and edge from s_i to c_j, otherwise there is no edge (lines 5 and 6).

Algorithm computeEdgeByMerge (not shown in the Figures), in contrast, simply computes a merge of the `ElemDB` using the `startPos` and `endPos` attributes to check for containment and precedence, depending on the $axis$ edge being computed.

5 Experimental Results

In this section we report running times of both SD construction and AxPRE refinements. We conducted five separate runs starting with a cold Java Virtual Machine (JVM), for each query. The best and worst times were ignored and the reported runtime is the average of the remaining three times. The experiments were carried on a Windows XP Virtual Machine running on a 2.4GHz dual Opteron server, and the JVM was allocated 1 GB of RAM.

```
refineMaterialized($sd, s, r_1, \ldots, r_n$)
Input: $sd$ is the SD, $s$ is the sid to be refined, $r_1 \ldots r_n$ is a family of refining XPath EEs
Output: Updated $sd$
1: get the XPath EE $e_s$ of $s$
2: for each input $r_i$ do
3:     create a new sid $s_i$
4:     for each $d$ s.t. there is a tuple $t_d$ in elemDB with $t_d.SID = s$ and $t_d.docID = d$ do
5:         create a DOM tree $t$ of $d$ in which each element has an $endPos$ attribute with the offset position of the end tag of the element
6:         assign to $extent$ the answer of $/e_s/r_i$
7:         for each element $n_j$ in $extent$ do
8:             locate the tuple $t_j$ in the elemDB table corresponding to $n_j$ by using ($s$, $d$, $n_j.endPos$) as a key
9:             assign $s_i$ to tuple $t_j$ by setting $t_j.SID = s_i$
10:        update labelMap by assigning the label of $s$ to the new $s_i$
11:        store the $r_i$ EE of $s_i$ in the EE XML file
12:        for each $axis$ in the $SD$ do
13:            call computeEdgeByMerge($sd, axis, s_i, extent, s$) to test the existence of an $axis$ edge from $s_i$
```

Figure 8. Refine materialized extents

Table 1. Test Collections

Collection	MB	#docs	Load (sec)
Wikipedia10	1050	90000	736
RSS4	420	19200	452

5.1 Comparing Initial SD Construction

Table 1 summarizes the size and number of documents of our test collections, and the load time for the p^* SD (Section 4.1), which includes computing the partitions and storing them in the ElemDB table. The first collection (Wikipedia10) was created from the Wikipedia XML Corpus provided in INEX 2006 [10]. The second collection (RSS4) was obtained by collecting RSS feeds from thousands of different sites.

Table 2 shows comparable results for SD graph construction times between DescribeX and an open-source

Table 2. SD Graph Construction Times (sec)

Collection	Size (MB)	DescribeX	XSum
XMark1	115	17.3	12.8
XMark5	580	60.8	62.2
XMark10	1150	118.1	122.1

Table 3. SD nodes and EEs

SD Node	Extent Expression (EE) for p^* AxPRE
w_1	/article/body/figure
w_2	/article/body/section/section/section/figure
w_3	/article/body/section/p/sub
r_1	/rss/channel/image
r_2	/rss/channel/item/item
r_3	/rss/channel/item/body/blockquote/p

Table 4. Refinement Times (sec)

SD Node	Extent Size		$p^*\mid c^*$			$p^*\mid c.fs$		
	Doc	Elem	#	V	M	#	V	M
w_1	17369	21296	85	67	11.3	6	208	40
w_2	317	687	26	7.9	2.8	2	64	27
w_3	581	2822	8	47	15.9	6	19	8
r_1	3300	3300	15	34	9.9	15	33	11
r_2	6	6	2	0.6	0.3	2	0.7	0.2
r_3	16	158	19	0.3	0.1	6	0.5	0.4

XML summarization tool, XSum [2], which constructs an annotated p^* SD graph (a dataguide). XSum does not store neither the extents nor EEs, it only creates a p^* SD graph. To the best of our knowledge, this is the only structural summarization system publicly available. Moreover, no other work in the extensive literature on summaries [13, 17, 15, 16, 21, 4, 20] reports construction times for their systems.

Since XSum can only summarize individual files, we were not able to test it with our benchmark collections. Thus, we decided to do the comparative evaluation using the XMark benchmark [5], which creates one single file of a chosen size.

5.2 Refinement Times

Table 3 shows the SD nodes we refined and their EEs before the refinement. For instance, w_1 corresponds to the node p^* SD node that has /article/body/figure/ as its EE. Our benchmark queries were designed with scalability in mind: smallest and largest extents and number of documents involved in the AxPRE refinements are at least three orders of magnitude apart (from 6 documents in r_2 to 17369 documents in w_1).

