EXsum - An XML Summarization Framework

José de Aguiar Moraes Filho
University of Kaiserslautern
67663 Kaiserslautern, Germany
aguiar@informatik.uni-kl.de

Theo Härder
University of Kaiserslautern
67663 Kaiserslautern, Germany
haerder@informatik.uni-kl.de

ABSTRACT [1]

We propose a new framework for the summarization of XML document properties called EXsum (Element-wise XML summarization), which can capture statistical information of all important XPath axes related to (the nodes having) the same element name in a document. Compared to conventional summaries, cardinality estimates for a richer spectrum of XPath/XQuery expressions can be provided for query optimization. For the important class of queries consisting of one or two location steps only, even accurate cardinalities are computed. Besides adequate storage consumption, it provides fast access times which helps to keep the query optimization overhead low. Using a collection of XML documents embodying considerable structural variations, we have empirically analyzed the EXsum framework by running a large number of experiments in our XML native database management system called XTC. These evaluations clearly show the predominance of EXsum as to important aspects when compared to competitor approaches.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *Query processing.*

Keywords

XML synopsis. Cost-based XML query processing. XML summarization. XML databases. Query processing and optimization.

1. INTRODUCTION AND MOTIVATION

To process XML documents effectively and efficiently, suitable auxiliary structures or meta-data have to be provided to support such important tasks as formulation of meaningful queries, query translation and optimization, document storage, as well as concurrency control. The oldest proposal called DataGuides [5] primarily addressed query formulation and provision of some statistical data. In turn, a so-called path synopsis [6] was used for structure virtualization, i.e., to store only the content part of an XML document while its structure is recomputed on demand using this path synopsis. Another use of the path synopsis is to effectively enable XML concurrency control [7]. XPath/XQuery translation and optimization need specific measures concerning statistical information about the distribution of document nodes and their axis relationships to provide the query optimizer with estimates as precise as possible about selectivities of location steps and node cardinalities of XML subtrees. For that reason, we need in addition to a path synopsis as introduced in [6] a suitable summarization structure, in particular to support XML query optimization.

This latter task is challenging and still an open issue in database research. Numerous methods exist [1, 2, 4, 9, 10, 12, 16] where the main focus is on summaries only supporting *child* (*/*) and *descendant* (*//*) axes of path expressions. Some of these proposals favor tree-based structures [2, 4, 9] and others extend them to graph-based structures [10, 16]. So far, an approach widely accepted as the *superior* method does not exist. Usually, when a query execution plan (QEP) is generated, a summary has to be traversed for each location step—possibly several times—to locate statistical information in the course of the optimization process. When inappropriate data structures are used, such auxiliary tasks can have considerable impact on QEP preparation time, i.e., on the time to derive sufficiently optimized QEPs.

Usually, summarization structures, as published in the literature, are incomplete in the sense that they are limited to the estimation of location steps having type */x* and *//x* only, while other important axes such as parent, ancestor, or sibling are neglected. Besides missing structural support, estimation quality often has room for improvement. Thus, developing a summary for an XML document, which captures as accurately and completely as possible the distribution of nodes and their axis relationships and, at same time, provides fast access for the query optimizer, is considered the "holy grail" of XML cost-based query optimization.

One difficulty in designing suitable XML summaries is due to the XML document structure itself, which often exhibits a "fuzzy degree" of variability. It normally contains some parts of shallow or skinny subtrees whereas other parts may have deep or broad ones. Furthermore, some subtrees may exist having no text values at all and, at the same time, other subtrees having a plethora of text values. Another issue is the presence of recursive structures in the document. A structural recursion occurs when the same element name appears several times in a document path. In such cases, summarization of statistical values and, in turn, estimation of location steps becomes more complex. The situation is even worse when recursion also appears in the path expression (multiple location steps with the same element reference). In such cases, summary access is not bounded by the number of location steps in a path expression.

[1] Supported by the Rheinland-Pfalz cluster of excellence "Dependable adaptive systems and mathematical modeling" (see www.dasmod.de).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IDEAS08 2008, September 10-12, Coimbra [Portugal]
Editor: Bipin C. DESAI
Copyright 2008 ACM 978-1-60558-188-0/08/09...$5.00.

1.1 Related Work

We can roughly classify the published approaches for XML summarization into three categories: table-based [1, 12], tree-based [4, 9], and graph-based [10, 16]. Table-based approaches only record root-to-leave paths together with their number of occurrences. Because of this simple structure, they can be only used to estimate queries restricted to child axis clauses and they cannot provide estimations for queries having any predicates (e.g., /a/b[./c]). Tree-based and graph-based approaches can support more complex queries (e.g., such referring to descendant axes) and some simple predicates (e.g., predicates at the end of a path expression and limited to a single location step), because the hierarchical summarization structures can immediately support the computation of such estimates.

Most of the tree-based and graph-based methods apply some kind of search pruning technique (i.e., the summary is only inspected down to a given level). The main reason for such a pruning is to reduce the overhead of computing location-step-related estimates. However, it goes along with lower estimation quality or erroneous decision information, i.e., an increase of false positive hits may be observed especially when applied to recursive documents and/or queries. Normally, for descendant and ancestor axes, the more nodes a summary contains, the more time has to be spent on structure traversal and estimation computation.

Furthermore, there are hardly any approaches which try to summarize content and structure of XML documents [10]. So far, all methods considered can be hardly extended to capture statistical information on value distributions.

While all summarization methods proposed so far closely follow the structure of the XML document and are oriented towards document paths, they can, as a consequence, provide only limited axes support. For example, *parent*, *descendant*, *previous sibling,* and *following sibling* axes are not or only marginally supported and may be, therefore, approximated by a very low estimation quality. Hence, when such selectivity information is needed, some kind of guessing or heuristics are used to decide on an optimization step of a QEP. In contrast, we use an entirely novel approach by turning the summarization problem upside down and collecting statistical information per element name and, by doing that, we gain much more expressiveness for axes support and other statistical information. Although we use a completely different structure, we can yet capture the structural characteristics of an XML document.

Due to its structure, EXsum lends itself to extensibility, that is the capacity to extend the EXsum framework to capture additional statistical information of a document, e.g., value distributions of element names. This paper does not directly cope with extensibility, but we can throw a glance at it when we apply EXsum to estimate recursive path expressions and document paths.

Many methods proposed in the literature [1, 2, 4, 9, 10, 12, 16] require extensive traversals of their data structures when selectivity estimates have to be derived. Therefore, they provide unsatisfactory estimation times burdening the QEP optimization. Furthermore, they often consume large memory partitions to achieve some estimation quality guarantees. In contrast, estimates based on the EXsum structure are very fast and rely on a low memory footprint while providing superior estimation quality. With these features, EXsum has only minimum impact on the query optimization process.

1.2 Our Contribution

We propose a framework called EXsum that enables the element-wise XML summarization of axis relationships. In addition to structural document properties, EXsum can also capture statistical information concerning the content of a document. Each EXsum component bundles information of all axis relationships for a distinct element/attribute name in the document and is therefore called axes summary per element (ASPE for short). As a consequence, EXsum enables the cardinality estimation of each location step in a path expression by only accessing the related ASPE structures instead of fetching the complete EXsum information when building QEPs. Thus, EXsum use is bounded by the number of location steps.

