Exploring XML Web Collections with DescribeX

MARIANO P. CONSENS and RENÉE J. MILLER
University of Toronto
FLAVIO RIZZOLO
University of Ottawa and Carleton University
and
ALEJANDRO A. VAISMAN
Universidad de Buenos Aires

As web applications mature and evolve, the nature of the semistructured data that drives these applications also changes. An important trend is the need for increased flexibility in the structure of web documents. Hence, applications cannot rely solely on schemas to provide the complex knowledge needed to visualize, use, query and manage documents. Even when XML web documents are valid with regard to a schema, the actual structure of such documents may exhibit significant variations across collections for several reasons: the schema may be very lax (e.g., RSS feeds), the schema may be large and different subsets of it may be used in different documents (e.g., industry standards like UBL), or open content models may allow arbitrary schemas to be mixed (e.g., RSS extensions like those used for podcasting). For these reasons, many applications that incorporate XPath queries to process a large web document collection require an understanding of the actual structure present in the collection, and not just the schema.

To support modern web applications, we introduce DescribeX, a powerful framework that is capable of describing complex XML summaries of web collections. DescribeX supports the construction of heterogenous summaries that can be declaratively defined and refined by means of axis path regular expression (AxPREs). AxPREs provide the flexibility necessary for declaratively defining complex mappings between instance nodes (in the documents) and summary nodes. These mappings are capable of expressing order and cardinality, among other properties, which can significantly help in the understanding of the structure of large collections of XML documents and enhance the performance of web applications over these collections. DescribeX captures most summary proposals in the literature by providing (for the first time) a common declarative definition for them. Experimental results demonstrate the scalability of DescribeX summary operations (summary creation, as well as refinement and stabilization, two key enablers for tailoring summaries) on multi-gigabyte web collections.

Categories and Subject Descriptors: H.2.5 [**Database Management**]: Heterogeneous Databases

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Semistructured data, XML, structural summaries, XPath

The authors appear in alphabetical order. Contact author: F. Rizzolo, University of Ottawa, SITE, 800 King Edward St., Ottawa, Canada; email: frizzolo@site.uottawa.ca. Additional authors: M. P. Consens, University of Toronto, 40 St. George St., Toronto, Canada; email: consens@mie.utoronto.ca; R. J. Miller, University of Toronto, 40 St. George St., Toronto, Canada; email: miller@cs.toronto.edu; A. A. Vaisman, Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Ciudad Universitaria, Buenos Aires, Argentina; email:avaisman@dc.uba.ar.

1. INTRODUCTION

The web is fueled by *semistructured* data, so called because of the lack of a clear separation between data and metadata. The most common format for web accessible data is XML which is used for hypertext document collections like Wikipedia as well as for data exchange among web applications (e.g., blogs, news feeds, podcasts, web services messaging). In semistructured data, such as XML, tags (metadata) and content (data) are mixed together in the same file.

The vast majority of applications for managing XML rely on XPath [W3C 2007]. Hence, web developers use XPath queries for many of the tasks involved in the processing of XML collections. Such collections are normally handled one document at a time, whether the document is an individual RSS[1] file (used by content distributors to deliver to subscribers frequently updated content over the Web), a single SOAP[2] message, or a Wikipedia article in XHTML.

Even when XML collections have a schema (which can be either a DTD [W3C 2006] or an XML Schema [W3C 2004]), the actual structure present in each document may exhibit significant variations for several reasons. First, schemas can be very lax. One reason for this is the extensive use of the `<xsd:choice>` construct in XML schemas, which allows optional elements to occur any number of times, including zero. Such a construct is very common in RSS for instance. Second, a schema can be very large and only subsets may be used in a given instance. This is the situation with several industry specific standards for different application domains that may contain hundreds of elements, such as UBL[3] or HR-XML[4]. UBL is designed to handle supply chain transactions and applications such as purchase orders, shipping notices, and invoices. HR-XML contains schemas for human resource management such as resumes, payroll information, and benefits enrollment. Both are domains requiring sophisticated applications to manage, publish, and exchange information among complex document collections. Finally, a schema can be extended by using the `<xsd:any>` XML Schema construct, which allows arbitrary content from other schemas to appear under a given element. Such a construct enables different user communities to pick and choose how to combine schemas. Consequently, it provides great flexibility, but makes it harder to determine the structure of the documents that actually appear in a given collection. Examples of the `<xsd:any>` extensions can be found in a wide variety of industry standards, including RSS, UBL and HR-XML. For instance, the UBL standard permits a contractor to represent invoice documents that include HR-XML TimeCard elements for the contractor employee's time and expenses. The actual structure of invoice collections will vary significantly across contractors and customers. If an enclosing messaging schema is used, even the UBL and HR-XML fragments in the document can be replaced by other invoicing and time billing schemas. In these scenarios, schemas alone are insufficient for understanding the structure (metadata) of the documents in the collection for either writing or optimizing XPath evaluation.

An application developer working with this type of collection faces several chal-

[1] http://www.rss-specifications.com/
[2] http://www.w3.org/TR/soap/
[3] http://oasis-open.org/committees/ubl/
[4] http://hr-xml.org

lenges. She must learn enough about the structure present in the XML collection to be able to write meaningful XPath queries. She must also develop an understanding of how the XPath expressions behave across different documents in the collection. Even when a task deals with a single document at a time, the developer needs to extrapolate the behaviour of queries over a single document across the entire collection over which the task may be repeatedly applied. In this context, understanding the actual metadata of a web collection can be a significant barrier, even for collections validated against a schema.

XML structural summaries are graphs representing relationships between sets of XML elements (i.e., extents). Extents can also be viewed as *mappings* between instance (document) nodes and summary nodes, i.e., all document nodes that are map to the same summary node form the extent of the summary node. Unlike schemas, which *prescribe* what may and may not occur in an instance, summaries *describe* the metadata that is actually present in a given collection.

There is an abundant literature on structural summaries (see Section 2). However, none of these proposals summarize ordering and cardinality metadata. Such information is very important in understanding collections of documents, and in helping users to write meaningful queries.[5] Consider for instance Wikipedia articles containing sections with a variable number of images and captions. A summary that provides information on the number of articles containing images and the number of images containing captions can help a user in writing queries related to images. For example, a user wanting all articles containing images of Barack Obama may (without metadata) simply query for images mentioning Barack Obama in the caption. However, if she knows that few images are captioned, she can understand that the empty answer to her first query may be more a reflection of the structure of the documents, rather than their contents. Based on this summary information, she can then modify her query to better find the information she is seeking.

Furthermore, since the previously known summaries are defined via their own unique creation and manipulation algorithms, it is hard to determine how they can be used together effectively for processing today's increasingly heterogeneous and large web collections. Specifically, the summary information is not defined declaratively, limiting the ease with which these summaries can be used within standard data management tasks.

In this paper, we propose a novel approach for flexibly summarizing the structure of metadata actually present in an XML collection. We introduce DescribeX, a framework that supports constructing *heterogeneous* summaries. DescribeX summaries create a partition of the document nodes, i.e., a set of pairwise disjoint subsets of document nodes whose union consists exactly of all document nodes in the collection. Each set of this partition is an extent defined by a path regular expressions on axes, or *axis path regular expression* (*AxPRE*, for short). AxPREs provide the flexibility necessary for *declaratively* defining complex mappings between document nodes and summary nodes capable of expressing order and cardinality, among other properties. Each AxPRE can be specified by the user or obtained from any expression in the complete XPath language (all the axes, document order, use of

[5]By user, we mean a *DescribeX user* who is an IT specialist that manages large document collections. Note that we are not referring to average web users.

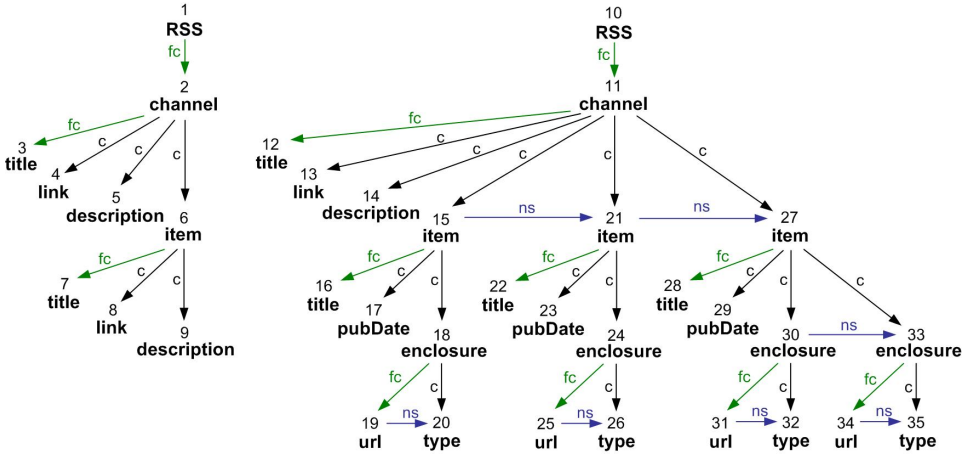

Fig. 1. Axis graphs of RSS feed samples

parentheses, etc.) Given an arbitrary XPath expression posed by a user, DescribeX can create a partition defined by an AxPRE that captures exactly the structural aspects expressed in the query.

AxPRE summaries have a unique capability that makes them suitable for describing the structure of XML collections: they are the first summaries capable of declaratively defining and refining the summary extents using a powerful language. In addition, DescribeX summaries express relationships between instance nodes that go beyond the traditional parent-child (e.g., next sibling, following, preceding, etc.) DescribeX can significantly help in the understanding of the structure of large collections of XML documents. As has already been established elsewhere [Consens and Rizzolo 2007], DescribeX can also improve the performance of XPath queries over collections. Most importantly, DescribeX captures most summary proposals in the literature by providing (for the first time) a common declarative definition for them.

1.1 Motivating example: exploring RSS feeds with summaries and XPath queries

Consider a web application developer, Sue, who works for a leading web content syndication company, like YellowBrix[6] and Comtex News[7]. One of the main services offered by these companies is to provide subscribers with customized content solutions aggregated from thousands of content providers worldwide. Sue has to implement the web application that retrieves RSS feeds from the content providers in order to produce aggregated meta-feeds tailored to the subscriber's needs. This kind of aggregation is typically implemented using languages ranging from Perl, PHP and Java to XSLT, XPath and XQuery. The most flexible option is to use XPath to extract the individual elements of the source RSS feeds and XQuery to generate each aggregated meta-feed.

Figure 1 shows instances of two sample RSS feeds represented as *axis graphs*.

[6]http://www.yellowbrix.com/
[7]http://www.comtexnews.net/

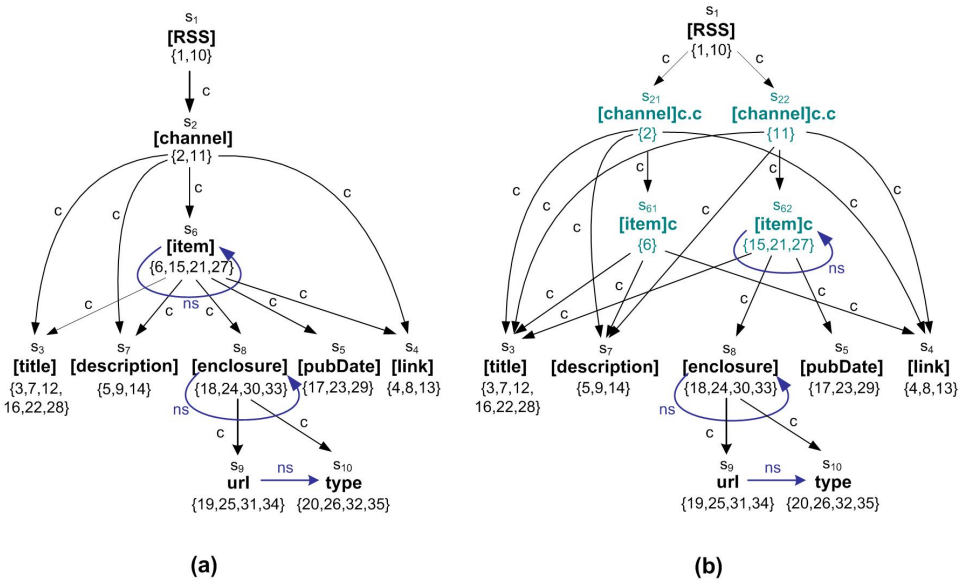

Fig. 2. Label SD (a), and heterogeneous SD (b) of the RSS feed samples

An axis graph can display selected binary relations between elements in an XML document tree, like *doc*, *c*, *ns*, and *fc* shown in the figure (shorthands for XPath axes *document*, *child*, *next-sibling*, and for the derived axis *firstchild*, respectively). The semantics of these axes is straightforward: the edge from element 6 to 7 labeled *fc* means that 7 is the first child of 6 in document order, and the edge from element 15 to 21 labeled *ns* means that 21 is the next sibling of 15, also in document order. For simplicity, we display only selected *ns* edges, and we do not draw the *c* edge between two nodes when a *fc* edge already exists between them (even though every first child is also a child). Being binary relations, axes have inverses, e.g., the inverse of *c* is *p* (shorthand for *parent*) and the inverse of *ns* is *ps* (shorthand for *preceding-sibling*). For clarity, these inverses are not shown in the figure.

Subscribers to this type of syndication service require tailored information products with customized content and structure. Each source RSS feed may span several days or weeks, and there might be more than one item in the feed per day. Some subscribers may ask for all the items in a feed whereas others are only interested in the most recent ones. Some want items in specific media formats or even in multiple formats (in separate enclosure elements). Others use readers that support only a few RSS extensions. Since the source feeds come from thousands of different providers using a large number of RSS extensions and formats, channels and items may contain hundreds of possible distinct combinations of subelements. From these large, heterogenous collections of feeds Sue needs to find and extract elements with similar substructure in order to create customized aggregations for the subscribers.

Sue has access to a repository containing several months of sample feeds published by the content providers (a collection with hundreds of thousands of XML files). She could manually open a few files from each RSS feed to get a sense of their

broad structure. With thousands of content providers and feeds, Sue would need to browse thousands of files. Even if she manages to do that, she will have to check if each sample file is indeed characteristic of the entire feed it comes from. Sue could use XPath queries for this task, but with such high degree of heterogeneity she will have to come up with a very large number of queries on her own.

Using DescribeX, Sue can create a *summary descriptor* (*SD* for short) like the one shown on Figure 2 (a). This *label SD*, created from the two feeds in Figure 1, partitions the elements in the feeds by element name. For example, SD node s_6 represents all the *item* elements in the two documents, $\{6, 15, 21, 27\}$ (this set is called the *extent* of s_6). An SD edge is labeled by the axis relation it represents. For instance, 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. (More expressive types of edges will be introduced in Section 4.)

From the *label SD*, Sue gets a first glimpse of the feeds structure. She learns that *channel* elements in the collection always contain *title*, *link*, *description*, and *item* subelements. The *ns* loop on node s_8 indicates that an *item* may contain repeated *enclosure* elements (different enclosures within the same item are often used to post the same content in different media formats or languages). Furthermore, the structure of *item* elements may vary: they always include a *title* element, but may contain any combination of *description*, *enclosure*, *pubDate*, and *link* elements. Note that the label SD does not provide information on exactly which combinations actually appear.

At this point Sue has two options:

(1) She can interactively *refine* the SD node s_2 in the label SD in order to learn how many different types of channels exist in the collection (i.e., how many subsets of *title*, *enclosure*, *description*, *link* and *pubDate* are present within *item* elements).

(2) Since she already knows that some *item* elements have a *pubDate* from the label SD and she is interested in channels that contain such items (most of their subscribers require items with a known publication date), she can write query Q1 to retrieve them.

```
Q1 = /rss/channel/item[pubDate]
```

Sue can now decide either to run Q1 using the current SD or to make DescribeX *adapt* the current SD to Q1. Adapting an SD to a query workload entails refining the SD according to characteristics common to the workload. If she picks the former option, DescribeX finds the only SD node that contains a superset of the answer (s_2) and runs Q1 on its entire extent. If Sue chooses the latter option, DescribeX changes the SD by partitioning the single *channel* node s_2 in Figure 2 (a), which represents all channels in the collection, into two *channel* nodes: one with a *pubDate* within their *item* elements and another without a *pubDate* (s_{22} and s_{21} in Figure 2 (b), respectively).