Table 4 reports refinement times for the SD nodes provided in Table 3. For each collection, we compare two AxPRE refinements, each one on three different SD nodes. That is, we picked three different p^* SD nodes from each collection and refine them by $p^*|c^*$ and $p^*|c.fs$. These two refinements were chosen to show the DescribeX's performance with AxPREs involving common axes used throughout the summary literature (e.g. p and c), together with novel axes (e.g. fs).

The number of new SD nodes created by the refinements are reported in the # columns of Table 4. For instance, the refinement $p^*|c.fs$ of w_3 using Algorithm 3.3 partitions w_3 into 6 new SD nodes. The EE of one them is $ee(w_3)[child :: *[ree(s_1)]][count(child :: *) = count(child :: *[ree(s_1)])]$, where $ree(s_1) = self :: emph2[following\text{-}sibling :: *[ree(s_2)]][count(following\text{-}sibling :: *) = count(following\text{-}sibling :: * [ree(s_2)])]$ and $ree(s_2) = self :: emph2 [count(following\text{-}sibling :: *) = 0]$ are the result of Algorithm 3.3 stabilizing the $p^*|c.fs$ neighbourhood of w_3.

We consider two scenarios, one in which the extents are pre-computed and stored in the ElemDB table, and another in which the extents are not materialized and are thus represented by the EEs (see Section 4.2). The average running times per EE evaluated are shown under the **V** columns for the "virtual" extents, and under the **M** columns for the "materialized" extents. The reported average times comprise locating the affected files using the SD, opening them and evaluating the EE.

The time differences between the **V** and **M** columns come from the fact that, for the virtual extents, DescribeX has to evaluate the XPath expression for computing the edges between the new SD nodes. This is more costly than evaluating the edges from the information stored in the ElemDB table.

In contrast, time differences between rows is mainly due to the different number of document and elements on which the EE is evaluated. For instance, the extent of w_1 (/article/body/figure) consists of 21296 elements on 17369 files, and the EEs of both refinements tested have to be evaluated on those files. In general, the refinement times increases proportionally to the number of documents that need to be opened for computing the AxPRE refinement.

Our results show that DescribeX can provide interactive response times (from sub second to just a few seconds) for most refinements, even with Gigabyte size collections. This is compelling evidence that DescribeX can be used in scenarios like the one described in Section 1.1.

6 Related Work

The large number of summaries that have been proposed in recent years clearly establishes the value and usefulness of these structures for describing semistructured data, assisting with query evaluation, helping to index XML data, and providing statistics useful in XML query optimization.

DescribeX summaries can be classified in a lattice that describes a *refinement* relationship between entire summaries. Figure 9 shows a fragment of a DescribeX summary lattice that captures earlier proposals based on the notion of bisimilarity [11]. Each node in the lattice of Figure 9 cor-

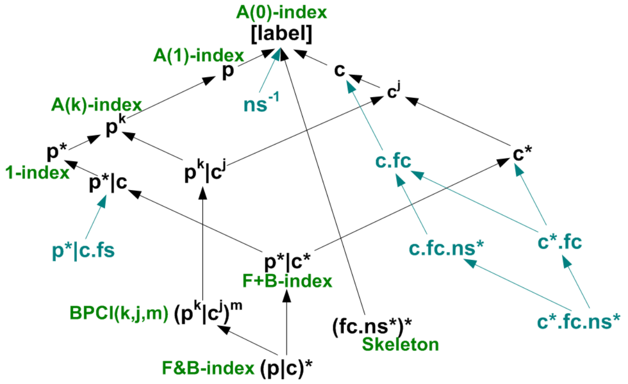

Figure 9. DescribeX lattice capturing earlier homogeneous proposals

responds to a homogeneous SD defined by an AxPRE. The node labels indicated in green are the names of the proposal that each node captures. Nodes and edges in blue are a sample of the richer SDs that were never considered in the literature, like the one that appears in Figure 3 ($c.fc.ns^*$) or in Section 5 ($p^*|c.fs$).

The earliest bisimilarity-based summary proposal is the family presented in [17], which contains a p^* summary: the 1-index. The 1-index partition is computed by using *bisimulation* as equivalence relation. The F&B-Index [15], is an example of a $(p|c)^*$ SD. The F&B-Index construction uses bisimulation like the 1-index, but applied to the edges and their inverses in a recursive procedure until a fix-point. The same work introduces the F+B-index (a $p^*|c^*$ AxPRE summary constructed by applying bisimulation to the edges and their inverses only once) and the BPCI(k,j,m) index (a $(p^k|c^j)^m$ AxPRE summary, where k, and j controls the lengths of the paths and m the iterations of the bisimulation on the edges and their inverses). The A(k)-index [16] is a p^k AxPRE summary based on k-bisimilarity (bisimilarity is computed for paths of lenght k).

There has been almost no activity on summaries that capture the node ordering in the XML tree: the only proposals we are aware of are the earlier region order graphs (ROGs) [7] and the Skeleton summary [4] that clusters together nodes with the same subtree structure. Skeleton uses an entirely different construction approach, but its essence can be captured by the $(fc.ns^*)^*$ AxPRE.