In non-recursive XML documents, we can compute for arbitrary combinations of standard axes (*child*, *descendant*, *parent*, *ancestor*)

- accurate cardinalities when a unique element name is referenced in the final axis step of an arbitrarily long path expression or greatly improve its accuracy when it occurs somewhere in the middle of it
- always accurate cardinalities for one- and two-step path expressions and
- for n-step path expressions estimated cardinalities derived by interpolation of several EXsum results.

Furthermore, we have extended EXsum to capture axes information for recursive documents and sketch the way how to derive estimates from them for non-recursive and recursive path expressions.

Having empirically evaluated the best approaches known so far in [2, 16] in terms of storage consumption and estimation quality, we can show that EXsum not only provides much richer estimation capabilities, but is also at least competitive, if not superior to conventional approaches.

As compared to competitor approaches, we claim four prime differences (and advantages) for EXsum: fast access time; capacity to only access selected data partitions for cardinality estimation—as opposed to loading the entire summary into main memory; support for recursive queries; natural extensibility to gather value distributions.

In this paper, we primarily focus on estimating structural aspects of an XML document. Section 2 details the EXsum structure and its building process. Section 3 focuses on the estimation methods when non-recursive documents are present, whereas Section 4 extends these methods to recursive documents. Section 5 and 6 throw a brief glance at predicate estimation and EXsum maintenance in case of dynamic documents. In Section 7, we present an empirical evaluation of our proposal, before we conclude the paper.

2. EXSUM FRAMEWORK

We have designed the EXsum framework to enable the collection of accurate and complete document statistics. Furthermore, EXsum's extensible structure can be configured to match the query characteristics of specific workloads. According to our experience, *child* and *descendant* use is of prime importance, whereas *parent* and *ancestor* support is less urgently required. The remaining axes (*previous* (*sibling*), *following* (*sibling*)) are only of minor importance in practical applications. Therefore, we do not deal with them in the first place. However, EXsum easily allows to integrate the required statistical information and estimation procedures.

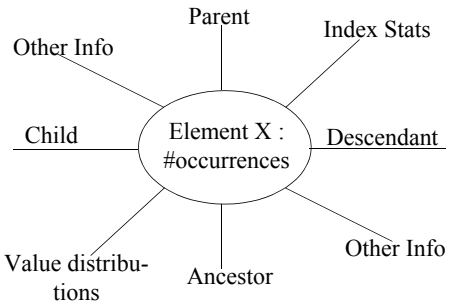

Figure 1. Possible structure of an ASPE node

An EXsum structure can be considered as a set of ASPE nodes where each ASPE node, in turn, represents a distinct element/attribute name in an XML document. An ASPE node is a compound of the element/attribute name, the total number of related node occurrences in the document, and a varying number of "spokes"—an ASPE node may be illustrated as a "spoked wheel" (see Figure 1). A spoke can represent structural (axis) distributions related to the element and also high-level concepts such as text value distributions and coarse-grained statistics (e.g., number of physical pages, index entries or depth, etc.). In other words, EXsum is extensible and configurable to summarize either structure-only or content-and-structure properties of an XML document. In the following sections, we detail the methods used by EXsum for structural summarization of a document and, in turn, for the related estimation of location step cardinalities.

2.1 Identifying Structural XML Properties

Usually, a path synopsis is kept to capture path-related information and often summary information for all elements and attributes of a document. To further explain its purpose, we refer to an XML document fragment (see Figure 2a) having a sufficiently rich structure needed for our discussion. In a path synopsis, all path instances of the document having the same sequence of element names are represented as a path class. For our document example, the path synopsis is illustrated in Figure 2b.

A cyclic-free XML schema captures all information needed for the path synopsis; otherwise, this data structure can be constructed while the document is stored. As shown in the following, such a concise description of the document structure is a prerequisite for effective query optimization and the other tasks sketched in Section 1. Because path instances of the same path class are often repeated very frequently, usually a path synopsis can be kept in a small memory-resident data structure (see also Table 1). For example in the popular *dblp* document, one of the dominating path classes */dblp/paper/author* has ~570,000 instances.

EXsum aggregates all statistical information per distinct element name. Although we have 14 elements in the path synopsis, only six distinct element names occur. Element names that occur more than once in the path synopsis are called *homonyms*. A *unique element name* such as *a*, *c*, or *u* in the path synopsis results in an unambiguous ASPE, which makes axes estimation very simple in some cases. In turn, a *homonym-free* document has only unique element names in its path synopsis, is non-recursive by definition, but may be an exception. In the typical case, a document contains a varying degree of homonyms, but most of its paths are *recursion-free*, i.e., homonyms[2] do not occur in the same path class. Hence, our reference document in Figure 2 is recursion-free. In contrast, we have to deal with *recursive paths* in a document as soon as an element name occurs more than once in a single path class.

2.2 Element-wise XML Summarization

To explain our approach, we refer to the document in Figure 2a with six distinct element names. Thus, EXsum has six ASPE nodes where the total number of elements with name *x* occurring in the document (occ(*x*)) is always recorded in the center of ASPE(*x*). Further, each of its spokes carries names and occurrences of all elements participating in the given axis relationships with *x*. This kind of recording is used to step-wise compute estimates for consecutive location steps of an XPath/XQuery [13, 14] expression. At building time of a path synopsis and an EXsum structure—typically in parallel when storing an incoming document sent by a client—, all statistical information contained in the ASPE nodes is derived. Depending on the types of queries anticipated, only spokes for the desired axes estimation support may be appended. Without loss of generality, we illustrate on a recursion-free document the building process for an EXsum structure with 4-spoke ASPE nodes including child, descendant, parent, and ancestor axes, respectively.

The document is traversed in document order where, when visiting a node with element name *x*, all its axis relationships (as far as specified for EXsum) are recorded in ASPE(*x*). In that way, we derive step by step the exact number of occurrences per axis this element participates in.

[2] dplp has 41 element names where 32 are homonyms resulting in 146 nodes for the path synopsis. Hence, the avg. repetition of a homonym is more than 4. The numbers for element names, homonyms, and path synopsis nodes are (100, 6, 264) and (70, 12, 111) for *swissprot* and *nasa*, respectively. Because *nasa* has only a share of 6% homonyms, the estimation procedure should be particularly simple and accurate. In all cases, the data structure for the path synopsis remains very small.