Summaries in DescribeX are defined and manipulated via axis path regular expressions (AxPREs). In short, AxPREs are path regular expressions over binary relations (in our case, the relations labeling the edges of an axis graph). These expressions define patterns over an axis graph. We denote each occurrence of a

pattern as a *neighbourhood.* More precisely, a neighbourhood of an element v by an AxPRE α is the subgraph local to v that matches α. For example, the AxPRE $[item].c$ describes the neighbourhoods composed of *item* elements and their children, while the AxPRE $[channel].c.c$ describes the neighbourhoods of *channel* elements containing not only their children, but also their grandchildren. Neighbourhoods also determine how nodes are clustered in the extents: the extent of an SD node labeled by an AxPRE α contains all axis graph nodes that have *similar* neighbourhoods according to α. Consider nodes 2 and 11 from Figure 1 and their neighbourhoods by AxPRE $[channel].c.c$. According to such an AxPRE together with DescribeX's notion of similarity, nodes 2 and 11 belong to different extents (those of nodes s_{21} and s_{22} in Figure 2 (b), respectively) because the labels of the grandchildren of 2 (*title*, *link* and *description*) are different from those of the grandchildren of 11 (*title*, *pubDate* and *enclosure*). This makes nodes 2 and 11 not similar with respect to AxPRE $[channel].c.c$. The notion of similarity used by DescribeX and the syntax and semantics of AxPREs are studied in detail in Section 3. AxPREs can be derived from a query in order to adapt an SD to it. Alternatively, a developer like Sue could have written the expression herself had she wanted to refine the s_2 SD node in the label SD according to the substructure of the elements in its extent.

The same process can be applied to more complex requirements. For instance, some subscribers need just the media content and associated information (author, duration, format, etc.) of podcasts that are published daily. The media content in podcasts can appear inside many different element names with associated information that varies drastically depending on the RSS extension used by the content provider (for instance, Yahoo Media RSS[8], Dublin Core[9], iTunes[10], etc.) Sue can easily identify which elements are actually present in the collection by using the SD of Figure 2 (b) without having to write a set of different queries for each potential extension used by the providers.

Adapting the SD to a query workload is also useful in a document-at-a-time approach to query evaluation. The adaptation process reduces the number of documents on which queries in the workload need to be evaluated, potentially yielding a significant speedup. That is, after adapting the SD to a given query Q, DescribeX can evaluate Q only on those documents that are guaranteed to provide a non-empty answer for the *structural subquery* of Q (the expression that results from removing all non-structural predicates such as those containing functions).

It is important to note that DescribeX can recognize two kinds of channels with different structure beyond the elements directly contained by them, a capability not available using DTD's (unless *channel* 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 [Bex et al. 2006; Garofalakis et al. 2003]) do not help to identify the two kinds of channels as done above. For instance, the DTD expression `<!ELEMENT channel (title, link, description, item)>` can be inferred for the *channel* elements occurring in Figure 1. However,

[8]http://search.yahoo.com/mrss/
[9]http://dublincore.org/documents/dcmi-namespace/
[10]http://www.apple.com/itunes/whatson/podcasts/specs.html

a DTD can only give a rule for the children of *channel*, there is no mechanism for giving rules relating *channel* elements to their grandchildren or any other elements farther away. In contrast, the AxPRE summary in Figure 2 (b) can distinguish between a *channel* containing an *item* with a *pubDate* element from those that contain a *description*. Since the XML Schema specification [W3C 2004] does not allow an element name to have two different types, no XML Schema can recognize the two kinds of channels represented by s_{21} and s_{22} either without renaming. More importantly, even though XML Schemas are more expressive than DTDs [Martens et al. 2006], schema types still depend solely on ancestors-descendant relationships between elements. Thus, as in DTDs, we cannot have an *item* type for item elements that are part of a series (e.g., items 15, 21 and 27) and a different type for single items (e.g., item 6). In contrast, DescribeX can distinguish between such types of items by considering neighbourhoods including the *ns* axis.

1.2 Other Applications

In addition to the increasingly popular content syndication services just described, many other web communities can benefit from a flexible summary framework like DescribeX. We already described applications to web-based supply chain management (UBL) and human resources (HR-XML). In the bioinformatics domain, the protein informatics community has develop a common XML-based format for exchanging protein-protein interaction (PPI) data, called Proteomics Standards Initiative Molecular-Interaction (PSI-MI) format[11]. PSI-MI is the de-facto model for PPI used by many molecular interaction databases such as BioGRID[12], Human Protein Reference Database [13], and IntAct[14]. These are all examples of "deep Web" content (not indexed by standard search engines) published or exchanged in XML format. The PSI-MI XML schema has a large number of optional elements to allow flexibility. Since different databases use different fragments of the schema, PSI-MI data can be very heterogeneous. As a result, finding common structural patterns and understanding schema usage can be challenging [Samavi et al. 2007].

Wikipedia[15] is a well known free content, multilingual encyclopedia written collaboratively by contributors around the world. XML collections extracted from the Wikipedia have existed for several years now. INEX has used several corpora based on the English part of Wikipedia for structured information retrieval since 2006 [Denoyer and Gallinari 2006]. Snapshots of the encyclopedia in XML format are also available at the Wikipedia site[16]. Wikipedia is an example of a classical hypertext document collection that can be used more effectively with the assistance of a summary management tool like DescribeX. In our evaluation, we will use both content syndication data (RSS feeds) and Wikipedia data as representative collections of the diverse data managed by web applications.

[11] `http://psidev.sourceforge.net/mi/xml/doc/user/`
[12] `http://www.thebiogrid.org/`
[13] `http://www.hprd.org/`
[14] `http://www.ebi.ac.uk/intact/`
[15] `http://wikipedia.org/`
[16] `http://en.wikipedia.org/wiki/Wikipedia_database`

1.3 Organization

The rest of the paper is structured as follows. Section 2 gives an overview of the large body of related work in the literature. Section 3 introduces the DescribeX framework, including the AxPRE language and some basic notions such as neighbourhood and bisimilarity, whereas Section 4 defines summary descriptor (SD). Section 5 revisits some of the related work discussed in Section 2 and explains how they can be captured by the DescribeX framework and how DescribeX offers significant new functionality. Section 6 presents two novel mechanisms, AxPRE refinement and stabilization, for declaratively changing the description provided by an SD using AxPREs. Refinement and stabilization are central to the use of DescribeX summaries. Section 7 describes the implementation of the DescribeX summarization engine for creating and manipulating SDs of XML collections and provides experimental results, using gigabyte size XML collections, that validate the performance of the techniques employed by our framework. Our evaluation is focused on showing that our summarizations can be created and managed efficiently. We also give examples of how they could be used interactively for document collection exploration, and an intuition for their usefulness. We conclude in Section 8 by presenting some future research issues including the development of document collection benchmarks focussed on the usability of such collections and the role of summarization in enhancing usability.

This paper considerably expands our previous work on structural summaries. An early version of the application example described in Section 1.1 appeared in a short (three page) poster paper [Consens et al. 2008]. A visual, interactive tool based on the DescribeX framework was presented in a system demonstration at the ICDE conference [Ali et al. 2008]. The notions of axis graph, AxPRE, AxPRE neighbourhood, labeled bisimulation, AxPRE partition and Summary Descriptor (Definitions 3.1, 3.4, 3.10, 3.12, 3.16, and 4.1, respectively) first appeared in a workshop paper [Consens and Rizzolo 2007], which also includes an experimental study of XPath query evaluation using DescribeX. Other than the definitions just mentioned, the rest of Sections 3 and 4 (including a new definition of a Summary Descriptor and a construction algorithm), together with Sections 5, 6 and 7 in their entirety, are published here for the first time. We should also mention that most of the material in this paper is part of an unpublished PhD thesis [Rizzolo 2008].

2. RELATED WORK

2.1 Structural summaries

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.

Most of the summary proposals in the literature define synopses of predefined subsets of paths in the data. They construct a labeled graph that represents relationships between sets of XML elements. Examples of such summaries are region inclusion graphs (RIGs) [Consens and Milo 1994], representative objects (ROs)[Nestorov et al. 1997], dataguides [Goldman and Widom 1997], 1-index, 2-index and T-index [Milo and Suciu 1999], and more recently, ToXin [Rizzolo and Mendelzon 2001],

A(k)-index [Kaushik et al. 2002], and F&B-Index [Kaushik et al. 2002]. Dataguides and ROs group nodes into sets according to the label paths incoming to them (each node may appear more than once in the dataguide if the document instance is not just a tree). RIGs, 1-index, T-index, ToXin, F&B-Index, and F+B-Index, on the other hand, partition the data nodes into equivalence classes (called *extents* in the literature) so that each node appears only once in the summary. The partition is computed in different ways: according to the node labels (RIGs), the label paths incoming to the nodes (1-index, ToXin, A(k)-index), the label paths going out from the nodes (reversed dataguides), or label paths both incoming and outgoing (F&B-Index and F+B-Index). The length of the paths in the summary also varies: ToXin, 1-index, F&B-Index and F+B-Index summarize paths of any length, whereas A(k)-index is a synopsis of paths of a fixed length. Updates to structural summaries have been studied in [Kaushik et al. 2002] and [Yi et al. 2004]. For XML documents with temporal information, an extension to ToXin (called TempIndex) summarizes paths that are valid continuously during a certain time interval [Rizzolo and Vaisman 2008].

Based on the A(k)-index, a recent proposal [Fletcher et al. 2007] defines partitions of paths, rather than nodes, called P(k)-partitions – where k is the maximum length of the paths being summarized. Since this proposal is based on navigational XPath, it supports only expressions containing composition of *parent*, *ancestor*, *child*, and *descendant* axes. In contrast, DescribeX can be used to evaluate arbitrary expressions in the complete XPath language (with all the axes, functions, use of parenthesis, etc.).

Other summaries are augmented with *statistical information* of the instance for selectivity estimation, including path/branching distribution [Polyzotis and Garofalakis 2006b], value distributions [Polyzotis and Garofalakis 2006a], and additional statistical information for approximate query processing [Polyzotis et al. 2004].

A few *adaptive* summaries like APEX [Chung et al. 2002], D(k)-index [Qun et al. 2003], and M(k)-index [He and Yang 2004] use dynamic query workloads to determine the subset of incoming paths to be summarized. APEX is a summary of frequently used paths that summarizes incoming paths to the nodes and adapts to changes in the workload by changing the set of path considered in the synopsis. The workload APEX considers are expressions containing a number of *child* axis composition that may be preceded by a *descendant* axis, without any predicate. APEX summarizes incoming paths to the nodes and adapts to changes in the workload by changing the set of paths summarized. D(k)-index and M(k)-index, in contrast, summarize variable-length paths based on both the workload and local similarity (the length of each path depends on its location in the XML instance).

There has been almost no work on summaries that capture the node ordering in the XML tree: the only proposals we are aware of are the early region order graphs (ROGs) [Consens and Milo 1994] and the Skeleton summary [Buneman et al. 2005] that clusters together nodes with the same subtree structure. Skeleton has additional structures that store relationships between individual nodes that belong to different equivalence classes.

In contrast to these proposals, DescribeX is capable of declaratively defining complex mappings between instance nodes and summary nodes for expressing order,

cardinality, and relationships that go beyond the traditional parent-child (e.g., next sibling, following, preceding, etc.) In addition, DescribeX provides a declarative definition for the first time for most of the proposals discussed above (for more details on how DescribeX captures other structural summaries see Section 5).

A summary can also be used for creating a simplified version of an XML Schema. In this so called *schema summary* [Yu and Jagadish 2006b] related elements in the original schema are represented by a single element in the summary, thus reducing the schema information to a manageable size. The user can then expand the summary to get more detailed information of the parts of the schema she is interested in for writing XQuery queries. This functionality is similar to the one provided by the refinements in DescribeX, with two significant differences. First, the schema is always required in this proposal whereas DescribeX summaries can be constructed directly from the instance. Second, the schema summaries are a concise description based on fixed statistical information of the instances. Therefore, the way in which the user can expand or collapse summary nodes is also fixed for a given instance, whereas DescribeX provides a declarative way (based on AxPREs) of obtaining more or less detailed summaries and refinements on different axes. We must point out that the notions of schema summary importance and coverage introduced in [Yu and Jagadish 2006b] could be applied to our framework to determine what are the most relevant parts of the data to refine.

2.2 Hierarchical encodings

We should mention that, in addition to the use of summaries, query evaluation can be facilitated by *encoding* the hierarchical structure of an XML instance. *Node encoding* evaluations use some sort of interval encodings [Santoro and Khatib 1985] to label each node with its positional information within the XML instance. This positional information is used by join algorithms to efficiently reconstruct paths and label paths. Recent proposals for node encoding evaluations are region algebras [Consens and Milo 1994; Young-Lai and Tompa 2003], path joins (XISS) [Li and Moon 2001], relative region coordinates [Kha et al. 2001], structural joins [Al-Khalifa et al. 2002; Chien et al. 2002], holistic twig joins [Bruno et al. 2002; Jiang et al. 2003], XR-Tree [Jiang et al. 2003], PBiTree [Wang et al. 2003; Vagena et al. 2004], extended Dewey encoding for holistic twig joins [Lu et al. 2005], and FIX [Zhang et al. 2006], a feature-based indexing technique.

Structural encoding proposals are based on mapping the XML tree structure into strings and use efficient string algorithms for query processing. Since the size of each string grows with the length of the encoded path, many approaches use some sort of compression to offset this overhead. Examples of those are Index Fabric [Cooper et al. 2001], tree signatures [Amato et al. 2004], tree sequencing (ViST [Wang et al. 2003], PRIX [Rao and Moon 2004]), and NoK [Zhang et al. 2004]. These encodings can be used in conjunction with structural summaries to improve query evaluation performance. In fact, the availability of summaries can be of great assistance to an XML optimizer [Barta et al. 2005].

DescribeX uses an interval encoding derived from [Santoro and Khatib 1985] in which each element in the collections is represented by its start and end positions (the character offset from the beginning of the document they belong to).

2.3 Query evaluation

Another area closely related to summarization is answering XPath and XQuery queries using schemas and views. As in traditional database systems, the performance of XPath queries can be improved by rewriting them using caching and materialized views containing information relevant to the computation of the query. A recent contribution in this area includes a framework for XPath view materialization and query containment [Balmin et al. 2004] that uses value and structure indexes on views. Another framework was proposed in [Mandhani and Suciu 2005] for maintaining a semantic cache of XPath query results as materialized views used to speed-up query processing. Other work has considered the problem of deciding the existence of a query rewriting and finding a minimal rewriting using XPath views [Xu and Özsoyoglu 2005], and computing maximal contained rewriting for tree pattern queries (a core subset of XPath) [Lakshmanan et al. 2006]. The problem of using DescribeX summaries for XPath query evaluation has been studied in [Consens and Rizzolo 2007].

XML schemas, whenever present, can provide useful information for writing meaningful queries. However, as we mentioned in the introduction, schemas can be very large and difficult to comprehend. Even with the help of structural or schema summaries, structure-free query models (such as labeled keyword search) might be more useful in some cases than pure XQuery. *Meaningful summary queries* [Yu and Jagadish 2007] combine both structural conditions obtained from the schema summary together with structure-free conditions for the parts of the data not fully described in the summary. In this same direction, an extension of XQuery based on the notion of finding the most meaningful XML fragment that relates nodes corresponding to variables in the XQuery expression was introduced in [Li et al. 2008].

Finally, recent work tries to exploit structural summaries in XML retrieval. This kind of retrieval combines IR-style queries with structural constraints for querying XML document collections. The use of summaries for efficient evaluation of retrieval queries in the context of the Initiative for the Evaluation of XML Retrieval [Kazai et al. 2003] has been considered in [Ali et al. 2006; Ali et al. 2007].

2.4 Validating summaries

DTDs [W3C 2006] and XML Schemas [W3C 2004] are proposals used for validation and verification of XML documents. A DTD is a context-free grammar and an XML Schema is a typed definition language. Both are schemas in the database sense, and thus describe classes of documents and constrain their structure. However, they provide only a limited description of the instances that satisfy them and no mechanism to locate specific instance fragments. In contrast, summaries are constructed for a particular instance and consequently provide a tighter description of the data. They also contain the necessary information for locating the instance fragments they describe. DTDs and XML Schemas can be used to constrain the construction of summaries but they are no substitute for them. Moreover, summaries can be constructed even when DTDs and XML Schemas are not present.