The D(k)-index [21], and M(k)-index [14] are heterogeneous SD proposals. All nodes s_i are described by $\mathcal{N}_d[p^k](s_i)$ with a different k per s_i. They use different construction strategies based on dynamic query workloads and local similarity (i.e. the length of each path depends on its location in the XML instance) to determine the subset of incoming paths to be summarized.

XSketch [20] manages summaries capturing many (but not all) heterogeneous SD's along the p and c axis, ranging from the label summary to the F&B-Index. However there is no control over the refinements chosen, nor a description of the intermediate summaries obtained. This makes sense given that XSketch objective is to provide selectivity estimates, as such its construction algorithm is guided by heuristics to optimize the space/accuracy trade-off.

Other summary proposals are defined without resorting to bisimulation. A number of them are equivalent to bisimulation-based summaries when the data instances are trees. These include Region Inclusion Graphs (RIGs) [7], Representative Objects (RO) [18], strong dataguides [13], and ToXin [22].

Another heterogeneous proposal that uses an ad-hoc construction mechanism is APEX [6], an adaptive path index that summarizes paths that appear frequently in a query workload. The workload considered by APEX are limited to expressions containing a number of $child$ axis composition that may be preceded by a $descendant$ axis, without any predicate.

7 Conclusion

This paper focuses on addressing the need to describe the actual structure of web collections of XML documents using a novel framework (and related tool, DescribeX) to manipulate summaries that can be conveniently tailored using AxPRE expressions. Our main results demonstrate the scalability of AxPRE summary refinements (the key enabler for tailoring summaries) using gigabyte XML collections. There are further opportunities for exploiting the flexibility available in AxPRE-based summaries in the context of traditional summary applications to query evaluation (see [8]), indexing, selectivity estimation, and query optimization.

Familiar research issues can be re-visited in the context of AxPRE summaries; how to give guidelines for selecting good summaries (similar to schema design); or how to infer general and succinct AxPRE expressions from an XML collection (similar to DTD inference from instances). Providing tools for metadata management is also addressed in a very complementary way by a recent schema summarization proposal [24]. A combination that creates summaries that describe how metadata labels (including some generated using schema abstraction and summarization techniques) are used in a given instance seems promising.

Finally, the notion of bisimulation originated in fields other than databases (concurrency theory, verification, modal logic, set theory), where it continues to find applications. It would be interesting to explore whether the more flexible notion used in this paper (selective bisimilarity applied to subgraphs described by AxPREs) can also find novel applications in such areas.

References

[1] M. S. Ali, M. P. Consens, S. Khatchadourian, and F. Rizzolo. DescribeX: interacting with AxPRE summaries. In *ICDE*, 2008.

[2] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. Technical report, INRIA, 2005.

[3] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise DTDs from XML data. In *VLDB*, pages 115–126, 2006.

[4] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large XML repositories. In *ICDE*, pages 261–272, 2005.

[5] R. Busse, M. Carey, D. Florescu, M. Kersten, I. Manolescu, A. Schmidt, and F. Waas. XMark: An XML benchmark project. http://www.xml-benchmark.org/, 2003.

[6] C.-W. Chung, J.-K. Min, and K. Shim. APEX: An adaptive path index for XML data. In *SIGMOD*, pages 121–132, 2002.

[7] M. P. Consens and T. Milo. Optimizing queries on files. In *SIGMOD*, pages 301–312, 1994.

[8] M. P. Consens and F. Rizzolo. Fast answering of XPath query workloads on web collections. In *XSym*, 2007.

[9] M. P. Consens, F. Rizzolo, and A. A. Vaisman. AxPRE summaries: Exploring the (semi-)structure of XML web collections. In *ICDE*, 2008.

[10] L. Denoyer and P. Gallinari. The Wikipedia XML Corpus. *SIGIR Forum*, 2006.

[11] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.

[12] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning document type descriptors from XML document collections. *Data Mining and Knowledge Discovery*, 7(1):23–56, 2003.

[13] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[14] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *ICDE*, pages 683–694, 2004.

[15] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, pages 133–144, 2002.

[16] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[17] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[18] S. Nestorov, J. D. Ullman, J. L. Wiener, and S. S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *ICDE*, pages 79–90, 1997.

[19] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[20] N. Polyzotis and M. N. Garofalakis. XSKETCH synopses for XML data graphs. *ACM TODS*, 31(3):1014–1063, 2006.

[21] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, pages 134–144, 2003.

[22] F. Rizzolo and A. O. Mendelzon. Indexing XML data with ToXin. In *WebDB*, pages 49–54, 2001.

[23] W3C. XPath 2.0. http://www.w3.org/TR/xpath20, 2002.

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