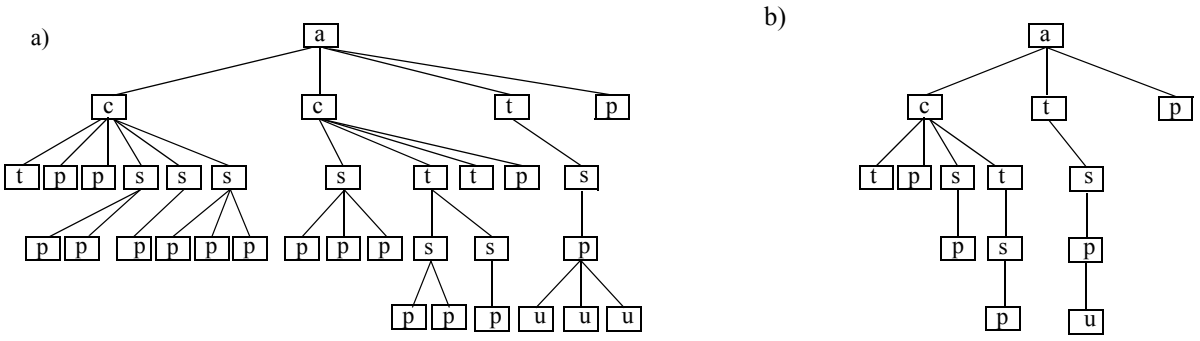

Figure 2. Sample XML document (fragment) and corresponding path synopsis

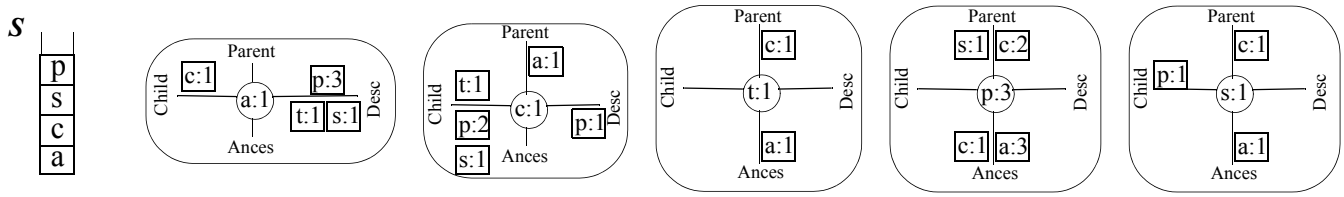

Figure 3. An intermediate EXsum state during the building process

Because we do not capture the order among the document nodes, we can locally process each node visited and summarize its axis relationships using a stack *S* which keeps, at any point in time, the path of the current node visited to the root. For example, when the root is visited, its element name *a* is pushed onto *S*. Furthermore, it causes the allocation of ASPE(*a*) and the recording of all axes information that can be evaluated at that time. In the next step, a node with element name *c* is located and pushed onto *S*. Because ASPE(*c*) is not present, it is created and the related axes information is added to *a* and *c*. In the subsequent step, a node with element name *t* is visited. Again, *t* is pushed onto *S*, ASPE(*t*) is created, and the axes information for *t* and its path elements *c* and *a* is completed.

Consider the general case, here described after having visited the seventh node in document order (leaf with element name *p*, see Figure 2a). Because *S* represents the path to the current node, we can —after *p* is pushed onto *S*—immediately update in ASPE(*p*) the axes information of p: *parent(s)*, *anc(c)*, *anc(a)*, and increment the related counters. Furthermore, using *S*, the axes information of the remaining path elements can also be maintained: in ASPE(*s*) for *child(p)*, in ASPE(*c*) for *desc(p)*, and in ASPE(*a*) for *desc(p)*. Note that information for the *child* (*parent*) axis is kept separately from that of the *descendant* (*ancestor*) axis to enable higher flexibility when axis-wise estimation of location steps is performed. In summary, when an element *x* is pushed onto *S*, the element frequency counter of ASPE(*x*) is incremented. Furthermore, if *S* contains a path of *n* elements, *n-1 child/desc* counters and *n-1 parent/anc* counters have to be maintained. When a leaf node of document *D* is encountered, the top-most entry (current element) of *S* is popped and the preorder traversal of *D* is continued with the next node not visited so far. The progress of the building process including stack *S* and the summary information for five ASPEs allocated after having visited the seventh node is shown in Figure 3.

When the document is fully traversed, EXsum building is complete, as shown in Figure 4. Note, when referring to descendant or ancestor axes in the estimation process, we need to consider the separated statistics for child and parent axes, too, because child $\subseteq$ descendant or parent $\subseteq$ ancestor always hold.

3. RECURSION-FREE PATHS

Because the kind of elements in the document (leading to unique-name or homonym-free documents or recursive paths) and, in turn, the number and type of location steps critically influence the quality of cardinality computations for path expressions, we have to make a suitable classification for the way cardinality can be determined.[3]

No matter what axis relationship is referred to in non-recursive document paths, the cardinality of all path expressions, which consist of a *single axis step* only, can be accurately determined. If *//x* is the first location step, then occ(*x*) directly delivers the cardinality of *//x*, i.e., the number of document nodes having element name *x*. The same information can be derived by accessing the ASPE of the root element and adding the values for *x* in the child and descendant spokes. Path expression */x* refers to the root element of a document. When accessing ASPE(*x*), we have to check whether or not its parent spoke is empty. If a parent is found, occ(*/x*) is necessarily 0, otherwise it must be 1. As an example evaluated on the document of Figure 2a, *//p* and */p* deliver cardinalities 17 and 0, respectively. The other types of relationships (axes) hardly make sense w.r.t. root and can, therefore, be neglected.

Let us first focus on documents using */* or *//* in the location steps. To estimate the cardinality of n-step path expressions (n > 2), we always have to know the kind of element names involved in the current location step. Therefore, the path synopsis has to be checked before the related ASPE node is inspected. Normally, we proceed step by step in a path expression until the last axis step (end step) is reached.

3.1 Unique Element Names

The simplest case occurs when the end step of an arbitrary long path expression refers to a unique element name *z*. No matter what axis references occur in the path expression, we immediately inspect ASPE(*z*) and—after having checked that the entire path expression matches with the path synopsis (some path instances are satisfying the path expression)—deliver occ(*z*) as the accurate cardinality information. For example, */a/t/s/p/u* or *//s/p/u* or *//t//p/u* can be evaluated in this way and all deliver by referring to Figure 4 cardinality 3. Note, the existence of unique element names—to be verified using the path synopsis—are most valuable for cardinality estimation. When referenced in some of the intermediate location steps, it can be used to begin the estimation "in the middle" starting with precise cardinality information. Assume some subtrees containing element name p are appended to the u nodes in the document of Figure 2a: then the estimation of *//t//s//u//p* would begin at ASPE(*u*) and return (for this example) accurate cardinality information.

3.2 Expressions with One or Two Steps

The construction principle of EXsum exactly covers two-step path expressions containing *child and descendant axes.* As an important property, the element-wise summarization, therefore, delivers accurate cardinalities for them, when the evaluation starts from the root

[3] In the XSeed approach [15], path queries are classified into simple path expressions (linear paths containing /-axes only), branching path expressions including branching predicates (but also limited to /-axes), and complex path expressions containing branching predicates and /- or //-axes. Because estimation complexity and accuracy is much more related to the kind of element names (unique, homonymous, recursive) and the length of the path expressions (number of node tests), we use an entirely different classification which is more oriented towards whether the expression cardinalities can be computed or only estimated (using interpolation heuristics).