In addition to describing an instance, DescribeX summaries could potentially be used for prescribing or constraining the data by adding schema constructs capable

of expressing XML schema languages like DTDs, XML Schemas, and Relax NG [Clark and Makoto 2001]. (For a survey on XML schema languages see [Murata et al. 2005].) There are many ways of integrating schema constructs with DescribeX summaries, but our study on this topic is very preliminar and we do not consider it further here.

3. AXPRE SUMMARIES BACKGROUND

This section provides an overview of the DescribeX framework. The framework includes a powerful language based on *axis path regular expressions* (AxPREs) for describing each set in a partition of instance nodes (extents). AxPREs provide the flexibility necessary for declaratively specifying the mapping between instance nodes and summary nodes for a given collection. These AxPRE mappings are capable of expressing order and cardinality, among other properties. AxPREs are evaluated on a graph (called *axis graph*) in which nodes are XML elements and edges are binary relations between them. Hence, AxPREs can be viewed as path regular expressions on binary relations. These relations include all XPath axes and additional ones that can be expressed in XPath.

Extents are defined using a novel approach: selective bisimilarity applied to subgraphs described by AxPREs (i.e., *AxPRE neighbourhoods*). This particular use of bisimulation supports the definition of summaries that go beyond the traditional parent and child hierarchical relationships covered by the abundant literature on summaries. Intuitively, nodes that have bisimilar subgraphs "around" them (i.e., neighbourhoods) belong to the same extent. For instance, DescribeX can define extents containing only nodes with the same set of outgoing label paths matching a given sequence of axes. Neighbourhoods are a key mechanism in the declarative definition of DescribeX summaries.

3.1 A regular expression language on axes

For representing an XML instance, DescribeX uses a model called an *axis graph*.

Definition 3.1 (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 named 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*. The edges of $\mathcal{A}$ are the tuples in the relations in *Axes* and each edge is labeled by the name of the relation to which it belongs. □

An axis graph is an abstract representation of the XPath data model [W3C 1999] extended with edges that represent XPath binary relations between elements. It can also include additional axes, such as *fc* (where $fc := child :: *[1]$, i.e., the first child), *ns* (where $ns := following\text{-}sibling :: *[1]$, i.e., the next sibling, same as the first of the following siblings), *id-idrefs* or any binary relation that can be expressed in XPath. When representing an XML instance, axis graph nodes are labeled by element or attribute names (including namespaces). Figure 1 depicts an axis graph for our running example.

In an axis graph we define paths and label paths as usual. We call a path defined on edges an *axis path*, and the string resulting from the concatenation of its labels is an *axis label path*.

Definition 3.2 (Axis Path and Axis Label Path). Let $\mathcal{A}$ be an axis graph, and v, v_n be two nodes in a connected subgraph of $\mathcal{A}$ such that there is a path $p = (v, axis_1, v_1, axis_2, \ldots, axis_n, v_n)$ from v to v_n. The *axis label path* of p is the string $\lambda(p) = axis_1[\lambda(v_1)].axis_2[\lambda(v_2)].\ldots. axis_n\ [\lambda(v_n)]$, where the string $ap = axis_1.axis_2.\ldots .axis_n$ is the *axis path* of p. □

Example 3.3. Consider the axis graph of Figure 1. Two of the paths from node 15 to 20 are $p = (15, c, 18, fc, 19, ns, 20)$ and $p' = (15, c, 18, c, 20)$. Their axis paths are $ap = c.fc.ns$ and $ap' = c.c$, respectively. Finally, the axis label paths of p and p' are $\lambda(p) = c[enclosure].fc[url].ns[type]$ and $\lambda(p') = c[enclosure].c[type]$, respectively. □

Definition 3.4 (Axis Path Regular Expression). An *axis path regular expression* (AxPRE) is an expression generated by the grammar

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

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

Definition 3.4 describes the syntax of path regular expressions on the binary relations (labeled edges) of the axis graph including node label tests. The function $B(l)$ is a boolean function on a label $l \in Label$ that supports elaborate tests beyond just matching labels.

An AxPRE defines a pattern we want to find in an instance. We need a way of computing all occurrences of such pattern in an axis graph – each occurrence will be called a neighbourhood. We do this by computing an automaton for the AxPRE, another for the axis graph, and then taking the intersection. Finally, a summary will group nodes with similar patterns together into an extent (DescribeX uses bisimulation as the notion of similarity).

The AxPRE semantics is given by the notion of *AxPRE neighbourhood* of a node (Definition 3.10). In order to compute an AxPRE neighbourhood we need first to define an automaton from the axis graph. Such an automaton will have two states for each node in the axis graph, one named *head* and the other *tail*. In addition, edges in the graph will be represented as transitions between *tail* and *head* states, and node labels as transitions between *head* and *tail* states.

Definition 3.5 (Axis Graph Automaton). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph and v a node in $\mathcal{A}$. The *axis graph automaton of $\mathcal{A}$ from v*, $\mathcal{M}_{\mathcal{A}}(v) = \{Q, \Sigma, \delta, q_0, F\}$, is an automaton [Hopcroft and Ullman 1979] defined as follows:

—For each node $w \in Inst$ there is a state $head(w) \in Q$, a state $tail(w) \in Q$ and a transition $\delta(head(w), [\lambda(w)]) = tail(w)$;

—For each edge (w_i, w_j) labeled $axis$ in $\mathcal{A}$ there is a transition $\delta(tail(w_i), axis) = head(w_j)$;

—All $tail(w)$ states in Q, $w \in Inst$, are final states in F, and $head(v)$ is the initial state q_0. □

Example 3.6. Figure 3 shows on the left hand side a fragment of the axis graph of our running example containing node 15. The axis graph automaton from node 15 (on the right hand side of the figure) has $head(15)$ as initial state and all *tail*

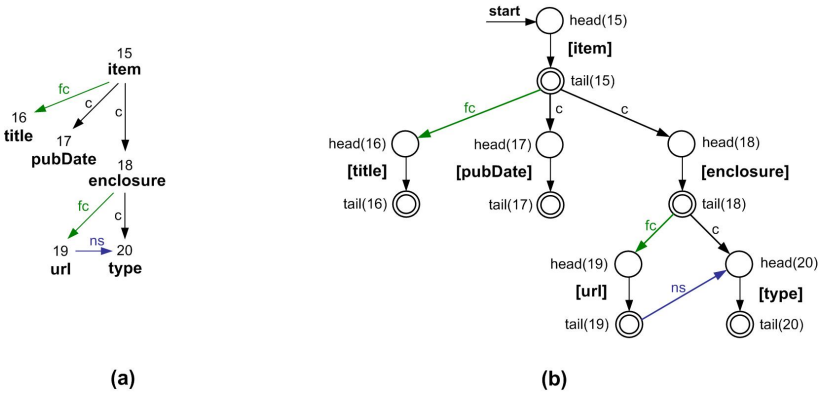

Fig. 3. Axis graph fragment from node 15 (a) and its automaton $\mathcal{M}_{\mathcal{A}}(15)$ (b)

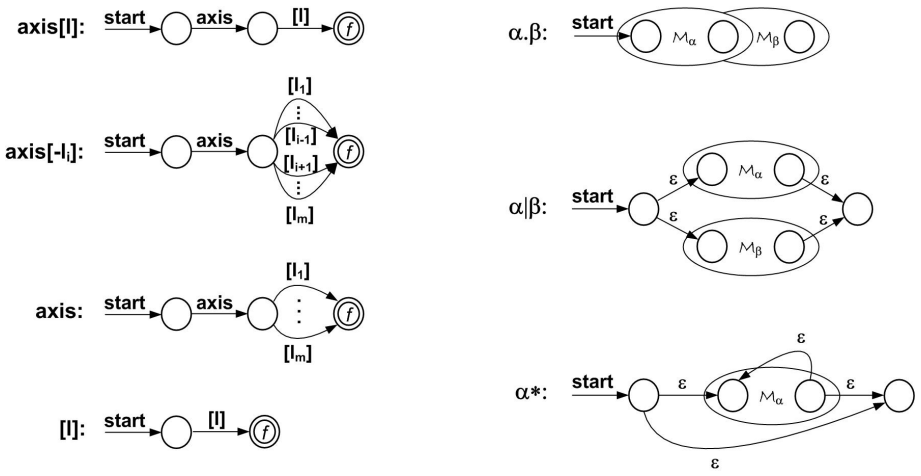

Fig. 4. Basic (left) and inductive (right) rules of the modified Thompson's construction

states as final. Each node in the axis graph fragment is unfolded into a head and a tail states in the automaton and its label is represented by a transition between them. For instance, node 20 with label *type* that has *ns* and *c* incoming edges in the axis graph and is represented in the automaton by a *head*(20) state that has *ns* and *c* incoming transitions and an outgoing transition [*type*] to *tail*(20). □

An automaton can be obtained from an AxPRE following the usual Thompson's construction for regular expressions with a minor change to the basis steps to account for AxPRE semantics (which requires accepting all prefixes of the language). The language accepted by the so called *AxPRE automaton* thus constructed will always be prefix-closed. (A language L is said to be prefix-closed if, given any word $l \in L$, all prefixes of l are also in L [Hopcroft and Ullman 1979].)

Definition 3.7 (AxPRE Automaton). Let α be an AxPRE. The *AxPRE automaton* of α is an automaton $\mathcal{M}_\alpha$ obtained from α with a modified Thompson's con-

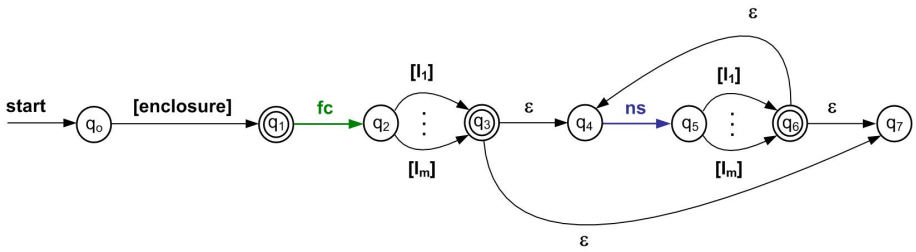

Fig. 5. AxPRE automaton $\mathcal{M}_{[enclosure].fc.ns^*}$

struction [Hopcroft and Ullman 1979] for accepting all prefixes (Figure 4), in which only the final states of the basis rules are kept as final in the resulting automaton (the inductive rules for concatenation, disjunction and Kleene closure do not mark any additional state as final). The transition function $\hat{\delta}(q_\alpha, axis)$ returns the states that can be reached by an $axis$ transition after following an arbitrary number (possibly zero) of ϵ transitions. □

Example 3.8. Consider the AxPRE $[enclosure].fc.ns^*$ and its automaton in Figure 5. The first rules that match are the concatenation (twice) and the Kleene closure (for ns). Then, the application of rule [*Label*] of the modified Thompson's construction creates states q_0, q_1 and the $[enclosure]$ transition between them. Next, the application of rule *axis* (with *fc* and *ns*) creates q_2, q_3, q_5, q_6, and the $[l_1], \ldots, [l_m]$ transitions from q_2 to q_3 and from q_5 to q_6 (there is one transition $[l_i]$ for each string in *Label*). □

An automaton for the intersection of two languages can be constructed by taking the product of the automata for the two languages [Mendelzon and Wood 1995; Yannakakis 1990].

Definition 3.9 (Intersection Automaton). Let $\mathcal{M}_\mathcal{A}(v)$ be the automaton of an axis graph $\mathcal{A}$ from a node v, and $\mathcal{M}_\alpha$ be the automaton of an AxPRE α. The *intersection automaton* $\mathcal{M}_\mathcal{A}(v) \cap \mathcal{M}_\alpha$ is an automaton in which states are pairs $(q_\mathcal{A}, q_\alpha)$ consisting of a state $q_\mathcal{A} \in \mathcal{M}_\mathcal{A}(v)$ and a state $q_\alpha \in \mathcal{M}_\alpha$, and there is a transition $\delta((q_\mathcal{A}, q_\alpha), \mathcal{X}) = (q'_\mathcal{A}, q'_\alpha)$ if there are transitions $\delta(q_\mathcal{A}, \mathcal{X}) = q'_\mathcal{A}$ in $\mathcal{M}_\mathcal{A}(v)$ and $\hat{\delta}(q_\alpha, \mathcal{X}) = q'_\alpha$ in $\mathcal{M}_\alpha$, where $\mathcal{X}$ is either an axis or a label. A state $\langle q_\mathcal{A}, q_\alpha \rangle$ is final (initial) if both $q_\mathcal{A}$ and q_α are final (initial). □

The machinery introduced in Definitions 3.5 through 3.9 is required for computing AxPRE neighbourhoods of nodes in the axis graph. The neighbourhood of a node v by α can be obtained by taking the intersection between the axis graph automaton from v and the AxPRE automaton of α, and then converting the resulting automaton to an axis graph fragment as described in Definition 3.10.

Definition 3.10 (AxPRE Neighbourhood of a Node). Let $\mathcal{A}$ be an axis graph, v a node in $\mathcal{A}$, α an AxPRE, and $\mathcal{M}_\mathcal{A}(v) \cap \mathcal{M}_\alpha$ the intersection automaton of $\mathcal{M}_\mathcal{A}(v)$ and $\mathcal{M}_\alpha$. The *AxPRE neighbourhood* of v by α, denoted $\mathcal{N}_\alpha(v)$, is the subgraph of $\mathcal{A}$ defined as follows:

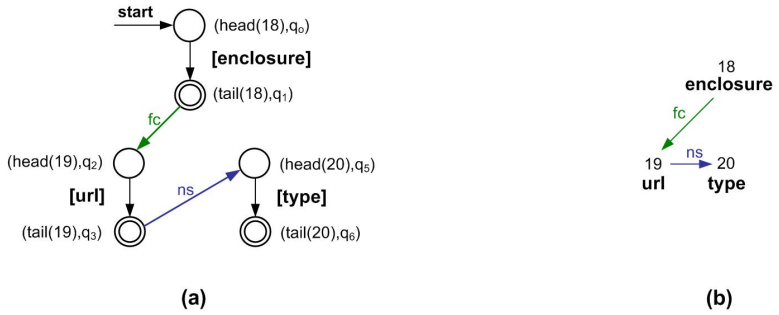

Fig. 6. Intersection automaton $\mathcal{M}_{\mathcal{A}}(18) \cap \mathcal{M}_{[enclosure].fc.ns^*}$ (a) and resulting AxPRE neighbourhood $\mathcal{N}_{[enclosure].fc.ns^*}(18)$ (b)

—For each transition $\delta((head(w), q_\alpha), l) = (tail(w), q'_\alpha)$, where $(tail(w), q'_\alpha)$ is a final state, there is a node w with label l in $\mathcal{A}$;

—For each transition $\delta((tail(w_i), q_\alpha), axis) = (head(w_j), q'_\alpha)$, where $(tail(w_i), q_\alpha)$ is a final state, there is an edge (w_i, w_j) labeled $axis$ in $\mathcal{A}$. □

The evaluation of an AxPRE α on an axis graph node v returns the AxPRE neighbourhood of v by α.

Example 3.11. Consider the intersection automaton of Figure 6 (a). States are labeled by pairs $(q_{\mathcal{A}}, q_\alpha)$, where $q_{\mathcal{A}}$ is a state in automaton $\mathcal{M}_{\mathcal{A}}(18)$ and q_α is a state in automaton $\mathcal{M}_{[enclosure].fc.ns^*}$. The intersection has been computed following Definition 3.9. The figure shows only the states that have some incoming or outgoing transition. Note that transition c between $tail(18)$ and $head(20)$ is not part of the intersection because fc is the only outgoing transition from q_1 in q_α. Figure 6 (b) shows the AxPRE neighbourhood of node 18, $\mathcal{N}_{[enclosure].fc.ns^*}(18)$, obtained by converting the intersection automaton to an axis graph fragment as described in Definition 3.10. Note that transitions from $(head(v), \ldots)$ to $(tail(v), \ldots)$ in the intersection are node labels in the AxPRE neighbourhood and that transitions from $(tail(v), \ldots)$ to $(head(w), \ldots)$ are edge labels (axes) in the neighbourhood.