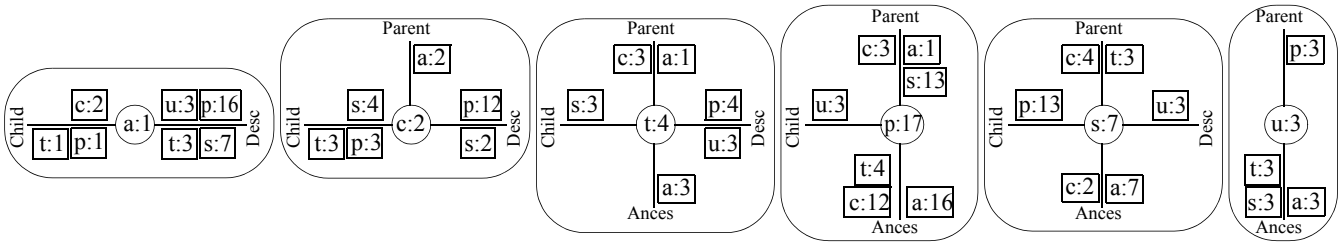

Figure 4. Complete EXsum structure for the sample document

or a unique element name. When evaluating two-step path expressions, we access the corresponding ASPE node of the first location step and follow the spoke referenced by the second location step to read the accurate path expression cardinality. Consider path expression *//c/p*. For the first step (*//c*), we directly access ASPE(*c*) and then follow the axis information for */p*. If the spoke child of node c with p exists (as in Figure 4), we deliver the exact cardinality 3. Thus, the optimizer can correctly estimate that there are 3 elements in the expression result. A little more complex is the computation of *//c//p,* because we have to include the matches for */p* in the second location step. In this case, the optimizer correctly estimates 15 elements in the expression result. Note, in all such cases, we access just a single ASPE node and inspect one or two spokes to return accurate cardinality information.

To summarize our discussion so far, EXsum delivers accurate cardinality results for all path expressions on homonym-free documents and for path expressions with one and two location steps on recursion-free document paths. We believe that these cases, where the EXsum structure reflects the document structure, cover the lion's share of all practically relevant estimation requests. For n-step path expressions (n>2), however, EXsum cannot always guarantee accurate estimation results. The structure of ASPE nodes does not capture the complete root-to-leaf paths of the document. Instead, it keeps axis relationships between pairs of element names and represents their distribution on the basis of element names. Therefore, we need a mechanism to approximate such expression cardinalities.

3.3 Cardinality Estimation of Multi-step Paths

When a unique element name is not encountered in the path expression, we proceed location step by location step from left to right. Consider a three-step path expression *//c/s/p* addressing the document in Figure 2a. For the first two location steps (*//c/s*), we follow the child spoke of ASPE(*c*) and find that *s* exists for this axis and has cardinality 4. To evaluate the subsequent location step (*/p*), we have to access ASPE(*s*) and follow the child spoke. Only if the value for *s* derived from *//c/s*, i.e., occ(*//c/s*), is equal to the value for *s* delivered by ASPE(*s*), we know that all *s* elements of the document are under *//c* and children of *c* elements. Hence, we can continue with accurate cardinality determination for *//c/s/p*. Applied to the document in Figure 2a, occ(*//c/s*) = 4 and ASPE(s) = 7, which means that three *s* elements are not reachable by the paths of *//c/s*. Obviously, EXsum cannot provide exact cardinalities in this case, because there may be *s* elements, not included in *//c/s*, but counted in ASPE(*s*) and, therefore, used for *s/p*. This problem always occurs when the path expression evaluation has to be continued via other ASPEs, that is, for an n-step path (n>2) n–2 times.

To cope with such situations, we decompose the path expression, e.g., *//x/y//z*, in overlapping two-location-step fractions and need a kind of heuristics, e.g., interpolation, to combine their results. To evaluate the partial expressions *//x/y* and *y//z,* we access ASPE(*x*) and ASPE(*y*) (whose values are equivalent to occ(*//x*) and occ(*//y*), respectively) and follow the ASPE axes for the second axis steps to obtain occ(*//x/y*) and occ(*//y//z*). Because not all *y* nodes of *//y//z* find a matching partner in the *y* nodes of *//x/y*, we assume uniform element distribution for the *z* nodes to enable a straightforward combination of estimates for such partial expressions. By using the ratio *C1/C2*, we linearly interpolate the number of occurrences of the subsequent step *y//z* to estimate occ(*//x/y//z*). For that, *C1* is given by occ(*//x/y*) and *C2*—equivalent to occ(*//y*)—is recorded as value of ASPE(*y*); thus, $C1 \leq C2$ always holds. This interpolation could be applied step by step, such that we gain an estimation heuristics for n-step path expressions. If more accurate information is present (e.g., by mining entire paths), it is used instead.

Estimating occ(*//c/s/p*) from the document in Figure 2a, we obtain by deriving *C1* = 4 and *C2* = 7 from EXsum as the interpolated cardinality *C1/C2**(occ(*s/p*)) = 4/7*(13), whereas the actual cardinality for the path expression *//c/s/p* is 9.

3.4 Estimating Parent and Ancestor Axes

Parent and ancestor axes are frequently evaluated in two situations. The first one occurs when using predicates. For example, when retrieving all nodes *p* under *s* having a node *t* as ancestor, the XPath expression *//s/p[./ancestor::t]* can be specified. In this case, a forward axis access in the main clause (*//s/p*) is accompanied by a reverse axis access for the evaluation of the predicate (*[./ancestor::t]*). Hence, for each forward instance of *//s/p* in the document, a backward navigation has to determine whether or not the forward instance evaluates to *True* for the given predicate. Other scenarios, where parent or ancestor axes are helpful, occur when IDREFs, used in the main clause, refer to other subtrees and predicates are needed to check for particular parents or ancestors.

The second and more general situation, in which parent and ancestor axes are evaluated, occurs when the user is not quite aware of the XML document structure, but he knows some relationships among nodes. For example, he knows that there is a parent relationship between nodes from *u* to *p.* Furthermore, he wants to retrieve all nodes *p* satisfying such a constraint, i.e., all nodes *p* which are parent of *u* nodes. However, he does not know the specific root-to-leaf paths in the document leading to *p* nodes as parents of *u*. An XPath expression solving this situation is *//u/parent::p*. On the other hand, if the user would know that all *p* parents of *u* nodes are reached by path *(a,t,s)*, he could directly locate them by */a/t/s/p[./u]*.

To show that EXsum is more expressive than its competitors while confronted with these complex situations, we discuss a heuristics how estimates can be derived for path expressions containing loca-

tion steps for ancestors and/or parents. While the EXsum construction principle counts the existing nodes when summarizing the child and descendant axis relationships, this does not hold for parent and ancestor axis relationships. Several (or all) of those relationships may refer to the same document node. Thus, our estimation algorithm needs more sophistication. Assume *//x/parent::y*. The parent spoke of ASPE(*x*) delivers the logical parent relationships from nodes *x* to *y* (*lpr(x–>y)*) which means that we potentially have (from the perspective of *x*) up to *lpr(x–>y)* parent nodes *y*. On the other hand, ASPE(*y*) gives us the number of existing nodes *y*. Therefore, we can infer MAX as the maximum number of parent nodes: If *lpr(x–>y)* > ASPE(*y*) then MAX = ASPE(*y*) else MAX = *lpr(x->y)*. Furthermore, the child spoke of ASPE(*y*) delivers the logical child relationships from *y* to *x* (*lcr(y–>x)*), which correspond to existing nodes *x*, together with those relationships of the remaining child nodes summarized as *lcr(y–>r)*. Using these factors and a kind of uniform distribution assumption, we can provide the heuristics: occ(*//x/parent::y*) = MAX/(*lcr(y–>r) + lcr(y–>x)*)) * *lcr(y–>x)*. A similar rationale can also be applied for ancestor axes, but, of course, with a greater error probability. In this way, we may also supply estimates for parent/ancestor axes in all path expressions with not more than two location steps on non-recursive documents. Hence, occ(*//p/parent::s*) delivers 7, which can be checked in Figure 4.

For n-step path expressions (n > 2), we again apply an interpolation heuristics similar to that derived in Section 3.3. For example, a path expression including both ancestor and parent axes is *//u/ancestor::s/parent::t*. Having a 3-step path expression, we have to combine the results coming from two estimates for *C1* = occ(*//u/ancestor::s)* and *C3* = occ(*s/parent::t*). The number of existing *s* nodes expressed by *C2* is used as the interpolation factor: *C1/C2 * C3*. For location step *//u/ancestor::s*, we yield *C1* = 9/16. With *C2* = occ(*s*) = 7 and C3 = occ(*s/parent::t)* = 3, our heuristics delivers 27/112 as the final estimate.

3.5 Estimating the Remaining Axes

Axes such as *preceding* (*sibling*) and *following* (*sibling*) are considered exotic and hardly appear in real-world applications. In general, their estimation is elusive, because these axes refer to relative positions of node instances. Hence, the data structures needed would explode when collecting statistics for them and would not be maintainable. Indeed, nobody has ever tried to give estimates for these axes. Nevertheless, we only want to point out here that EXsum carries some information which could be used and would be helpful, at least for the upper document levels. Because the root (*a*) is the first node in document order, counting all relationships in ASPE(a) delivers the number of following elements. When the expression */c/following-sibling::p* has to be estimated, we identify via ASPE(*c*) the parent of c and, in turn, figure out via the child spoke of ASPE(*a*) that there is no sibling *p*. Of course, we often need to apply some heuristics at lower levels. For example, expression */t/preceding-sibling::c* could be estimated by accessing the root ASPE(*a*) and finding in the child spoke that there is only a single t which has two c nodes as siblings. Because order information is not available, the number of c nodes in the role of *preceding/following sibling* has to be guessed.

4. DEALING WITH RECURSION

Documents such as *treebank* are considered exotic outliers, i.e., highly recursive documents are not frequent in practice and do not deserve first-class citizenship. However, some degree of recursion may be anticipated in (a small class of) documents. Thus, we have a look at recursiveness for reasons of generality and extend our method to support summarization on documents exhibiting a limited kind of structural recursion, too.[4] General recursion, however, seems to be elusive and does not allow for a meaningful estimation process which could deliver approximations of sufficient quality.

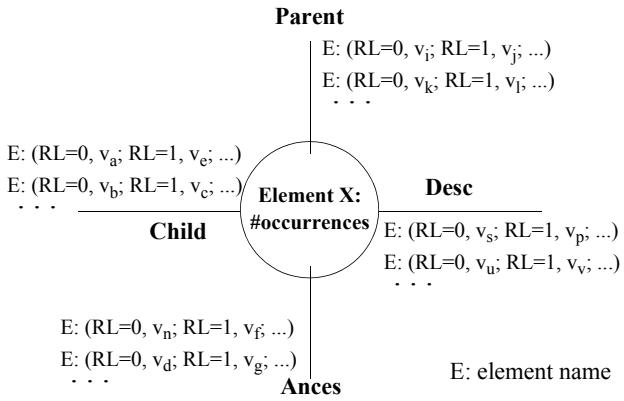

Figure 5. Recursion levels in EXsum

The concept of recursion level (RL) was introduced in [16] and explained for the case where only a single element name could recur in a path. Recursion levels were defined as: *Given a rooted path in the XML tree, the maximum number of occurrences of any label (element name) minus 1 is the path recursion level (PRL). The recursion level of a node in the XML tree is defined to be the PRL of the path from root to this node*. Thus, given a rooted path *(a,c,s,s,t)* of the document in Figure 6a, the RL of the first *s* node is 0 and that of the second *s* node is 1, whereas the PRL of this path is 1. RL was applied in XSeed to capture *parent-child* relationships in recursive paths.

4.1 Extending EXsum by Recursion Levels

We have extended the RL concept to capture for the nodes recurring in a path the *ancestor and descendant* relationships, too. To smoothly integrate RL information into the EXsum framework, we have slightly modified the ASPE node structure. For each spoke representing a structural distribution, we compute the recursion level of each distinct element in the spoke (see sketch in Figure 5). If recursion is not present in the document, the recursion level indicators could be dropped, resulting in the format shown in Figure 4. If recursion occurs in some paths, only the spokes of the elements affected need to carry the recursion level indicators.

Figure 6 is derived from Figure 2 to illustrate the differences needed for the application of recursion levels[5]. As in Section 3, we use stack *S* for separately capturing the axis relationships of the current node. Assume the current state of S = [*a, c, s, s, s, p*] where for the top element *p* the incremental changes to EXsum have to be found. To update EXsum with the axis relationships of *p*, we have to consider the recursion levels of the elements in *S*: *a*: RL=0; *c*: RL=0; *s*: RL=0; *s*:

[4] [15] states that recursive XML documents represent the most difficult cases for path query processing and cardinality estimation and that none of the existing approaches addressed this problem and, in particular, the effects of recursion over the accuracy of cardinality estimation. Claiming that recognizing and capturing recursion is a unique feature of the XSeed kernel, N. Zhang, however, only presents a limited solution.

[5] Only element names appearing in recursive paths could be represented in this extended format and empty recursion levels could be omitted. To facilitate reading, we have chosen a uniform structure for all element names.

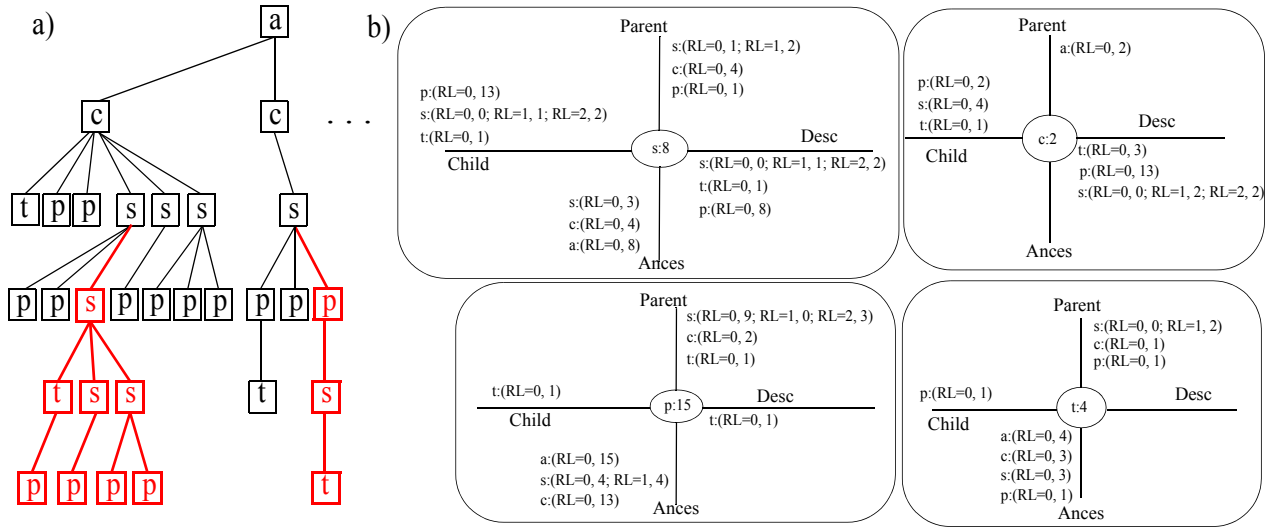

Figure 6. Extended XML document (a) and EXsum structure capturing recursion (cut-out) (b)