Consider now the four $[item].c^*.c.ns$ neighbourhoods depicted in Figure 7. They match different number of iterations of the Kleene closure of c (c^*): 0 iteration for neighbourhood (a) and 1 iteration for the others. □

3.2 Neighbourhoods and bisimulation

AxPRE neighbourhoods allow us to define a notion of similarity between nodes in an axis graph. The idea underlying DescribeX is that nodes with similar AxPRE neighbourhoods will be grouped together. In particular, DescribeX uses the familiar concept of *labeled bisimulation* applied to AxPRE neighbourhoods, formalized by Definition 3.12.

Definition 3.12 (Labeled Bisimulation and Bisimilarity). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph, and $\mathcal{N}_\alpha(v_0)$, $\mathcal{N}_\beta(w_0)$ be two AxPRE neighbourhoods of $\mathcal{A}$ such that $Axes_\alpha \subseteq Axes$ and $Axes_\beta \subseteq Axes$. A *labeled bisimulation* between $\mathcal{N}_\alpha(v_0)$ and $\mathcal{N}_\beta(w_0)$ is a symmetric relation $\approx$ such that for all $v \in \mathcal{N}_\alpha(v_0)$, $w \in$

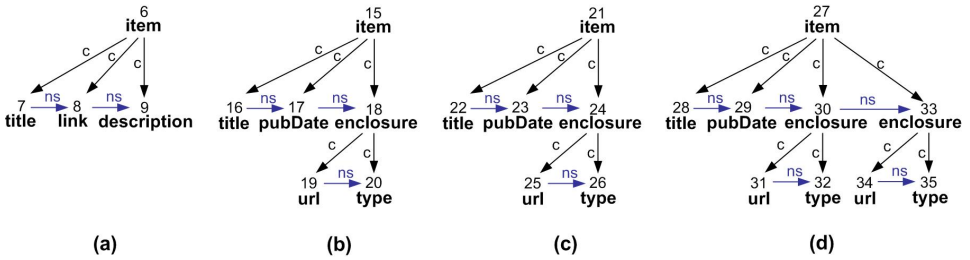

Fig. 7. All $[item].c^*.c.ns$ neighbourhoods

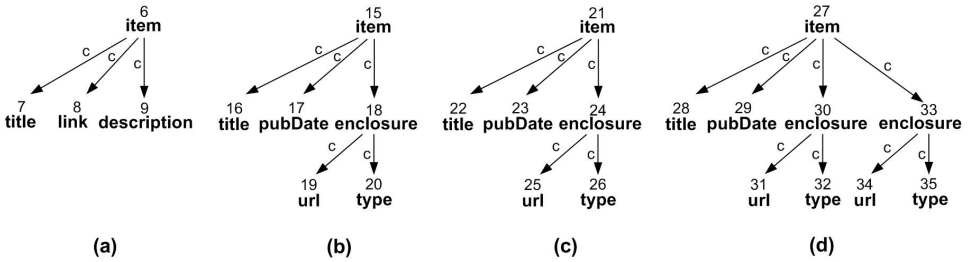

Fig. 8. All $[item].c^*$ neighbourhoods

$\mathcal{N}_\beta(w_0)$, $E_i^\alpha \in Axes_\alpha$, and $E_i^\beta \in Axes_\beta$: if $v \approx w$, then $\lambda(v) = \lambda(w)$; if $v \approx w$, and $\langle v, v' \rangle \in E_i^\alpha$, then $\langle w, w' \rangle \in E_i^\beta$ and $v' \approx w'$. Two nodes $v \in \mathcal{N}_\alpha(v_0)$, $w \in \mathcal{N}_\beta(w_0)$ are *bisimilar*, in notation $v \sim w$, iff there exist a labeled bisimulation $\approx$ between $\mathcal{N}_\alpha(v_0)$ and $\mathcal{N}_\beta(w_0)$ such that $v \approx w$. Similarly, two neighbourhoods $\mathcal{N}_\alpha(v_0)$ and $\mathcal{N}_\beta(w_0)$ are *bisimilar*, in notation $\mathcal{N}_\alpha(v_0) \sim \mathcal{N}_\beta(w_0)$, iff $v_0 \sim w_0$. □

Definition 3.12 captures outgoing label paths from the nodes. Bisimulation provides a way of computing a double homomorphism between graphs. The widespread use of bisimulation in summaries is motivated by its relatively low computational complexity properties. The bisimulation contraction of a labelled graph can be done in time $O(m \log n)$ (where m is the number of edges and n is the number of nodes in a labelled graph) as shown in [Paige and Tarjan 1987], or even linearly for acyclic graphs, as shown in [Dovier et al. 2004]. Using bisimulation also allows us to capture all the existing bisimulation-based proposals in the literature (Section 5).

Example 3.13. Let us consider the nodes 21 and 27 in the axis graph of Figure 1. Their $[item].c^*.c.ns$ neighbourhoods are depicted in Figure 7 (c) and (d), respectively. Based on Definition 3.12, we can define a labeled bisimulation $\approx$ between nodes 26 and 32 because they have the same labels and they do not have outgoing edges. For the same reasons we have $26 \approx 35$, $25 \approx 31$, and $25 \approx 34$. However, it is not possible to define a labeled bisimulation between 30 and 24 because, even though they have the same labels, 30 has one *ns* outgoing edge whereas 24 does not. Thus, $30 \not\approx 24$. This prevents us from defining a labeled bisimulation between 21 and 27 because 27 has a child (node 30) that is not bisimilar to any child of 21. Consequently, neighbourhoods (c) and (d) of Figure 7 are *not* bisimilar.

Let us now compare nodes 21 and 27 but with respect to their $[item].c^*$ neigh-

bourhoods, which are depicted in Figure 8 (c) and (d), respectively. In this case we have $30 \approx 24$ and $33 \approx 24$ because they have the same label and there is a labeled bisimulation between their children: $25 \approx 31$, $25 \approx 34$, $26 \approx 32$, and $26 \approx 35$. Similarly, we can conclude that $21 \approx 27$. Consequently, neighbourhoods (c) and (d) of Figure 8 are in fact bisimilar. □

Definition 3.14 (AxPRE Bisimilarity). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$. When two nodes v_0 and w_0 in $\mathcal{A}$ have bisimilar neighbourhoods by the same AxPRE α, that is $\mathcal{N}_\alpha(v_0) \sim \mathcal{N}_\alpha(w_0)$, we say that v_0 and w_0 are *AxPRE bisimilar* by α or *α-bisimilar*, in notation $v_0 \sim^\alpha w_0$. □

Example 3.15. Consider again the neighbourhoods in Figure 7. Nodes 21 and 27 have non-bisimilar $[item].c^*.c.ns$ neighbourhoods and thus $21 \not\sim^\alpha 27$, where AxPRE $\alpha = [item].c^*.c.ns$. However, if we consider now their $[item].c^*$ neighbourhoods, which are bisimilar, then $21 \sim^{\alpha'} 27$ for AxPRE $\alpha' = [item].c^*$. □

AxPRE bisimilarity is used for defining partitions of an axis graph. Intuitively, a so called *AxPRE partition* assigns two nodes v and w in an axis graph to the same class if their AxPRE neighbourhoods by a given α are bisimilar. This is formalized by Definition 3.16.

Definition 3.16 (AxPRE Partition). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph and α an AxPRE. An *AxPRE partition* of *Inst* by α, denoted $\mathcal{P}_\alpha$, is a set of pairwise disjoint subsets of *Inst* whose union is *Inst* defined as follows: two nodes $v, w \in Inst$ belong to the same set $P^i_\alpha \in \mathcal{P}_\alpha$ iff $v \sim^\alpha w$. □

Definition 3.17 (Positive Classes). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph, α an AxPRE and $P^\emptyset_\alpha = \{v \in Inst \mid \mathcal{N}_\alpha(v) = \emptyset\}$ the set of the empty neighbourhoods in the AxPRE partition of *Inst* by α. Then, $\mathcal{P}^+_\alpha = \mathcal{P}_\alpha - P^\emptyset_\alpha$ is the set of *positive classes* of $\mathcal{P}_\alpha$. □

Since all nodes that have an empty AxPRE neighbourhood belong to the same equivalence class, $\mathcal{P}_\alpha$ and $\mathcal{P}^+_\alpha$ differ in at most one set.

Given an AxPRE, the positive classes plus one additional class for the empty neighbourhood forms a partition. If we have another AxPRE whose positive classes fall exclusively within this empty neighbourhood class, then these two AxPREs may be used together to summarize an axis graph. We are interested in sets of AxPREs whose positive classes define a partition of *Inst*, which is formalized next.

Definition 3.18 (Positive Partition). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph. A set $\mathbb{A} = \{\alpha_1, \ldots, \alpha_n\}$ of AxPREs defines a *positive partition* of $\mathcal{A}$, denoted $\mathcal{P}_\mathbb{A}$, iff $\bigcup_i \mathcal{P}^+_{\alpha_i}$ is a partition of *Inst*. □

Note that a Positive Partition is indeed a partition (in particular, a node in a document can only appear once). The intuition behind the notion of positive partition from a set of AxPREs $\mathbb{A} = \{\alpha_1, \ldots, \alpha_n\}$ can be explained as follows. We know, by Definition 3.18, that each α_i in $\mathbb{A}$ defines an AxPRE partition which has positive classes and a unique empty neighbourhood class. In order for the set $\mathbb{A}$ to define a positive partition, the empty neighbourhood class of α_i has to be further partitioned by some α_j in $\mathbb{A}$. In other words, when the entire set $\mathbb{A}$ is considered, every node that belongs to the empty neighbourhood of some α_i also belongs to some positive class of some α_j.

Example 3.19. Let us consider first the AxPRE ϵ, which evaluated on each axis graph node will produce as many different neighbourhoods as different labels in the axis graph (each neighbourhood containing a single node). Since all nodes with bisimilar neighbourhoods will belong to the same class, if there are n different labels in the axis graph the ϵ positive partition will contain n classes (Figure 2 (a) shows the sets of the ϵ positive partition for our running example below each SD node). The same positive partition can be obtained with the set of expressions $\mathbb{A} = \{[l_1], \ldots, [l_n]\}$, where $l_1, \ldots, l_n$ are all the different node labels that appear in the axis graph. In our running example, the set of expressions equivalent to ϵ would contain $[RSS]$, $[channel]$, $[item]$, etc.

Let us consider now the AxPRE $[item]$. The partition by $[item]$ is obtained as follows: for each node in the axis graph, we compute the AxPRE neighbourhood corresponding to $[item]$, and all nodes with bisimilar neighbourhoods (i.e., all nodes that are $[item]$-bisimilar) will belong to the same class. Thus, the partition will consist of two classes: one containing all the nodes v such that $\lambda(v) = item$, which is the set {6, 15, 21, 27} (the positive class), and the other one with the remaining nodes (the empty neighbourhood class). On the other hand, the $[\neg item]$ partition will create as many classes as nodes v with labels $\lambda(v) \neq participant$ exist in *Inst*. In our running example, the $[\neg item]$ partition will have nine positive classes (one per label different from "item") whereas all nodes with "item" label will belong to the empty neighbourhood class. The two AxPREs $[item]$ and $[\neg item]$, when put together, define a positive partition with ten classes (one for each label). □

4. DESCRIBING SUMMARIES WITH AXPRES

In the previous section, we have introduced the basic machinery we need to define *summary descriptor* (SD, for short). Intuitively, an SD consists of an axis graph in which each node has associated an AxPRE and a set in its AxPRE partition, and whose edges represent axis relationships between those sets.

Definition 4.1 (Summary Descriptor). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph of an instance. A *summary descriptor* (SD for short) of $\mathcal{A}$ is a structure $\mathcal{D}_{\mathbb{A}} = (\mathbb{A}, \mathcal{G}, axpre, extent)$ that consists of:

—a set $\mathbb{A} = \{\alpha_1, \ldots, \alpha_n\}$ of AxPREs such that $\mathcal{P}_{\mathbb{A}}$ is the positive partition of $\mathcal{A}$ by $\mathbb{A}$;

—an axis graph $\mathcal{G} = (Sum, Axes^{\mathcal{D}}, Label, \lambda^{\mathcal{D}})$, called *SD graph*, representing axis relationships between nodes in the sets (extents) of the positive partition $\mathcal{P}_{\mathbb{A}}$ where:

  —Sum is a set of nodes;

  —$Axes^{\mathcal{D}}$ is a set of binary relations $\{E_1^{\mathcal{D}}, \ldots, E_n^{\mathcal{D}}\}$ in $Sum \times Sum$ such that there is a tuple $\langle s_j, s_k \rangle$ in $E_i^{\mathcal{D}}$ iff $\exists E_i^{\mathcal{A}} \in Axes, \exists v \in extent(s_j), \exists w \in extent(s_k) \wedge \langle v, w \rangle \in E_i^{\mathcal{A}}$ (edges are labeled by axis names);

  —$Label$ is the set of node labels from $\mathcal{A}$;

  —$\lambda^{\mathcal{D}}$ is a function that assigns labels in $Label$ to nodes in Sum.

—a bijective function $axpre$ that assigns AxPREs from $\mathbb{A}$ to nodes in Sum;

—a bijective function $extent$ that assigns a set from the positive partition $\mathcal{P}_{\mathbb{A}}$ to each node in Sum (the set assigned is called the extent of the node). □

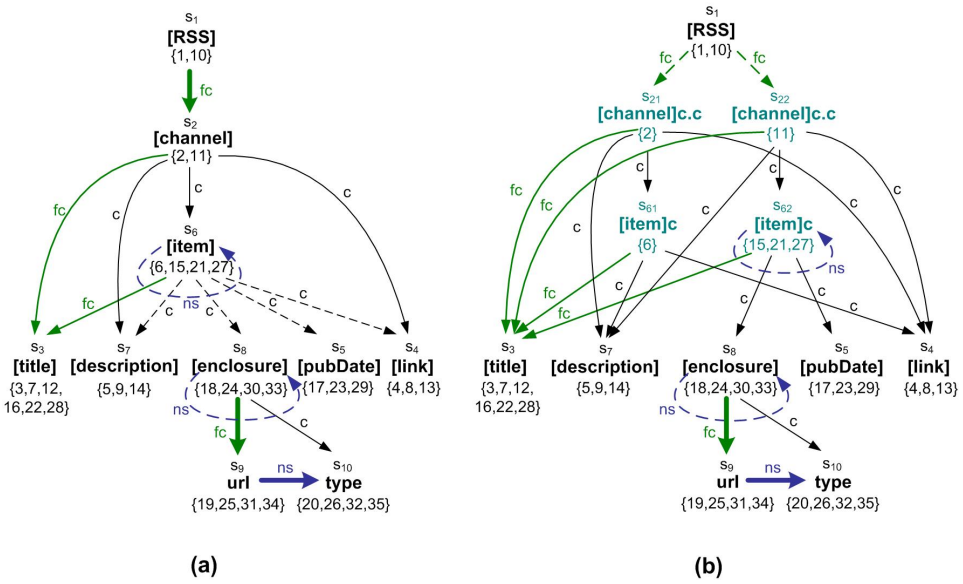

Fig. 9. Label SD (a), and heterogeneous SD (b), with stability-denoting edges

An SD has some particular characteristics. The set $\mathbb{A}$ uniquely defines the extents of the SD, and therefore its nodes, for any particular axis graph instance. In other words, given an axis graph $\mathcal{A}$ and the set $\mathbb{A}$ we can create the SD of $\mathcal{A}$ by $\mathbb{A}$. On the other hand, not any set of AxPREs define a positive partition and thus an SD. The first SDs we can distinguish are those that are defined by a unique AxPRE from those that have a multi-AxPRE definition. We denote the former ones as *homogeneous* SDs because all their nodes are defined uniformly. Homogeneous SDs are the most common in the summary literature (e.g., dataguides [Goldman and Widom 1997], 1-index [Milo and Suciu 1999], ToXin [Rizzolo and Mendelzon 2001], A(k)-index [Kaushik et al. 2002], F&B-Index [Kaushik et al. 2002], Skeleton [Buneman et al. 2005]). SDs defined by multiple AxPREs are called *heterogeneous*.

Definition 4.2 (Summary Axis Stability). Let $e = \langle s_i, s_j \rangle$ be an SD graph edge with label *axis*. We say that e is an *existential* edge if $\exists x \in extent(s_i), \exists y \in extent(s_j) \wedge \langle x, y \rangle \in axis$, and a *forward-stable* edge if $\forall x \in extent(s_i)$, $\exists y \in extent(s_j) \wedge \langle x, y \rangle \in axis$. □

Summary axis stability captures the relationship between edges in the SD graph and the axis graph, and generalizes to several axes the edge stability representation in XSketch [Polyzotis and Garofalakis 2006b].