RL=1; *s*: RL=2. Moreover, from *p*'s point of view, (*s*: RL=2) is parent, whereas the other elements in *S* are ancestors. Hence, occ(*p*) of ASPE(*p*) is increased by 1. Then in ASPE(p), the *target elements* of *p*'s relationships—the parent axis for (*s*: RL=2) and the ancestor axis for (*s*: RL=1), (*s*: RL=0), (*c*: RL=0), and (*a*: RL=0)—are incremented by 1. Furthermore, the axis relationships of the remaining elements in *S* have to be maintained, because an element *p* at RL=0 (as the *target element* of these relationships) becomes a new child resp. descendant for them. Therefore in ASPE(*s*), the child axis is incremented for (*p*: RL=0). Furthermore, value 1 is added to (*p*: RL=0) and again to (*p*: RL=0) in the descendant axes of ASPE(*s*). In ASPE(*c*) and ASPE(*a*), the descendant axes are each incremented by 1 for (*p*: RL=0).

Because EXsum extended with recursion levels now captures axis distributions in a more fine-grained way, we can explore the RL property for path expression estimation. Note, however, our element-wise summarization is partially confused by recurring elements such that a kind of redundant counting (or overestimation) is provoked. For example, the four resp. three leaf elements *p* in Figure 6a are captured as descendants resp. children of *s* (for *s*:(RL=0) and *s*:(RL=1) resp. *s*:(RL=2)). In contrast, the recursion in path *a/c/s/p/s/t* does not cause any imprecision. Hence, this redundant counting is guided by the recursion pattern and, therefore, it is not easy to remove it by an adjusted EXsum building algorithm (and beyond the considerations in this paper). As a consequence, our approach—while delivering accurate results for the initial two location steps for recursion-free documents—will compute guaranteed results for recursive paths only for the first location step.

4.2 Cardinality Estimates by Recursion Levels

Capturing summary information for recursive documents is clumsy and more cumbersome. In addition, it implies some evaluation complexity. Here, we only consider the case where a single element recurs in a path, e.g., *s* in Figure 6. Our discussion is meant to sketch the problems rather than to present a solution.

Consider the estimation of *//c//s*. The cardinality is computed by following the child and descendant spokes in ASPE(*c*) and summing up the values over all recursion levels of *s*, yielding occ(*//c//s*) = 8. In contrast, *//c/s* would follow the child spoke of ASPE(*c*) and only aggregate the child relationships related to all recursion levels of *s*, resulting in occ(*//c/s*) = 4. Path expression *//s/p* yields an accurate answer, whereas *//s//p* delivers a substantial overestimation: occ(*//s/p*) = 13 and occ(*//s//p*) = 21, which could be corrected by measures depending on the recursion pattern.

Of course, interpolation has to be used for n > 2 steps. For example, to estimate a 3-step path expression *//c/s/p*, we yield occ(*//c/s*) = 4 (used as *C1*) for the first two steps. To continue the evaluation with *s/p*, ASPE(*s*) delivers 8 (used as *C2*) and its child spoke *p*:(RL=0, 13). Again, the only way to combine both cardinalities is to assume uniform distribution such that (*C1/C2*)*occ(*s/p*) = (4/8)*13 delivers the estimate.

4.3 Handling Recursive Path Expressions

Recursion can also occur in path expressions making the estimation even more difficult (and imprecise). Nevertheless, we have applied the RL concept to the estimation of recursive queries, too.

For recursive path expressions, we follow the definition in [15]: *A path expression is recursive with respect to an XML document if an element in the document could be matched by more than one node test in the expression.* Referring to this definition, it is easy to see that path expressions only consisting of /-axes (or parent axes) are not recursive. However, *//s//s* is a recursive path expression on the XML tree in Figure 6a, because a recursively occurring *s* node could be matched by both node tests at a node recursion level RL>0. Hence, recursive path expressions always involve at least one //-axis (or ancestor axis) and are usually applied to recursive documents. In some cases, recursive sub-expressions such as *//*//** or *//***/s* may also address non-recursive documents.

Consider the estimation of *//s//s*. The cardinality is estimated by following the child and descendant spokes in ASPE(*s*) and summing up the values over all recursion levels of *s*, yielding an overestimation of occ(*//s//s*) = 6. In contrast, occ(*//s/s*) = 3 delivers the accurate cardinality.

In the same way, cardinalities for n-step path expressions have to be approximated based on interpolation. Additionally, specific infor-

mation concerning the recursion levels has to be applied. As a final example, *//s/s/p* is estimated. Being a recursive 3-step path expression, the cardinality is derived by following the child spoke of ASPE(*s*) and summing the values over all recursion levels of *s*, yielding occ(*//s/s*) = 3. Furthermore, occ(*s/p*) delivers 13. With the interpolation factor of ASPE(*s*) = 8, the estimation is 3/8*13.

5. A LOOK AT PREDICATE ESTIMATION

The set of possible predicates in XQuery is so rich and complex that a single structure would not suffice to encompass all possibilities. Basically two kinds of predicates may appear in XQuery statements: value predicates and path predicates where both are represented in brackets ([]). The former has the traditional meaning inherited from relational databases in which techniques as histograms [8] and q-Grams [11] can be applied. The latter is a novel feature of XQuery and is on our focus. Of course, XQuery allows the coexistence of both kinds of predicates in a path expression.

Predicates (also called existential predicates) may contain one or more path expressions, e.g., */a/c[./s]/t* and *//s[./s and .//s/t]*. A path expression qualifies a path instance only, if the included predicate evaluates to *True*. If the predicate, in turn, contains several path expressions logically connected by AND, then all path expressions must be evaluated to *True*.

To estimate the cost of QEPs, we need to know the selectivity of predicates for which we use a specific calculation method: Path Related Selectivity (PRS) is defined by the following expression:

$$PRS(/v_1/\ldots/v_n[v_{n+1}]) = Card(/v_1/\ldots/v_n[v_{n+1}]) / Card(|/v_1/\ldots/v_n|)$$

In this formula, the function *Card* estimates the cardinality of a set of location steps according to the mechanism explained in Sections 3 and 4.2. PRS tries to estimate the selectivity of a predicate by dividing the estimated cardinality of the location steps in the predicate clause by the estimated cardinality of the location steps in main path expression, which is close to the XQuery/XPath definitions.

To better explain this method, consider the path expression *//s[.//s/t]* and Figure 6. For the main path expression (*//s*), the factor $Card(|/v_1/.../v_n|)$ is translated to a cardinality $Card(|//s|) = 8$. For the predicate estimation, we apply the techniques sketched in Section 4.2. The estimated cardinality of factor $Card(|/v_1.../v_n[v_{n+1}]|)$ results in $Card(|//s[.//s/t]|) = 6/8*1$. By using PRS, the estimated selectivity of this predicate is (6/8)/8 = 0.09.