Example 4.3. Consider the label SD of Figure 9 (a). Since there are ten different labels in the axis graph of the instance, there are ten summary nodes in the label SD. Nodes in the figure are labeled by their AxPREs, so we are considering a heterogeneous label SD in which $\mathbb{A}$ contains one AxPRE per label. The extent of each node is depicted below it. Edges represent summary axis relations. The nodes and edges in the figure constitute the SD graph of the label SD. Figure 9 (b) shows

another heterogeneous SD with a different set $\mathbb{A}$ where $[channel]$ and $[item]$ from the label SD have been replaced by $[channel].c.c$ and $[item].c$, respectively.

In both figures, existential edges are represented by dashed lines and forward-stable edges by solid lines. (Note that by definition all forward-stable edges are also existential.) 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 the figure 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. Intuitively, dashed edges, like (s_6, s_5) labeled c, mean that some element in the extent of s_6 has a child in the extent of s_5. Regular edges, like (s_6, s_3) labeled fc, mean that every element in the extent of s_6 has a first child in the extent of s_3. Finally, bold edges, like (s_9, s_{10}) labeled ns, mean that every element in the extent of s_{10} is the next sibling of some element in the extent of s_9 and that every element in the extent of s_9 has a next sibling in the extent of s_{10}.

Consider again Sue, the syndication company developer of our motivating example. Bold edges give Sue an idea of where the homogeneous part of the data is. For instance, edges (s_8, s_9) and (s_9, s_{10}) tell her that *enclosure* elements have *always* a *URL* as a first child followed by a *type*. From the label SD she knows that every *item* has a *title* because (s_6, s_3) is forward-stable, but she cannot tell which items have exactly *description*, *enclosure*, *pubDate* and *link* because their edges are just existential. However, after the refinement all edges from the *item* elements are forward-stable and thus the sets of items with each combination of children are clearly indicated in the *item* SD nodes. □

Algorithm 4.4 computes an SD D from an axis graph A and a set X of AxPREs that define a positive partition of A. Essentially, the algorithm creates the positive partition in one pass over A (outer loop spanning steps 2-25). Loop 4-24 computes the AxPRE neighbourhood of v for each α in X (step 5) and then finds the α for which the AxPRE neighbourhood of v is non-empty. Since X defines a positive partition as a precondition, then for every v there is one and only one α in X such that $\mathcal{N}_\alpha(v) \neq \emptyset$. This guarantees that condition in step 6 is true exactly once for every v in A.

The next task in the algorithm is to find the extent where v belongs. Loop 7-13 compares by bisimulation $\mathcal{N}_\alpha(v)$ with every node in D that has the same AxPRE α. If there is a node s in D with α but the α neighbourhoods of v and s are not bisimilar (step 10), then a new node *candidate* is created (step 15). The same happens if there is no s in D with α at all. In all cases v is added to the extent of *candidate* (step 19). Since each v in A may be in an *axis* relationship with nodes in any extent, the final loop 20-22 checks edge existence (for the input set of axes $Axes^{\mathcal{D}}$) between the node *candidate* and every other node in D. The result of the algorithm is an SD D where the extent of each node is a set in the positive partition of A by X and the axes in $Axes^{\mathcal{D}}$ satisfy the conditions in Definition 4.2.

As shown, the outer loop 2-25 performs $|Inst|$ iterations. At any given moment, there is at most the same number of nodes in D as in A (each extent having only one node) and all have the same AxPRE. Therefore, loop 7-13 performs $|Inst|$ iterations in the worst case. Each iteration computes an AxPRE bisimulation (step 10) with time complexity $O(m.log|Inst|)$, where m is the total number of tuples (edges) in

ALGORITHM 4.4. *CreateSD*(A, X)

Input: *An axis graph A, a set X of AxPREs that defines a positive partition of A, and a set $Axes^{\mathcal{D}}$ of SD axes where each axis contains only the empty tuple*
Output: *An SD D*

```
1: create empty SD D
2: for every v in A do
3:    candidate := ∅
4:    for every α in X do
5:       compute the α neighbourhood of v: N_α(v)
6:       if N_α(v) ≠ ∅ then
7:          for every node s in D such that axpre(s) := α  do
8:             let w be a node in extent(s)
9:             compute the α neighbourhood of w: N_α(w)
10:            if v ~^α w (i.e., N_α(v) ~ N_α(w))  then
11:               candidate := s
12:            end if
13:         end for
14:         if candidate = ∅ then
15:            create a new node candidate in D
16:            axpre(candidate) := α
17:            λ^D(candidate) := λ(v)
18:         end if
19:         add v to extent(candidate)
20:         for every node s' ≠ candidate in D do
21:            add tuple ⟨candidate, s'⟩ and ⟨s', candidate⟩ to the corresponding axis in Axes^D
               if conditions in Definition 4.2 are satisfied
22:         end for
23:      end if
24:   end for
25: end for
```

all axes in $Axis$. The worst case for loop 20-22 is the same as that of loop 7-13, so it also performs $|Inst|$ iterations. Thus, the total time complexity of Algorithm 4.4 is $O(|Inst|.m.log|Inst|)$.

The notion of an AxPRE neighbourhood can also be defined for an SD graph, and it is called *summary AxPRE neighbourhood* of a node. Since an SD Graph is in fact an axis graph $\mathcal{G} = (Sum, Axes^{\mathcal{D}}, Label, \lambda^{\mathcal{D}})$, for any given SD node s and AxPRE α we can define its SD graph automaton $\mathcal{M}_{\mathcal{G}}(s)$ (Definition 3.5) and intersect it with the AxPRE automaton $\mathcal{M}_\alpha$ (Definition 3.7) in order to obtain an AxPRE neighbourhood (Definition 3.10) of s.

Definition 4.5 (Partition Refinement). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph. If $\mathcal{P}_{\mathbb{A}}$ and $\mathcal{P}_{\mathbb{B}}$ are positive partitions of $\mathcal{A}$, $\mathcal{P}_{\mathbb{A}}$ is a *partition refinement* of $\mathcal{P}_{\mathbb{B}}$ if every set of $\mathcal{P}_{\mathbb{A}}$ is contained in a set of $\mathcal{P}_{\mathbb{B}}$. □

Definition 4.6 (SD Refinement). Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph and $\mathcal{D}_{\mathbb{A}} = (\mathbb{A}, \mathcal{G}, extent)$ and $\mathcal{D}_{\mathbb{B}} = (\mathbb{B}, \mathcal{G}', extent')$ be two SDs of $\mathcal{A}$. $\mathcal{D}_{\mathbb{A}}$ is an *SD refinement* of $\mathcal{D}_{\mathbb{B}}$ if $\mathcal{P}_{\mathbb{A}}$ is a partition refinement of $\mathcal{P}_{\mathbb{B}}$. □

PROPOSITION 4.7. *Let $\mathcal{A} = (Inst, Axes, Label, \lambda)$ be an axis graph, α and β be AxPREs, and $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ be AxPRE partitions of $\mathcal{A}$. If α is contained in β then $\mathcal{P}_\beta$ is a refinement of $\mathcal{P}_\alpha$.* □

PROOF. (Sketch) The proof follows from the notion of AxPRE neighbourhoods. If α is contained in β then for any given node v, its α neighbourhood is contained in its β neighbourhood. Consequently, two nodes that are not distinguished by α (i.e., they are α-bisimilar) may be distinguished by β, but not the other way around. This guarantees that β creates either the same partition as α or a refinement. □

COROLLARY 4.8. *Let* $\mathcal{A} = (Inst, Axes, Label, \lambda)$ *be an axis graph and* $\mathcal{D}_{\mathbb{A}} = (\mathbb{A}, \mathcal{G}, extent)$ *and* $\mathcal{D}_{\mathbb{B}} = (\mathbb{B}, G', extent')$ *be two SDs of* $\mathcal{A}$. *If every* $\beta \in \mathbb{B}$ *is contained in some* $\alpha \in \mathbb{A}$ *then* $\mathcal{D}_{\mathbb{A}}$ *is an* SD refinement *of* $\mathcal{D}_{\mathbb{B}}$. □

Example 4.9. Consider the label SD of Figure 9 (a). Recall that in the label SD, $\mathbb{A} = \{[l_1], ..., [l_n]\}$, where $l_i \in Label$, $l_i \neq l_j \ \forall \ i, j$, and $\bigcup_i \ l_i = Label$. Suppose we want to refine node s_6. For this node, the partition represented in the figure was produced by the AxPRE $[item]$. If we replace this AxPRE by $[item].c$ in $\mathbb{A}$, and apply this set of AxPREs to *Inst*, s_6 will be replaced by two new nodes: s_{61} and s_{62} (which appear in Figure 9 (b) with their respective extents). The new extents represent the fact that node 6 in the axis graph has children labeled *title*, *link* and *description* whereas nodes 15, 21 and 27 have children labeled *title*, *pubDate* and *enclosure*. Thus, applying $[item]c$ we obtain two different AxPRE neighbourhoods of s_6, plus the empty neighbourhood, which is itself partitioned by the remaining AxPREs.

Finally, suppose now that the label SD is defined using $\mathbb{A} = \epsilon$, and we want to refine node s_6 with $[item].c$. In this case, just adding the new AxPRE does not suffice, because we would not obtain an SD: the union of positive partitions will not be a partition of *Inst* because ϵ will still produce its own partitions. We solve this by adding the AxPRE $[\neg item]$, which will produce the remainder of the label SD and will send all nodes labeled *item* to the empty neighbourhood class. □

The notions of partition and SD refinement, besides describing the axis structure of an axis graph, allows us to define a *hierarchy* of SDs. This provides the basis for recognizing a lattice among different SDs, where each node corresponds to a different AxPRE definition. We will show that this lattice covers all the summaries addressed in the literature, plus more complex new ones. At the top of this hierarchy (i.e., the coarsest partition), the empty AxPRE defines a SD where each node is partitioned by label (as shown in Figure 10), a typical summary found in the literature [Consens and Milo 1994; Nestorov et al. 1997]. The bottom of the lattice may vary, although the finest partition granularity can be represented by the expression $(fc.ns^*)^*$, that produces a partition in which each node in the axis graph will belong to a different equivalence class.

Definition 4.10 (DescribeX Lattice). A DescribeX lattice with respect to a set of axes $A = \{a_1, \ldots, a_n\}$ is defined as follows: each node corresponds to an AxPRE generated by the grammar of Definition 3.4 when the terminal axis is one of $a_1, ..., a_n$. Also, there is an edge (n_1, n_2) in the lattice if and only if the AxPRE of n_2 is contained in the AxPRE of n_1. □

From Definition 4.10 it follows that the coarsest partition that the lattice may define is the label SD. The finest partition depends on the chosen set of axes.

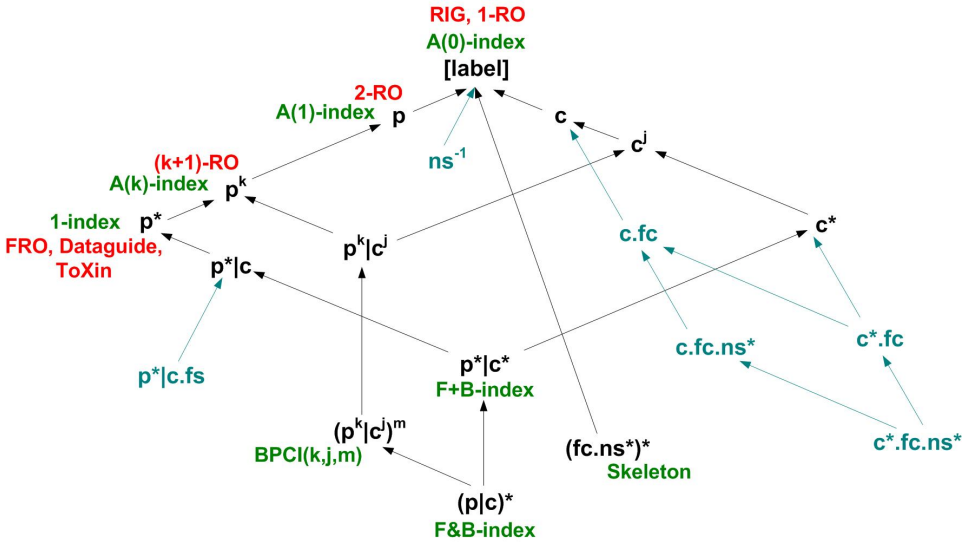

Fig. 10. AxPRE summary lattice capturing earlier homogeneous proposals

5. CAPTURING EARLIER LITERATURE PROPOSALS WITH DESCRIBEX

DescribeX summaries can be classified in a lattice that describes a *refinement* relationship between entire summaries (Definition 4.10). In this section we revisit some of the related work discussed in Section 2 that can be captured in such a lattice by the DescribeX framework.

Figure 10 shows a fragment of a DescribeX summary lattice that captures earlier proposals based on the notion of bisimilarity (in green) and ad-hoc constructions (in red). Each node in the figure corresponds to a homogeneous SD defined by an AxPRE. DescribeX not only captures most summary proposals but also provides a declarative way of defining entirely new ones: 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 7 ($[item].c^*.c.ns$).

The earliest bisimilarity-based summary proposal is the family presented in [Milo and Suciu 1999], which contains a p^* summary: the 1-index. The 1-index partition is computed by using *bisimulation* as the equivalence relation. The F&B-Index [Kaushik et al. 2002], 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. With this construction, the F&B-Index's equivalence classes are computed according to the incoming and outgoing label paths of the nodes. 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 F+B-index and the F&B-index are BPCI$(\infty, \infty, 1)$ and BPCI(∞, ∞, ∞) respectively. The A(k)-index [Kaushik et al. 2002] is a p^k AxPRE summary based on k-bisimilarity (bisimilarity computed for paths of length

k). Thus, the A(0)-index is a label SD, the A(1)-index is a p SD, the A(2)-index is a $p.p$ SD, and the A(h)-index is the p^h SD.

Unlike standard definitions in the bisimulation literature [Paige and Tarjan 1987; Dovier et al. 2004], 1-index, A(k)-index, F&B-index, and BPCI(k,j,m) use a bisimulation defined backwards in order to capture incoming paths to the nodes.

The notion of backwards k-bisimilarity used in the A(k)-index was defined to capture incoming paths on c and *idref* edges of length up to k. We provide next a more general definition for axis graphs that supports paths on all types of axes.

Definition 5.1 (Backwards k-Bisimilarity). Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two rooted subgraphs of an axis graph $\mathcal{A} = (Inst, Axes, Label, \lambda)$, such that $Axes_{\mathcal{G}_1} \subseteq Axes$ and $Axes_{\mathcal{G}_2} \subseteq Axes$, and let $r_1, r_2 \in Inst$ be the roots of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively. A *backwards k-bisimulation* between $\mathcal{G}_1$ and $\mathcal{G}_2$ is a symmetric relation $\approx_b^k$ 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 $v \approx_b^0 w$, then $\lambda(v) = \lambda(w)$; if $v \approx_b^k w$, and $\langle v', v\rangle \in E_i^{\mathcal{G}_1}$, then $\langle w', w\rangle \in E_i^{\mathcal{G}_2}$ and $v' \approx_b^{k-1} w'$. Two nodes $v \in \mathcal{G}_1$, $w \in \mathcal{G}_2$ are *backward k-bisimilar*, in notation $v \sim_b^k w$, iff there exist a backwards k-bisimulation $\approx_b^k$ between $\mathcal{G}_1$ and $\mathcal{G}_2$ such that $v \approx_b^k w$. □

Note that backwards k-bisimilarity defines an equivalence relation on the nodes in the axis graph. The partition created by the backwards k-bisimilarity corresponds to the A(k)-index, where k is a parameter that represents the length of the incoming paths summarized by the index.

Proposition 5.2. *Let G be an axis graph with $Axes = \{c\}$ ($Axes = \{c, idref\}$, respectively). The A(k)-index of G is a p^k SD (a $(p|idref)^k$ SD, respectively).* □

Proof. Consider an axis graph G with $Axes = \{c\}$. Two nodes v, w belong to the same extent in the p^k SD iff they are p^k-bisimilar. In addition, we know that $v \sim_{p^k} w$ iff there exists neighbourhoods $\mathcal{N}_{p^k}(v)$ and $\mathcal{N}_{p^k}(w)$ such that $v \sim w$. This means we can define a backwards k-bisimulation $\approx_b^k$ between $\mathcal{N}_{p^k}(v)$ and $\mathcal{N}_{p^k}(w)$ such that $v \approx_b^k w$ and thus $v \sim_b^k w$. □

The BPCI index is also based on the notion of backwards k-bisimulation. Algorithm 5.3 [Kaushik et al. 2002] constructs a BPCI(k_{in}, k_{out}, td) index using k_{in}-bisimilarity for the reversed edges (line 5), k_{out}-bisimilarity for the original edges (line7), and a td number of iterations (loop 3-8). The next proposition shows that the index constructed by Algorithm 5.3 can be captured by a specific SD.

Proposition 5.4. *Let G be an axis graph with $Axes = \{c\}$ ($Axes = \{c, idref\}$, respectively). The BPCI(k_{in}, k_{out}, td)-index of G is a $(p^{k_{in}}|c^{k_{out}})^{td}$ SD (a $(p^{k_{in}}|c^{k_{out}}|\ idref^{k_{out}}|\ (idref^{-1})^{k_{in}})^{td}$ SD, respectively).* □

Proof. The input data graph G can be viewed as an axis graph with the c axis, in which the reversed edges correspond to the c^{-1} (or p) axis. If id-idrefs are considered, then $Axes = \{c, idref\}$. Let us consider first the case of $Axes = \{c\}$. Lines 4 and 5 are equivalent to refining all nodes in the initial label SD (line 2) by the $c^{k_{out}}$ AxPRE. This produces a $c^{k_{out}}$ SD. Then, lines 6 and 7 produce a refinement of all $c^{k_{out}}$ nodes by the $p^{k_{in}}$ AxPRE, thus obtaining a $c^{k_{out}}.p^{k_{in}}$ SD. The iterative process is repeated td times (loop 3-8), which is equivalent to constructing a $(c^{k_{out}}.p^{k_{in}})^{td}$ SD. Again, by identity of regular expressions $(c^{k_{out}}.p^{k_{in}})^{td}$ is

ALGORITHM 5.3. *BPCI−construction*(G, k_{in}, k_{out}, td)

Input: *Data graph G, local similarities* k_{in} *and* k_{out}*, tree depth td*
Output: *BPCI*(k_{in}, k_{out}, td) *index I*

```
1: let P be a partition of the nodes in G
2: P ← label SD partition of G
3: for i=1 to td do
4:     reverse all edges in G
5:     P ← compute the backwards k_in-bisimilarity partition of G initializing the computation
       with P
6:     reverse all edges in G, obtaining the original G
7:     P ← compute the backwards k_out-bisimilarity partition of G initializing the computation
       with P
8: end for
9: for each equivalence class P_i ∈ P do
10:    create an index node s ∈ I
11:    extent(s) ← P_i
12: end for
13: for each edge from v to w in G do
14:    let s ∈ I be an index node such that v ∈ extent(s)
15:    let s' ∈ I be an index node such that w ∈ extent(s)
16:    if there is no edge from s to s' then
17:        create an edge from s to s'
18:    end if
19: end for
```

equivalent to as $(c^{k_{out}}|p^{k_{in}})^{td}$. The remaining of the algorithm (lines 9-16) creates existential edges like in Definition 4.2. When $Axes = \{c, idref\}$, the final AxPRE for the SD is $(c^{k_{out}}|p^{k_{in}}|idref^{k_{out}}|(idref^{-1})^{k_{in}})^{td}$. □

The Skeleton summary [Buneman et al. 2005] clusters together nodes with the same subtree structure, thus capturing node ordering in subtrees. Skeleton uses an entirely different construction approach, but its essence can be captured by the $(fc.ns^*)^*$ AxPRE.

The D(k)-index [Qun et al. 2003], and M(k)-index [He and Yang 2004] are heterogeneous SD proposals. All nodes s_i are described by p^k AxPREs 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 [Polyzotis and Garofalakis 2006b] 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.

Region inclusion graphs (RIGs) [Consens and Milo 1994] and representative objects of length 1 (1-RO) [Nestorov et al. 1997] are label SDs, that is ϵ SDs (because all their nodes s_i are described by the ϵ AxPRE). In general, representative objects are p^k SDs for XML tree instances. Therefore, the 1-RO is a label SD, the 2-RO is a p SD, the 3-RO is a $p.p$ SD, and the FRO (full RO) is the p^* SD.

Dataguides [Goldman and Widom 1997] group instance nodes into sets called *target sets* according to the label paths from the root they belong to. The dataguide construction is basically a nondeterministic-to-deterministic automaton translation. When the data instance is a tree, the dataguide's target sets are equivalent to the extents in our framework: a dataguide of an XML tree is a p^* SD.

ToXin [Rizzolo and Mendelzon 2001] also has a component that can be viewed as an p^* SD. ToXin consists of three index structures: the ToXin schema, the path index, and the value index. The ToXin schema is defined only for tree instances, and it is equivalent to a p^* SD graph.

6. REFINING SUMMARIES

The theoretical framework presented in previous sections set the foundation for developing an SD-based tool to help understand the metadata structure of highly heterogenous XML collections. Such a tool should provide the ability to interactively change an SD in order to obtain descriptions at different levels of detail. An obvious first choice for this would be to rebuild the entire SD from scratch using Algorithm 4.4. A better approach consists in defining an operation that changes the description provided by a single SD node by modifying its AxPRE. When the new AxPRE partition constitutes a refinement of the old one, we denote the operation an *AxPRE refinement*. (This operation was illustrated in Section 1.1 when Sue wanted to find out the different kinds of *channel* elements that appeared in the collection she was analyzing). The notion of refinement is tightly related to that of *stabilization* (discussed in Section 4). An edge stabilization determines the partition of an extent into two sets based on the participation (total or partial) of the extent nodes in the axis relation the edge represents. In this section, we discuss in detail the mechanisms for refinement and stabilization provided by DescribeX.

6.1 Concise descriptions

Since several SD nodes can share the same AxPRE, we need a mechanism for uniquely describing each SD node and its extent. The most straightforward way to do that would be just to list all nodes that belong to the extent (extensional representation). A more concise description is provided by the α neighbourhood of any node in the extent (intensional representation). Since all nodes in an extent are bisimilar, any α neighbourhood can be used to find all the other nodes in the extent by bisimulation.

In order to get the most concise description of the intensional representation, we need to find the *smallest* (in terms of number of nodes) neighbourhood in the extent of s that is bisimilar to all the others. This is achieved by computing a *bisimulation contraction* over all neighbourhoods in the extent of s. The bisimulation contraction of a given graph is the smallest graph that is bisimilar to it, which can be computed in time $O(m \log n)$ (where m is the number of edges and n is the number of nodes) [Paige and Tarjan 1987], or even linearly for acyclic graphs [Dovier et al. 2004]. Based on bisimulation contraction we define the notion of *representative neighbourhood*.

Definition 6.1 (Representative Neighbourhood). Let $\mathcal{D}$ be an SD and s a node in $\mathcal{D}$ such that $axpre(s) = \alpha$. The *representative neighbourhood* of node s for

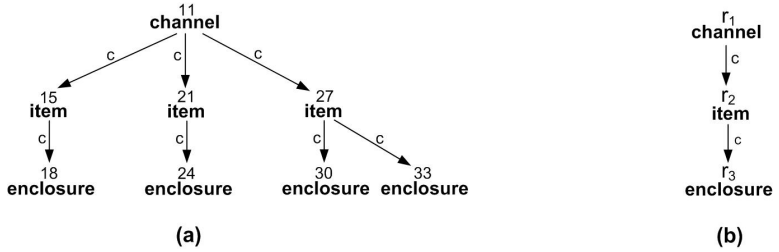

Fig. 11. The [*channel*].*c*[*item*].*c*[*enclosure*] neighbourhood (a) and its representative neighbourhood (b)

AxPRE α, denoted $\mathcal{R}_\alpha(s)$, is an axis graph that is the bisimulation contraction of all neighbourhoods $\mathcal{N}_\alpha(v_i)$, where $v_i \in extent(s)$. $\mathcal{R}_\alpha(s)$ has a single root node v_0 that is bisimilar to all $v_i \in extent(s)$. □

Note that the bisimulation contraction is not necessarily one of the neighbourhoods in the extent – it could be smaller than any of them. Rather, a representative neighbourhood is an entirely new axis graph that happens to be the smallest that is bisimilar to all neighbourhoods in an extent.

Example 6.2. Consider the AxPRE partition of our running example described by AxPRE [*interaction*].*c*[*participantList*].(*c*|*p*). It has only one set, the extent of node s, which contains nodes 2 and 14, and their representative neighbourhood is the graph shown in Figure 11. Note that such a neighbourhood does not belong to the extent of s (there is no *participantList* in the axis graph with only one *participant* node). □

For some neighbourhoods, deciding bisimilarity is equivalent to comparing the sets of simple label paths from their roots to their leaves. (A path is *simple* when it has no repeated edges.) In those cases, the neighbourhoods of an SD node s can be described by an *extent expression* (EE for short), denoted $ee(s)$, which is capable of computing precisely the set of elements in the extent of s. Interestingly enough, these EEs can be expressed in XPath [W3C 1999] and their functionality is similar to that of virtual views, as studied in [Rizzolo 2008].

Definition 6.3 (Path and LPath Sets). Let $\mathcal{N}$ be a neighbourhood in an axis graph $\mathcal{A}$, and v a node in $\mathcal{N}$. We denote by $Path(v)$ and $LPath(v)$ the set of simple axis paths and simple axis label paths from v, respectively. □

If deciding bisimilarity between a given set of neighbourhoods is equivalent to comparing their $LPath$ sets, we say that such neighbourhoods are *LPath distinguishable*.

Definition 6.4 (LPath Distinguishable Neighbourhoods). Let $\mathcal{A}$ be an axis graph and $\mathcal{N}_1(v_1), \ldots, \mathcal{N}_m(v_m)$ be neighbourhoods in $\mathcal{A}$. We say that $\mathcal{N}_1, \ldots, \mathcal{N}_m$ are *LPath distinguishable* when, for all $1 \leq i, j \leq m : \mathcal{N}_i(v_i) \sim \mathcal{N}_j(v_j)$ iff $LPath(v_i) = LPath(v_j)$. □

We are interested in LPath distinguishable neighbourhoods because they can be described by EEs. In general, determining whether a given set of neighbourhoods is

ALGORITHM 6.5. *RefineNode*(D, s, α)

Input: *An SD D, a node s in D, and an AxPRE $\alpha \subseteq axpre(s)$*
Output: *An SD D where s has been refined by α*

```
1: for every v in extent(s) do
2:    candidate := ∅
3:    compute the α neighbourhood of v: N_α(v)
4:    for every node s' in D such that axpre(s') = α  do
5:       let w be a node in extent(s')
6:       compute the α neighbourhood of w: N_α(w)
7:       if v ~^α w (i.e., N_α(v) ~ N_α(w))  then
8:          candidate := s'
9:       end if
10:   end for
11:   if candidate = ∅ then
12:      create a new node candidate in D
13:      axpre(candidate) := α
14:      λ^D(candidate) := λ(v)
15:   end if
16:   move v from extent(s) to extent(candidate)
17: end for
18: let S be the set of nodes connected to s
19: for every node s'' in S do
20:   add edges ⟨candidate, s''⟩ and ⟨s'', candidate⟩ if conditions in Definition 4.2 are satisfied
21: end for
22: delete s and all its incoming and outgoing edges from D
```

LPath distinguishable entails computing the bisimulation between them and then comparing the result to their LPath sets.

There is a class of neighbourhoods, however, that are guaranteed to be always LPath distinguishable. For neighbourhoods in that class, we can bypass the bisimulation computation and obtain the EEs directly from the LPath sets. Such is the class of the *tree neighbourhoods*. How to characterize other classes of LPath distinguishable neighbourhoods without resorting to bisimulation remains an open problem.

6.2 Changing descriptions

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. The refinement of an SD node is computed by changing the AxPRE of the node.

Algorithm 6.5 computes a refinement of an SD node s by changing its former AxPRE into a new AxPRE α. Loop 1-17 iterates over every node v in the extent of s and moves each v to the corresponding new SD nodes one by one. For every SD node s' with AxPRE α, loop 4-10 takes any node w in its extent and compares by bisimulation $\mathcal{N}_\alpha(v)$ with $\mathcal{N}_\alpha(w)$. (All nodes in the extent of s' are bisimilar by α, so any one suffices for the test.) If they are bisimilar (step 7), then s' is a candidate and v is moved to the extent of s' (step 16). If they are not bisimilar, then a new candidate node is created (step 12) with AxPRE α (step 13) and label $\lambda(v)$ (step 14). After every node v in the extent of s has been moved to some new node, loop 19-21 creates new edges if necessary and delete the original SD node

ALGORITHM 6.6. *StabilizeEdge*(D, s_i, s_j)

Input: *An SD D containing a non forward-stable edge $e = \langle s_i, s_j \rangle$ with label axis*
Output: *An SD D where e has been replaced by forward-stable $e' = \langle s_i', s_j \rangle$*

1: $\alpha := axpre(s_i)|axis.axpre(s_j)$
2: ***for*** *every node s in D such that $axpre(s) = axpre(s_i)$* ***do***
3: *RefineNode*(D, s, α)
4: ***end for***

ALGORITHM 6.7. *UnfoldEdge*$(D, s_i, axis)$

Input: *An SD D, a node s_i such that there exists a non forward-stable $e = \langle s_i, s_i \rangle$ with label axis*
Output: *The SD D where any edge $e = \langle s_i, s_i \rangle$ with label axis is forward-stable*

1: $\alpha := axpre(s_i)|axis^*$
2: ***for*** *every node s in D such that $axpre(s) = axpre(s_i)$* ***do***
3: *RefineNode*(D, s, α)
4: ***end for***

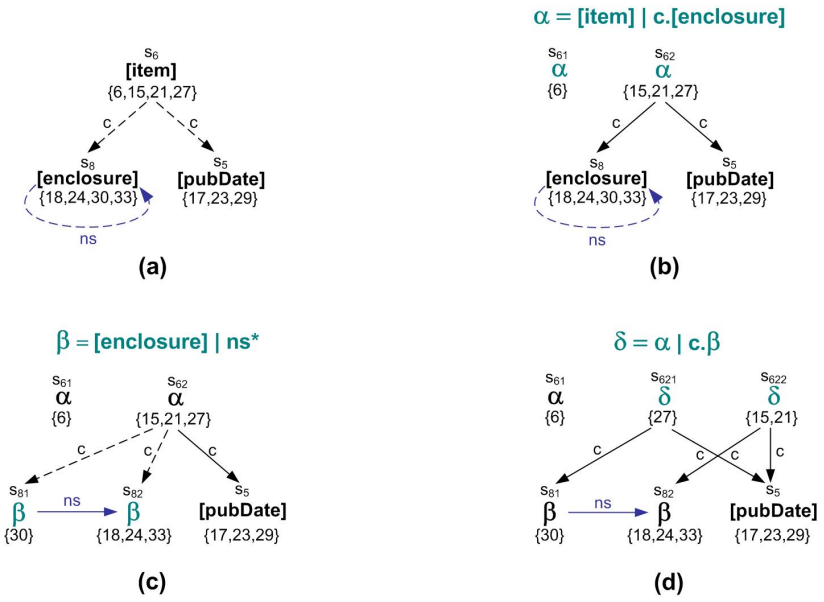

Fig. 12. Stabilization of the $c[pubDate]|c[enclosure].ns$ neighbourhood of s_6

s, whose extent is now empty. Note that the algorithm in fact changes one of the AxPREs in the definition of the SD, so all nodes that share the modified AxPRE will be affected.

Alternatively, the description provided by an SD node can be changed by stabilizing the edges (according to Definition 4.2) in the summary neighbourhood of the SD node. The goal of this particular refinement operation is to make forward-stable all edges in the neighbourhood of an SD node. As usual, the neighbourhood to stabilize is given by an AxPRE. DescribeX uses Algorithm 6.6 to stabilize an

edge linking two different nodes and Algorithm 6.7 to stabilize an edge forming a loop. Both algorithms essentially reduce edge stabilization to refinement: step 1 composes a new AxPRE and step 3 refines the affected nodes by calling Algorithm 6.5. The next example illustrates how non forward-stable edges are stabilized by Algorithms 6.6 and 6.7.