6. DYNAMIC XML DOCUMENTS

After having considered non-recursive and recursive documents and the related estimation of path expressions, we want to mention that the statistical information needed has to be maintained in the case of dynamic documents. In contrast to competitor approaches [2, 16], EXsum therefore provides another strong asset. Its building mechanism can be easily extended to dynamic XML documents such that we can always guarantee up-to-date statistics. Because the derivation of axes information is independent of the document order and only relies on the path from the root to the current node to be modified, we may incrementally proceed with the building mechanism, as described in Section 2.2. Each node insertion separately increments the axis relationships it participates in the corresponding EXsum elements. For each node to be deleted, EXsum can be updated in a context-free way by just decrementing all axis relationships involved.

Table 1. Characteristics of documents considered

doc-name	description	size in MB	#nodes (inner /text)	#elem. names	max. depth	avg. depth	observation
dblp	Comp. Sc. index	330.0	9,070,558 / 8,345,289	41	7	3.39	middle size, less regular, non-recursive
nasa	Astron. data	25.8	532,967 / 359,993	70	9	6.08	small size, less regular, non-recursive
swiss-prot	Protein data	109.5	5,166,890 / 2,013,844	100	6	4.07	middle size, quite regular, non-recursive
tree-bank	Wall Street Journal	86.1	2,437,667 / 1,391,845	251	37	8.44	middle size, completely irregular, highly recursive

7. EXPERIMENTS

To determine the practical use of our proposal, we have performed a number empirical experiments using the well-known set of XML documents listed in Table 1 and whose results were analyzed regarding estimation time, sizing, and accuracy. To evaluate the estimation accuracy, we have built, for each document, a query workload and computed estimates of query results thereby contrasting actual and estimated values.

7.1 Documents and Query Workload

In Table 1, column *#nodes* represents the total number of nodes in a document according to the DOM specification [14]. Column *#elem.names* indicates that the number of distinct element/attribute names is very small compared to the total number of nodes. Hence, documents typically have a very repetitive structure. Columns *max.depth* and *avg.depth* give some hints on the variability of documents. For example, we can consider *swissprot* as quite a regular document, because its average depth is close to its maximum depth. In contrast, *treebank* is quite an irregular one. Within this spectrum, the other documents are less regular.

We have built a query workload as follows. For all documents except *treebank*, we have generated all possible queries encompassing only child axes. We have randomized the generation of queries whose path expressions contain the remaining axes, i.e., descendant, parent, and ancestor, and queries with predicates. For *treebank*, we have randomly generated queries having four axes and predicates. In total, we have experimented with approximately 3,000 queries.

The empirical exploration of our approach is based on an EXsum structure with four spokes (as illustrated in Figure 5) and compares the results against those achieved by XSeed [16] and LH [2] as the main competitors. XSeed uses a graph structure to capture statistics of XML document paths and necessarily accepts—because of pruning the graph search—false positive hits in the computation of estimation results. Because of its structure, XSeed is one of the most compact summaries proposed in the literature. LH is a tree-based summary which is enhanced by histograms to compress the element distributions at each level. By intensively evaluating it using End-biased histograms [8], we have shown that, in terms of estimation quality, LH certainly belongs to the superior methods known, i.e., it delivered the lowest estimation errors among all summaries compared [3].

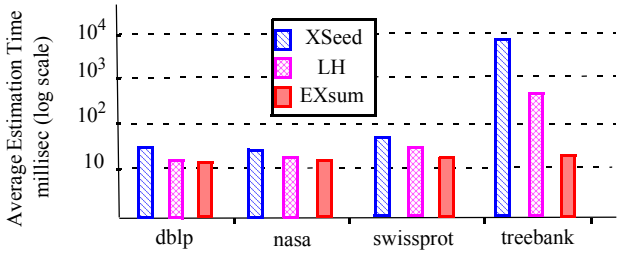

Figure 7. Estimation times

Because LH and XSeed cannot support the estimation of parent and ancestor axes, we have compared EXsum against them with queries encompassing only child and descendant steps as well as predicates having the same kinds of steps.

All empirical experiments were run on a platform consisting of a 2-GHz Pentium Centrino Duo processor with 1GB main memory, running under Windows XP SP2. For that purpose, we have implemented and integrated all methods to be compared into our native XDBMS XTC [7], which is written in Java 6. Throughout the performance measurements, we used 512 MB of main memory for the Java Virtual Machine and 16 MB for the XTC buffer.

7.2 Estimation Time Analysis

A particular advantage of EXsum is that, in the worst case, the number of ASPE nodes to be accessed is equal to the number of axis steps, even for recursive queries. Because we have implemented the ASPE nodes as hash tables, the time complexity for each access is in *O(1)*. As a consequence, our approach generally achieves low estimation times, as illustrated in Figure 7.

Contrary to EXsum, XSeed and LH only provide low estimation times for non-recursive documents, which are typically not greater than 75 msec. However, XSeed reaches up to 10 seconds for the estimation of cardinalities concerning queries which address the *treebank* document. For the same kind of estimation tasks, LH only requires on the average 300 msec.

EXsum, in contrast, consumes on average 25 msec for the estimation support of a query considering all documents (including *treebank*). Indeed, it required not more than 56 msec for cardinality estimation in the worst case. This demonstrates that EXsum scales well for any type of XML document and provides the minimum impact on the query optimization process.

7.3 Sizing Analysis

For the summaries compared, we require more than 3 – 5 orders of magnitude less storage space than for the full document, except for *treebank*, which reaches only 2 – 3 orders of magnitude. For example in almost all documents, the summary sizes vary between 0.001% and 0.2% of the document sizes (Figure 8b). Even for *treebank*, the summary structures only consume 0.1% and 0.2% of the document size using XSeed and EXsum, respectively. XSeed exhibits the least space consumption for highly recursive documents, thus outperforming EXsum and LH (0.8%) for such documents.

Figure 8 shows that all summaries are compact enough to enable memory-resident use, although they may consume hundreds of kilobytes. Regarding the main-memory buffer to keep statistical data re-

a)

doc-name	Ø bytes per ASPE node
dblp	198.78
nasa	163.69
swis-sprot	179.45
tree-bank	739.71

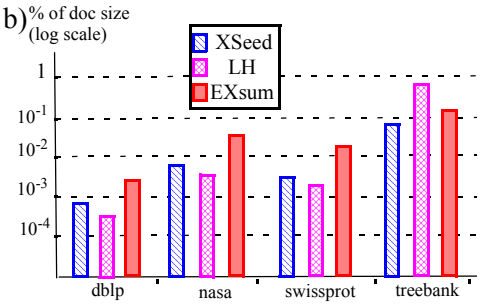

Figure 8. Storage consumption related to the document size

quired for the estimates supporting the query optimization process, EXsum exhibits the lowest memory footprint. While competitor structures have to be entirely kept in main memory, EXsum loads only the needed ASPE nodes. Consider *treebank* as a kind of bad case, for which EXsum requires 740 bytes on average per ASPE node (Figure 8a). As ASPE nodes, loaded on demand to memory, are bounded by the number of axis steps, the estimation of a path expression on *treebank* with ten axis steps (covering, possibly, more than half of the *treebank* depth) would consume less than 8KB. Hence, according to our observations, EXsum needs 10 to 100 times less memory than that required for XSeed (70KB) and LH (700KB) for the estimation process.