Example 6.8. Consider the $\langle s_6, s_8 \rangle$ edge from Figure 12 (a). This edge is not forward-stable because node 6 is not related to any node in $extent(s_8)$ via the c axis. Edge stabilization (Algorithm 6.6) creates two nodes, s_{61} and s_{62}, such that $extent(s_{61}) = \{6\}$ and $extent(s_{62}) = \{15, 21, 27\}$. Since $axpre(s_6) = [item]$ and $axpre(s_8) = [enclosure]$, then line 1 of Algorithm 6.6 creates the new AxPRE $[item]|c[enclosure]$, which is then used to refine all $[item]$ nodes (in this case just node s_6) in lines 2 and 3. The new edge $\langle s_{62}, s_8 \rangle$ is forward-stable. The result of stabilizing edge $\langle s_4, s_6 \rangle$ is shown in Figure 12 (b).

Consider now the ns loop on node s_8. The edge is not forward-stable because some element in $extent(s_8)$ is not in a ns relation with elements in the same extent (for instance, there is no node that is the next sibling of node 18). Since $axpre(s_8) = [enclosure]$, then line 1 of Algorithm 6.7 creates the new AxPRE $[enclosure]|ns^*$, which is then used to refine all $[enclosure]$ nodes (just node s_8 in our example) in lines 2 and 3. The new edges are forward-stable. The result of unfolding ns loop on s_8 is shown in Figure 12 (c). This last stabilization created two new non forward-stable edges that can be in turn stabilized by applying Algorithm 6.6 one more time. The final stable neighbourhood appears in Figure 12 (d). □

Refinements can also be the result of adapting an SD to an XPath expression. DescribeX can transform the structural subquery of an XPath expression Q (the expression that results from removing all non-structural predicates such as those containing functions) into an equivalent AxPRE α. Once DescribeX has computed α, it needs to find the SD node whose AxPRE contains α in order to get the candidate documents for evaluating Q. Candidate documents are those that are guaranteed to provide a non-empty answer for the structural subquery of Q. If there is an SD node s with the exact AxPRE α, then all documents in the extent of s are in fact candidate documents. In contrast, if s has an AxPRE α' *containing* α, DescribeX can *adapt* the SD by *refining* s *with* α and then get the candidate documents as in the previous case. Note that, by adapting the SD to the structural subquery, DescribeX has found a restricted superset of the answer and hence has considerably reduced the search space for computing the entire query. Once the candidate documents are found, finding the answer documents entails running Q on all candidates. (See [Consens and Rizzolo 2007] for a detailed study on how to evaluate XPath queries on SDs.)

7. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

In this section, we present the results of an empirical study we conducted using the DescribeX framework. The study evaluates the performance of the initial p^* SD construction and the feasibility of two approaches, materialized and virtual, for computing extents and edges in DescribeX's main exploration operations, refinement and stabilization. From nodes in the initial p^* SD we performed three sets of experiments: c^* refinements, AxPRE refinements and edge stabilizations. The

first one is an example of a refinement easy to express, but relatively expensive to compute, since the number of different outgoing label paths in an axis graph (captured by c^*) is usually large for any given node. In addition, the c^* allows us to evaluate DescribeX with a common axis used throughout the summary literature. In contrast, AxPRE refinements and stabilizations are intended to test operations unique to DescribeX by specifying more expressive AxPREs including label selection, disjunctions and/or combinations of axes. The overall objective of this study is twofold. First, to understand how key parameters, namely extent size (documents and elements) and number of resulting new SD nodes and edges, impact each operation. Second, to determine what method performs better under what kind of conditions. The experiments demonstrate that DescribeX easily scales up to gigabyte sized XML collections with important performance results.

Before introducing the experimental results, we discuss how DescribeX is implemented in the DescribeX engine.

7.1 DescribeX Engine

The DescribeX 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. Such an approach supports the interactive creation and refinement of AxPRE SDs for large collections of XML documents.

The DescribeX engine is implemented in Java using Berkeley DB Java Edition[17] to store and manage indexed collections (tables). The implementation can invoke an arbitrary JAXP 1.3[18] XPath processor for the evaluation of XPath EEs. JAXP is an implementation independent portable API for processing XML with Java. For our experiments, we employed the Saxon[19] XPath processor. Saxon conforms to the XPath 2.0 standard set by the W3C [W3C 2007].

We tested two different storage strategies for the extents. One is based on materializing the SD partitions whereas the other is a virtual approach that relies exclusively on XPath EEs to compute extents and edges. When materializing the partitions, extents are stored in an indexed table named `elemDB` with schema `elemDB`(`SID`, `docID`, `endPos`, `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 positions, in the document, where the element starts and ends, respectively; they are used to quickly determine the ancestor-descendant relationship between any pair of XML elements. This encoding is a variant of Dietz's numbering scheme [Dietz 1982] in which nodes are labeled with their preorder and postorder traversals [Kaplan et al. 2002]. `SID2` allows us to maintain an SID for a second SD. For testing the virtual approach, the DescribeX engine stores a `docDB` table instead of the `elemDB` table described

[17]http://www.oracle.com/technology/products/berkeley-db/je/
[18]http://jaxp.dev.java.net/1.3/
[19]http://saxon.sourceforge.net/

Table I. Test collections

Name	Size (MB)	#Docs	#SD Nodes		Load Time (s)	
			p^*	label	p^*	label
RSS2	210	9600	1058	301	64.2	41.4
Wiki5	545	30000	15602	259	438.6	175.7
Wiki45	4520	659388	66073	1245	8089.1	6201.2

above. The schema of the `docDB` table is `docDB(SID, docID)`, 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 EE of s in order to get the extent of s. The DescribeX engine keeps the SD graph 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. The XPath EEs are stored in a separate XML file.

We discuss next how a refinement is computed from the data structures just described. Suppose that SD node s_i is one of the refinements of SD node s. The extent of s_i is computed by evaluating $ee(s_i)$ on the set of documents that contain elements in the extent of s. This set of documents are obtained from `ElemDB` (if the 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 or from `ElemDB` when the extent is materialized. In order to update the edges with virtual extents, DescribeX needs to check whether there is an $axis$ edge between s_i and a set of candidate SD nodes $c_1, \ldots, c_n$ that are mapped to s in `axisMap`. This is performed by evaluating the XPath expression $ee(s_i)/axis :: * \cap ee(c_j)$, for $1 \leq j \leq n$, which tests whether it is possible to reach elements in the extent of c_j from elements in the extent of s_i by following $axis$. If the evaluation of the expression is not empty, then there exists an edge from s_i to c_j, otherwise there is no edge. When using materialized extents, in contrast, DescribeX simply computes a merge of the `ElemDB` using the `startPos` and `endPos` attributes to check for containment (in case of fc, c, p, a, and d axes) or precedence (for ns, fs, f, and p axes).

7.2 Experimental Setup

Our experiments were conducted over three collections of documents. Table I summarizes the size and number of documents in each collection, and the number of nodes and load times for the p^* and label SDs, which includes computing the SD graph and the partitions, and storing the extents in the `ElemDB` table. For measuring times, 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 out on a Windows XP machine with a 2.4GHz Intel Core 2 Quad processor, and the JVM was allocated 1 GB of RAM.

The selected collections have different characteristics, namely total size, size and number of individual documents, and document heterogeneity. The first collection

Table II. Selected p^* SD nodes and EEs from the RSS collection

p^* SD Node	Extent Expression (EE)	RSS2 Extent Size	
		Docs	Elems
r_{468}	/rss/channel/image	3296	3296
r_{449}	/rss/channel/item	6509	90583
r_{653}	/rss/channel/item/body	18	320
r_{452}	/rss/channel/item/description	6253	82022

Table III. Selected p^* SD nodes and EEs from the Wikipedia collections

p^* SD Node	Extent Expression (EE)	Wiki5 Extent Size		Wiki45 Extent Size	
		Docs	Elems	Docs	Elems
w_{372}	/article/body/section/section/section/figure	252	522	898	2166
w_{199}	/article/body/section/p/sub	463	2194	1479	6963
w_{333}	/article/body/section/section/section/section	128	500	736	3714
w_{967}	/article/body/template/template/wikilink	155	241	2330	3662

(RSS2) was obtained by collecting RSS feeds from thousands of different sites listed in specialized search engines like Fagan Finder[20] and Syndic8[21], which index news feeds on a wide variety of topics. The second and third collections (Wiki5 and Wiki45, respectively) were created from the Wikipedia XML Corpus provided in INEX 2006 [Denoyer and Gallinari 2006]. Document sizes in the collections ranges from 1 KB to a few hundred KBs. The number of nodes in both p^* and label SDs provide a measure of heterogeneity and structural complexity. Wiki45 is the most heterogenous collection with over one thousand different labels and over 66 thousand different label paths from the root.

7.3 Refinements

We tested the performance of two operations: refinements and stabilization. We report here results for refinements, stabilizations are discussed in the next section.

Times were measured on an initial p^* SD. We identified a number of parameters that influence the performance of the operations: extent size, location in the SD and number of new SD nodes resulting from the operation. Extents were chosen based on two criteria: selectivity of the extents (i.e., size) and location in the summary. For testing selectivity, we ran experiments with extents varying several orders of magnitude in size, both in terms of number of documents and elements. The location in the summary has two components: the depth of the initial p^* nodes (closer and farther from the root, given by the length of the EE) and how far from the initial p^* node the refinements reach (which could be all the way down to the leaves in some cases, such as the c^* and d AxPREs). The location in the summary also affects the number of resulting new SD nodes (and thus edges) in the c^* refinement. These parameters allowed us to probe a wide range of summary and instance fragments.

[20]http://www.faganfinder.com
[21]http://www.syndic8.com

Table IV. RSS2 c^* refinements

p^* SD Node	# New Nodes	Times (s)		
		Extent	M. Edges	V. Edges
r_{468}	7	100.1	1.7	119.1
r_{449}	201	575.2	23.0	14211.0
r_{653}	42	3.7	0.8	47.8
r_{452}	3	185.1	4.5	173.3

Table V. Wiki5 c^* refinements

p^* SD Node	# New Nodes	Times (s)		
		Extent	M. Edges	V. Edges
w_{372}	16	25.7	0.6	270.1
w_{199}	4	29.9	3.9	418.5
w_{333}	61	79.1	8.7	2059.8
w_{967}	6	10.4	2.5	225.5

Table VI. Wiki45 c^* refinements

p^* SD Node	# New Nodes	Times (s)		
		Extent	M. Edges	V. Edges
w_{372}	37	446.1	9.1	1003.0
w_{199}	14	748.7	24.4	1744.8
w_{333}	203	537.5	37.2	12275.9
w_{967}	8	552.3	17.1	1283.6

Tables II and III show the SIDs, EEs and extent sizes of the selected p^* SD nodes in our test collections. For instance, r_{468} corresponds to the p^* SD node that has /rss/channel/image as its EE in the RSS2 collection. Refinements were selected to test a wide range of selectivity on the extents: smallest and largest extents are three orders of magnitude apart with respect to number of documents and elements, ranging from 18 to 6509 documents and from 320 to 90583 elements. The SD nodes reported in these tables are those we use as a starting point for refinements and edge stabilization in our experiments reported below.

Tables IV, V and VI report c^* refinement times for the selected p^* SD nodes. We choose the c^* refinement to show the performance of an AxPRE $(p^*|c^*)$ that captures a well-known homogeneous summary from the literature (see Figure 10) that is expensive to compute in general. The number of new SD nodes created by the refinements (which is the same as the number of EEs evaluated) are reported in the **# New Nodes** columns. The number of new SD nodes span two orders of magnitude, from 3 in the r_{452} refinement (Table IV) to 203 as a result of the w_{333} refinement (Table VI). Times reported under **Extent** comprise locating the affected files using the SD, opening them and evaluating the EE in order to update the materialized extent information in the `ElemDB` table. For instance, the c^* refinement partitions node r_{449} into 201 new SD nodes, which requires the evaluation of 201 XPath expressions on the documents in the extent of r_{449}. Times reported under **M. Edges** and **V. Edges** correspond to edge computation using the materialized and the virtual approaches, respectively. Columns **M. Edges** and **V. Edges** give us an idea of how much overhead DescribeX incurs on the edges. It is easy to see

Table VII. AxPRE refinements for the RSS2 collection

p^* SD Node	Refining AxPRE	RSS2		
		Extent Size		Times
		Docs	Elems	(s)
r_{468}	c[title] | c[link] | c[width] | c[height]	172	172	3.9
r_{449}	c[enclosure].fs[enclosure].fs[enclosure]	9	37	10.8
r_{449}	c[pubDate]	3552	53148	9.9
r_{653}	d[img]	12	119	0.4
r_{452}	fs[link]	688	13885	10.1

Table VIII. AxPRE refinements for the Wikipedia collections

p^* SD Node	Refining AxPRE	Wiki5			Wiki45		
		Extent Size		Times	Extent Size		Times
		Docs	Elems	(s)	Docs	Elems	(s)
w_{372}	c[caption]	242	522	1.7	1350	1929	36.2
w_{199}	c[sub].c[sub]	1	1	2.1	3	3	37.1
w_{333}	c[p].fs[p].fs[p]	39	79	1.0	155	320	26.5
w_{967}	fs[collectionLink]	24	34	1.4	53	64	51.2

from the tables that the virtual approach tends to have worse performance than the materialized one. The reason for this is that computing edges with EEs always involves at least one intersection operation, which becomes quite expensive when the expressions are complex.

In general, the extent size affects the refinement times more than the number of new nodes created. Consider SD nodes r_{449} and r_{452} for instance, with roughly the same number of documents and elements in the their extents: 6509 vs. 6253 documents and 90583 vs. 82022 elements, respectively (Table II). Let us take a look at the refinements information in Table IV. Despite having almost a two orders of magnitude difference in the number of EEs needed to compute the refinement (201 vs. 3, respectively), the time difference between them for computing extents is only a little over three times (575.2 s. vs. 185.1 s., respectively). In contrast, consider SD nodes w_{199} and w_{967} from Table V. Computing the refinement of the one with the smaller number of EEs (w_{199}) takes almost three times longer than the other (w_{967}), mainly because the extent size of the former is much larger than that of the latter: 463 vs. 155 in terms of documents and 2194 vs. 241 in terms of elements (see Wiki5 extent sizes in Table III). Clearly, the number of documents that need to be opened for computing the refinement and the number of elements involved weighs more than the number of EEs to be evaluated in those documents.

Tables VII and VIII report refinements that are intended to test functionality unique to DescribeX. Such refinements are specified by AxPREs containing novel axes (like *fs* and *d*), label selection and disjunctions. The tables show the refining AxPRE for each p^* SD node, the number of documents and elements that contain neighbourhoods matching the entire AxPRE (**Docs** and **Elems** columns, respectively), together with how long it takes to compute the extent (**Times** column). For any given expression, the number of elements with either empty neighbourhoods or matching prefixes of the AxPRE is the complement of the number reported under **Elems**. For instance, the first r_{449} row of Table VII indicates that 37 elements in 9

documents have exact $c[enclosure].fs[enclosure].fs[enclosure]$ neighbourhoods and obtaining them from the r_{449} extent takes 10.8 seconds. In addition, we know that the number of elements either matching prefixes or with empty neighbourhoods is 90546, which comes from the number in column **Elems** and row r_{449} in Table II (90583) minus the number in column **Elems** and row r_{449} in Table VII (37). Such subtraction would not be meaningful for the **Docs** columns because the same document may contain elements in different extents (remember that an SD contains a partition of elements, not documents, so document extents may in fact overlap).

The expressions were chosen with practical scenarios in mind, like the motivating example of Sections 1.1. For example, the refining AxPRE of SD node r_{468} will distinguish $/rss/channel/image$ elements that have a *title*, a *link*, a *width* and a *height* from those that do not. Although these elements are required for better formatting and linking an image, they are not always present in RSS feeds and a robust syndication application would process differently all images that have the additional information. Another useful information is to distinguish between multiple instances of the same element. For example, the first refining AxPRE of SD node r_{449} will cluster in different nodes $/rss/channel/item$ elements based on the number of enclosure elements they have. This happens when an item has the same content in different media formats, which need to be handled differently by some applications.

These results suggest that, even though computing generic refinements like c^* may be expensive, more specific refinements can be performed in less than a minute and most of them in just a few seconds for under-a-gigabyte collections.

7.4 Edge stabilization

We report here the experimental results for stabilization of selected SD edges. As in the refinement experiments, we start from selected nodes in an initial p^* SD. The edges to stabilize were chosen in order to get extents that vary several orders of magnitude in size. We stabilize two different edges for most of our selected p^* SD nodes. After one edge stabilization, the resulting SD node that does not have the stabilized edge is indicated by an SID with an apostrophe (in Tables IX, X, and XI). The second edge stabilized always corresponds to a node with an apostrophe from the previous stabilization.

Tables IX, X, and XI report edge stabilization times and extent sizes for the selected p^* SD nodes. The edge stabilized is indicated in the tables by an AxPRE containing the axis and the label of the target node. The four **Extent Sizes** columns show the number of document and elements that do contain the edge (under **Stable Edge**) and the number of those that do not (under **No Edge**). The times reported correspond to computing the extents in the materialized approach, as explained in the previous section for refinements. The times for computing the edges are not reported because it does not involve additional computation (once we know the two resulting extents, we just draw an edge to the stable extent). For instance, the first edge stabilized from node r_{449} (Table IX) was the *ps* edge to an *item* node, which resulted in two SD nodes: one containing a *stable ps* edge with 84063 elements in its extent, and another one (r'_{449}) with *no edge* and 6520 elements. From node r'_{449}, we stabilize then the *c* edge to a *body* node obtaining again two nodes: one with a stable *c* edge with 15 elements in its extent, and the

Table IX. RSS2 edge stabilization

p^* SD Node	Edge Stabilized	Extent Sizes				Times (s)
		Stable Edge		No Edge		
		Docs	Elems	Docs	Elems	
r_{468}	c[description]	492	492	2804	2804	0.5
r'_{468}	c[link]	2792	2792	12	12	0.2
r_{449}	ps[item]	6263	84063	6509	6520	2.9
r'_{449}	c[body]	15	15	6494	6505	3.7
r_{653}	d[font]	8	14	18	306	0.3
r'_{653}	d[table]	7	7	3	14	0.2
r_{452}	c[br]	12	12	6249	81968	5.9

Table X. Wiki5 edge stabilization

p^* SD Node	Edge Stabilized	Extent Sizes				Times (s)
		Stable Edge		No Edge		
		Docs	Elems	Docs	Elems	
r_{372}	d[collectionLink]	125	207	169	315	0.6
r'_{372}	d[small]	2	4	169	311	0.5
r_{199}	c[sub]	3	3	462	2191	0.8
r'_{199}	c[small]	5	35	458	2156	0.8
r_{333}	c[outsideLink]	7	12	126	488	0.7
r'_{333}	c[unknownLink]	10	14	123	474	0.6
r_{967}	c[template]	4	5	151	236	0.6
r'_{967}	c[sup]	66	123	85	113	0.5

Table XI. Wiki45 edge stabilization

p^* SD Node	Edge Stabilized	Extent Sizes				Times (s)
		Stable Edge		No Edge		
		Docs	Elems	Docs	Elems	
w_{372}	d[collectionLink]	335	592	695	1574	2.3
w'_{372}	d[small]	3	5	694	1569	2.2
w_{199}	c[sub]	28	33	1469	6930	5.4
w'_{199}	c[small]	18	83	1454	6847	5.4
w_{333}	c[outsideLink]	34	83	724	3631	3.6
w'_{333}	c[unknownLink]	68	131	705	3500	3.8
w_{967}	c[template]	26	27	2304	3635	3.5
w'_{967}	c[sup]	174	246	2130	3389	3.4

other one with 6505 elements and no edge. The time for computing the *ps* edge stabilization is 2.9 seconds. The times for the *c* edge stabilization are 10.9 and 3.7 seconds respectively.

Our results show that DescribeX can provide interactive response times (from sub-second to just a few seconds) for all edge stabilizations tested when using the materialized approach for computing the extents. This is compelling evidence that DescribeX can be used in scenarios in which SDs need to be manipulated interactively in order to selectively explore the structure of an XML collection (e.g., providing subscribers with customized content solutions aggregated from thousands of content providers, as described in Section 1.1).

8. CONCLUSION AND FUTURE WORK

DescribeX is a powerful new tool for describing the actual heterogeneous structure of web collections of XML documents. Understanding the metadata structure of such collections is fundamental for writing meaningful XPath queries. We proposed a novel framework for declaratively describing the structure of a web collection based on highly customizable summaries that can be conveniently tailored by axis paths regular expressions (AxPREs). Our main results demonstrate the scalability of AxPRE summary refinements and stabilization (the key enablers for tailoring summaries) using gigabyte XML collections.

Familiar research issues can be re-visited in the context of AxPRE summaries. For instance, providing guidelines for selecting good summaries (similar to schema design) and inferring general and succinct AxPRE expressions from an XML collection (similar to DTD inference from instances). In this direction, extending recent proposals for discovering and modeling XML data redundancies via functional dependencies [Yu and Jagadish 2006a; 2008] from XML schemas to DescribeX summaries seems promising. The development of benchmark tasks for evaluating and comparing solutions designed to help in large-scale document collection management and exploration is also an important area to pursue.

In the context of XML messaging, we came across the problem of creating schema mappings when the schemas are too general to create meaningful mappings between them. The schema mapping problem consists of defining correspondences between two schemas in order to translate data from one to the other [Miller et al. 2000; Popa et al. 2002]. An interesting research direction would be to develop a strategy to do summary mapping in the same spirit as schema mapping, perhaps using EEs definitions to create the correspondences in XPath.

We also plan to study the impact of adjusting the workload (e.g, by finding frequent patterns), and also how to optimize SD selection given budget constraints. There are also opportunities for exploiting the flexibility available in AxPRE summaries in the context of the more traditional summary applications to indexing, selectivity estimation, and query optimization.

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 introduced in this paper (selective bisimilarity applied to subgraphs described by AxPREs) can also find novel applications in such areas.

REFERENCES

Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N., and Srivastava, D. 2002. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 18th International Conference on Data Engineering*. 141–152.

Ali, M. S., Consens, M. P., Gu, X., Kanza, Y., Rizzolo, F., and Stasiu, R. K. 2006. Efficient, effective and flexible XML retrieval using summaries. In *Proceedings of the 5th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2006*. Lecture Notes in Computer Science, vol. 4518. Springer, 89–103.

Ali, M. S., Consens, M. P., and Khatchadourian, S. 2007. XML retrieval by improving structural relevance measures obtained from summary models. In *Proceedings of the 6th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX 2007*. Lecture Notes in Computer Science. Springer, 34–48.

Ali, M. S., Consens, M. P., Khatchadourian, S., and Rizzolo, F. 2008. DescribeX: interacting with AxPRE summaries. In *Proceedings of the 24th International Conference on Data Engineering (Demonstrations)*. 1540–1543.

Amato, G., Debole, F., Rabitti, F., Savino, P., and Zezula, P. 2004. A signature-based approach for efficient relationship search on XML data collections. In *Proceedings of the 2nd International XML Database Symposium, XSym*. 82–96.

Balmin, A., Ozcan, F., Beyer, K. S., Cochrane, R., and Pirahesh, H. 2004. A framework for using materialized XPath views in XML query processing. In *Proceedings of the 30th International Conference on Very Large Data Bases*. 60–71.

Barta, A., Consens, M. P., and Mendelzon, A. O. 2005. Benefits of path summaries in an XML query optimizer supporting multiple access methods. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 133–144.

Bex, G. J., Neven, F., Schwentick, T., and Tuyls, K. 2006. Inference of concise DTDs from XML data. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 115–126.

Bruno, N., Koudas, N., and Srivastava, D. 2002. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. 310–321.

Buneman, P., Choi, B., Fan, W., Hutchison, R., Mann, R., and Viglas, S. 2005. Vectorizing and querying large XML repositories. In *Proceedings of the 21st International Conference on Data Engineering*. 261–272.

Chien, S.-Y., Vagena, Z., Zhang, D., Tsotras, V. J., and Zaniolo, C. 2002. Efficient structural joins on indexed XML documents. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 263–274.

Chung, C.-W., Min, J.-K., and Shim, K. 2002. APEX: An adaptive path index for XML data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. 121–132.

Clark, J. and Makoto, M. 2001. RELAX NG specification. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

Consens, M. P. and Milo, T. 1994. Optimizing queries on files. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. 301–312.

Consens, M. P. and Rizzolo, F. 2007. Fast answering of XPath query workloads on web collections. In *Proceedings of the 5th International XML Database Symposium, XSym*. 31–45.

Consens, M. P., Rizzolo, F., and Vaisman, A. A. 2008. AxPRE summaries: Exploring the (semi-)structure of XML web collections. In *Proceedings of the 24th International Conference on Data Engineering*. 1519–1521.

Cooper, B. F., Sample, N., Franklin, M. J., Hjaltason, G. R., and Shadmon, M. 2001. A fast index for semistructured data. In *Proceedings of the 27th International Conference on Very Large Data Bases*. 341–350.

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

Dietz, P. F. 1982. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, San Francisco, California, USA*. 122–127.

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

Fletcher, G. H. L., Gucht, D. V., Wu, Y., Gyssens, M., Brenes, S., and Paredaens, J. 2007. A methodology for coupling fragments of XPath with structural indexes for XML documents. In *Proceedings of the 11th International Symposium on Database Programming Languages, DBPL 2007*. 48–65.

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

Goldman, R. and Widom, J. 1997. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*. 436–445.

He, H. and Yang, J. 2004. Multiresolution indexing of XML for frequent queries. In *Proceedings of the 20th International Conference on Data Engineering*. 683–694.

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Jiang, H., Lu, H., Wang, W., and Ooi, B. C. 2003. XR-Tree: Indexing XML data for efficient structural joins. In *Proceedings of the 19th International Conference on Data Engineering*. 253–263.

Jiang, H., Wang, W., Lu, H., and Yu, J. X. 2003. Holistic twig joins on indexed XML documents. In *Proceedings of the 29th International Conference on Very Large Data Bases*. 273–284.

Kaplan, H., Milo, T., and Shabo, R. 2002. A comparison of labeling schemes for ancestor queries. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. 954–963.

Kaushik, R., Bohannon, P., Naughton, J. F., and Korth, H. F. 2002. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. 133–144.

Kaushik, R., Bohannon, P., Naughton, J. F., and Shenoy, P. 2002. Updates for structure indexes. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 239–250.

Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E. 2002. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering*. 129–140.

Kazai, G., Gövert, N., Lalmas, M., and Fuhr, N. 2003. The INEX evaluation initiative. In *Intelligent Search on XML Data*. 279–293.

Kha, D. D., Yoshikawa, M., and Uemura, S. 2001. An XML indexing structure with relative region coordinate. In *Proceedings of the 17th International Conference on Data Engineering*. 313–320.

Lakshmanan, L. V., Wang, H. W., and Zhao, Z. J. 2006. Answering tree pattern queries using views. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 571–582.

Li, Q. and Moon, B. 2001. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*. 361–370.

Li, Y., Yu, C., and Jagadish, H. V. 2008. Enabling Schema-Free XQuery with meaningful query focus. *The International Journal on Very Large Data Bases 17,* 3, 355–377.

Lu, J., Ling, T. W., Chan, C. Y., and Chen, T. 2005. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 193–204.

Mandhani, B. and Suciu, D. 2005. Query caching and view selection for XML databases. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 469–480.

Martens, W., Neven, F., Schwentick, T., and Bex, G. J. 2006. Expressiveness and complexity of XML schema. *ACM Transactions on Database Systems (TODS) 31,* 3, 770–813.

Mendelzon, A. O. and Wood, P. T. 1995. Finding regular simple paths in graph databases. *SIAM Journal on Computing 24,* 6, 1235–1258.

Miller, R. J., Haas, L. M., and Hernández, M. 2000. Schema mapping as query discovery. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 77–88.

Milo, T. and Suciu, D. 1999. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory*. 277–295.

Murata, M., Lee, D., Mani, M., and Kawaguchi, K. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology (TOIT) 5,* 4, 660–704.

Nestorov, S., Ullman, J. D., Wiener, J. L., and Chawathe, S. S. 1997. Representative objects: Concise representations of semistructured, hierarchial data. In *Proceedings of the 13th International Conference on Data Engineering*. 79–90.

Paige, R. and Tarjan, R. E. 1987. Three partition refinement algorithms. *SIAM Journal on Computing 16,* 6, 973–989.

Polyzotis, N. and Garofalakis, M. N. 2006a. XCluster synopses for structured XML content. In *Proceedings of the 22nd International Conference on Data Engineering*.

Polyzotis, N. and Garofalakis, M. N. 2006b. XSketch synopses for XML data graphs. *ACM Transactions on Database Systems (TODS) 31,* 3, 1014–1063.

Polyzotis, N., Garofalakis, M. N., and Ioannidis, Y. E. 2004. Approximate XML query answers. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 263–274.

Popa, L., Velegrakis, Y., Miller, R. J., Hernández, M. A., and Fagin, R. 2002. Translating web data. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 598–609.

Qun, C., Lim, A., and Ong, K. W. 2003. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 134–144.

Rao, P. and Moon, B. 2004. PRIX: Indexing and querying XML using prufer sequences. In *Proceedings of the 20th International Conference on Data Engineering*. 288–300.

Rizzolo, F. 2008. DescribeX: A framework for exploring and querying XML web collections. Ph.D. thesis, University of Toronto. CoRR arXiv:0807.2972v1, http://arXiv.org/abs/0807.2972.

Rizzolo, F. and Mendelzon, A. O. 2001. Indexing XML data with ToXin. In *Proceedings of 4th International Workshop on the Web and Databases*. 49–54.

Rizzolo, F. and Vaisman, A. A. 2008. Temporal XML: Modeling, indexing, and query processing. *The International Journal on Very Large Data Bases 17,* 5, 1179–1212.

Samavi, R., Consens, M., Khatchadourian, S., and Topaloglou, T. 2007. Exploring PSI-MI XML collections using DescribeX. *Journal of Integrative Bioinformatics 4,* 3.

Santoro, N. and Khatib, R. 1985. Labelling and implicit routing in networks. *The Computer Journal 28*, 5–8.

Vagena, Z., Moro, M. M., and Tsotras, V. J. 2004. Efficient processing of XML containment queries using partition-based schemes. In *Proceedings of the 8th International Database Engineering and Applications Symposium, IDEAS 2004*. 161–170.

W3C. 1999. XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath.

W3C. 2004. XML Schema. http://www.w3.org/TR/xmlschema-0.

W3C. 2006. Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml.

W3C. 2007. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20.

Wang, H., Park, S., Fan, W., and Yu, P. S. 2003. ViST: A dynamic index method for querying XML data by tree structures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 110–121.

Wang, W., Jiang, H., Lu, H., and Yu, J. X. 2003. PBiTree coding and efficient processing of containment joins. In *Proceedings of the 19th International Conference on Data Engineering*. 391–.

Xu, W. and Özsoyoglu, Z. M. 2005. Rewriting XPath queries using materialized views. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 121–132.

Yannakakis, M. 1990. Graph-theoretic methods in database theory. In *Proceedings of the 9th Symposium on Principles of Database Systems*. 230–242.

Yi, K., He, H., Stanoi, I., and Yang, J. 2004. Incremental maintenance of XML structural indexes. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 491–502.

Young-Lai, M. and Tompa, F. W. 2003. One-pass evaluation of region algebra expressions. *Information Systems 28,* 3, 159–168.

Yu, C. and Jagadish, H. V. 2006a. Efficient discovery of XML data redundancies. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 103–114.

Yu, C. and Jagadish, H. V. 2006b. Schema summarization. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 319–330.

Yu, C. and Jagadish, H. V. 2007. Querying complex structured databases. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 1010–1021.

Yu, C. and Jagadish, H. V. 2008. XML schema refinement through redundancy detection and normalization. *The International Journal on Very Large Data Bases 17,* 2, 203–223.

Zhang, N., Kacholia, V., and Özsu, M. T. 2004. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proceedings of the 20th International Conference on Data Engineering*. 54–65.

Zhang, N., Özsu, M. T., Ilyas, I. F., and Aboulnaga, A. 2006. FIX: Feature-based indexing technique for XML documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 259–270.

Received July 2008; revised June 2009; accepted January 2010