7.4 Accuracy Analysis

To measure the accuracy of the methods compared, we use the NRMSE (normalized root mean square error) metric which is defined by the formula:

$$\left(\sqrt{(\Sigma_{i=1}^{n}(e_i - a_i)^2)/n}\right) \div ((\Sigma_{i=1}^{n} a_i)/n),$$

where n is the number of queries in the query workload, e is the estimated result size and a is the actual result size. NRMSE measures the average error per unit of the accurate result size.

To analyze the estimation accuracy (see Figure 9), we have applied to each document a reference workload, involving different types of queries. With a smaller memory footprint than the competitors, EXsum reaches at least comparable and often more accurate estimation results. For example, EXsum has an estimation error close to zero for *swissprot* (quite a regular document). For *treebank*, it provides an estimation error with almost one order of magnitude less than XSeed. For *dblp* and *nasa*, EXsum is comparable to the others.

For queries with reference to parent and ancestor axes, we cannot cross-compare EXsum's estimation quality to that of the competi-

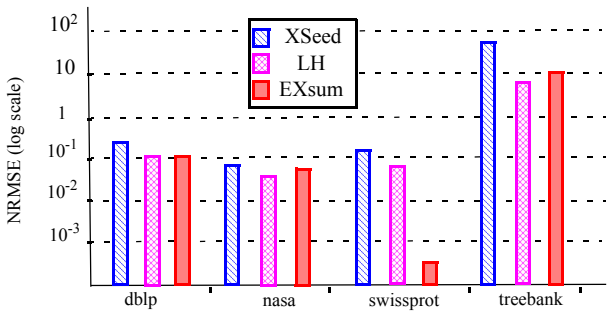

Figure 9. Accuracy analysis (reference workload)

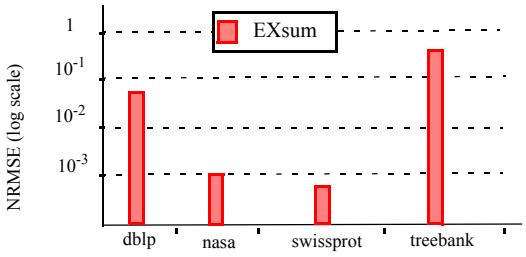

Figure 10. Accuracy obtained for parent/ancestor axes

tors. To give some indicative hints, we have evaluated the subset of queries having parent and ancestor axes in our workload (e.g., *//i/ancestor::sup/parent::title* on *dblp*) and plotted the estimation errors obtained for the four documents in Figure 10. Hence, EXsum achieves an impressive accuracy for these types of queries, too, and confirms its quality also for axes not supported by the competitors.

8. CONCLUSION

In this paper, we proposed a framework to summarize XML data called EXsum which can, due to its expressiveness, support the cardinality estimation of all important document axes. Note, Exsum delivers accurate cardinalities for the most common cases (i.e., unique element names in the end step of and one- and two-step path expressions). We have also extended EXsum to cope with recursive document paths based on the concept of recursion levels. In addition, the EXsum framework is extensible to incorporate other types of statistical information.

EXsum consumes slightly more storage space than the competitor approaches for non-recursive documents. However, the entire storage consumption reaches at most 0.2% of the document size, even for highly recursive documents. Moreover, as queries are evaluated by only loading the required ASPE nodes on demand, EXsum achieves the lowest memory footprint for evaluating queries, even when these queries refer to a large part of the document (e.g., descendant/ancestor axes). Providing fast access time, EXsum keeps the query optimization overhead low, even for the estimation of complex queries addressing recursive document paths.

Our experiments have shown that, regarding estimation quality, EXsum yields very accurate estimation results, both for non-recursive or highly recursive path expressions. For the latter, EXsum delivers an estimation error almost one order of magnitude less than that of XSeed (so far, the most compact summary) and comparable to that of LH (so far, the most accurate one). All these properties make EXsum at least comparable and often superior to its competitors.

Motivated by these results, we will extend our research in three directions. First, we want to explore the summarization of text values in XML documents (together with structural summarization). Second, we will further explore the estimation of the 'remaining' axes, such as following (sibling) and preceding (sibling), in differing contexts of path expressions. Last but not least, we plan experiments with EXsum in real cost-based XML query optimization scenarios to support QEP construction.

REFERENCES

[1] Aboulnaga, A., Alameldeen, A. R., and Naughton, J. F.: Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In Proc.VLDB Conf., 591-600 (2001)

[2] Aguiar Moraes Filho, J. and Härder, T.: Accurate Histogram-based XML Summarization. In Proc. ACM SAC, Vol. II, 998-1002 (2008)

[3] Aguiar Moraes Filho, J. and Härder, T.: Tailor-Made XML Synopses, in: Proc. 8th Int. Baltic Conf. on Databases and Information Systems, Tallinn, Estonia, 25-36 (2008)

[4] Freire, J., Haritsa, Jayant R., Ramanath, M., Roy, P., and Simeon, J.: StatiX: making XML count. In Proc. ACM SIGMOD Conf., 181-191 (2002)

[5] Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Proc. VLDB Conf., 436-445 (1997)

[6] Härder, T., Mathis, C., and Schmidt, K.: Comparison of Complete and Elementless Native Storage of XML Documents. In Proc. IDEAS Symp., 102-113 (2007)

[7] Haustein, M. P. and Härder, T.: An Efficient Infrastructure for Native Transactional XML Processing. Data & Knowledge Engineering 61:3, 500-523 (2007)

[8] Ioannidis, Y. and Poosala, V.: Histogram-based Solutions to Diverse Database Estimation Problems. IEEE Data Engineering Bulletin 18:3, 10-18 (1995)

[9] Lim, L., Wang, M., Padmanabahn, S., Vitter, Jeffrey S., and Parr, R.: XPathLearner: An On-Line Self Tuning Markov Histogram for XML Path Selectivity Estimation. In Proc. VLDB Conf., 442-453 (2002)

[10] Polyzotis, N. and Garofalakis, M.: Structure and Value Synopses for XML Data Graphs. In Proc. VLDB Conf., 466-477 (2002)

[11] Surajit C., Venkatesh G., and Gravano, L.: Selectivity estimation for string predicates: overcoming the underestimation problem. In Proc. ICDE Conf., 227-238 (2004)

[12] Wang, W., Jiang, H., Lu, H., and Yu, J. X.: Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In Proc.VLDB Conf., 240-251 (2004)

[13] XPATH XML Path Language 2.0. W3C Candidate Release (Nov. 2005)

[14] XQuery 1.0 and XPath 2.0 Data Model (XDM) W3C Recommendation (Jan. 2007)

[15] Zhang, N.: Query Processing and Optimization in Native XML Databases, Ph. D. thesis, Technical Report CS-2006-29, University of Waterloo (Aug. 2006)

[16] Zhang, N., Özsu, M. T., Aboulnaga, A., and Ilyas, I. F.: XSeed: Accurate and Fast Cardinality Estimation for XPath Queries. In Proc. ICDE Conf., 61-66 (2006)