Enhancing the Estimation Quality of Element-centered XML Summarization Methods

José de Aguiar Moraes Filho, Theo Härder, Caetano Sauer
University of Kaiserslautern, P.O. Box 3049, 67653, Kaiserslautern, Germany
{aguiar, haerder, csauer}@cs.uni-kl.de

Abstract

An XML summary should enable cardinality estimations of different kinds on an XML document to flexibly support query optimization for languages such as XPath or XQuery. In contrast to conventional methods which typically emulate the document structure and record path-oriented statistics for it, element-centered XML summarization methods collect statistical information for document nodes and their axes relationships and aggregate them separately for each distinct element/attribute name. It has already partially proven its superiority in quality, space consumption, and evaluation performance. Surprisingly, this kind of inversion seems to have more service capability than conventional approaches. It is not only confined to the cardinality estimation of child and descendant axes, but also allows to approximate parent and ancestor axes, too. Therefore, we refined and extended element-centered XML summarization methods to capture more statistical information and propose new estimation procedures. We tested our ideas on a set of documents with largely varying characteristics.[1]

Keywords: *XML summarization, EXsum, statistics, query optimization.*

1. Introduction

Path expression optimization (for XPath/XQuery) relies on the quality of statistical information about distribution of XML document nodes and their axis relationships, because the success of a query optimizer critically depends on estimates as precise as possible about selectivities of location steps and node cardinalities in subtrees. Although many publications [1, 2, 3, 5, 7, 8, 9, 11, 12] focus on such an estimation support, no commonly agreed-upon solution is available. Capturing a variety of structural relationships and statistical information, the methods widely vary in estimation accuracy delivered, storage space occupied, and memory footprint needed, which together may heavily influence the superordinate process of query optimization. We have observed that the result quality of existing methods is strongly influenced by the shapes of the underlying summary structures in which the document statistics are recorded.

Tree-based methods summarize a document by means of a tree (or tree-like) underlying structure trying to mirror the hierarchy of the document structure. Their simple construction, however, potentially consumes much storage space for deeply structured documents. To overcome this drawback, some compression (e.g., histograms) [3, 5] and pruning techniques are applied [1], which trade storage gain with degraded accuracy. However, these methods are difficult to balance on XML documents with increasing structural variability.

[1] This paper is an extended and revised version of [4]

Methods whose underlying structure is a graph [8, 9, 12], by virtue of this structure, tend to consume less storage than tree-based approaches. But, they have to cope with false positive hits, because path information for paths not present in the document can be derived from the summary. To avoid unacceptable estimation quality, the only way out is to prune the graph search, thereby trying to lower the false positive rate. While this approach may be feasible in some situations, the trade-off incurred has to be controlled by "manual" tuning parameters. Hence, correct parameter setting and, therefore, estimation quality is left to the DBA expertise.

Table-based methods, in turn, are completely dependent on the number of distinct path instances existing in the document, which enforces the use of pruning [7] and compression techniques [11] similar to those of tree-based methods. As most important limitation, they can only estimate path expressions with child (/) axes. Thus, differing from tree-based and graph-based approaches, they do not support expressions containing either predicates or descendant axes.

To overcome the drawbacks sketched so far, an element-centered method called EXsum has been developed [2]. The main idea is to focus on the set of distinct element/attribute names of an XML document, putting aside the strict hierarchy between them. Instead of summarizing over the entire tree structure at a time and to keep track of (root-to-leaf) paths in the document, this method independently gathers structural information for every distinct element name in the document tree. Here, we contribute to the state of the art by augmenting EXsum's capabilities:

- An extension of EXsum enables to capture more information from XML documents, especially the fan-in and fan-out of the axis relationships controlled.
- We elaborate the influence of recursion in XML path classes to the summarization quality.
- To enhance the estimation quality, we developed new heuristics especially path expressions involving more than two steps.
- By implementing and empirically evaluating our methods, we cross-compared them to competitor algorithms using a set of well-known and widely varying XML documents.

The remainder of this paper is organized as follows. We start by providing basic concepts (Section 2) and a look at the EXsum structure (Section 3). We explore ways to enrich EXsum and provide tailor-made estimation methods (Section 4) for these extensions, before we empirically investigate their suitability for varying document characteristics (Section 5). Section 6 concludes the paper.

2. Basic Concepts and Definitions

XML documents usually exhibit a high degree of redundancy in their structural part, i.e., they contain many paths having identical sequences of element/attribute names. To distinguish the summarization challenge in practical applications, the most important characteristics of a well-known collection of XML documents—widely used as a reference collection for comparative experiments in scientific publications—are listed in Table 1. Column *#nodes* shows the total number of nodes in the respective document according to the DOM specification, whereas column *#E/A names* indicates that, despite the huge amount of nodes, only a few distinct element/attribute names exist for them. Hence, documents typically

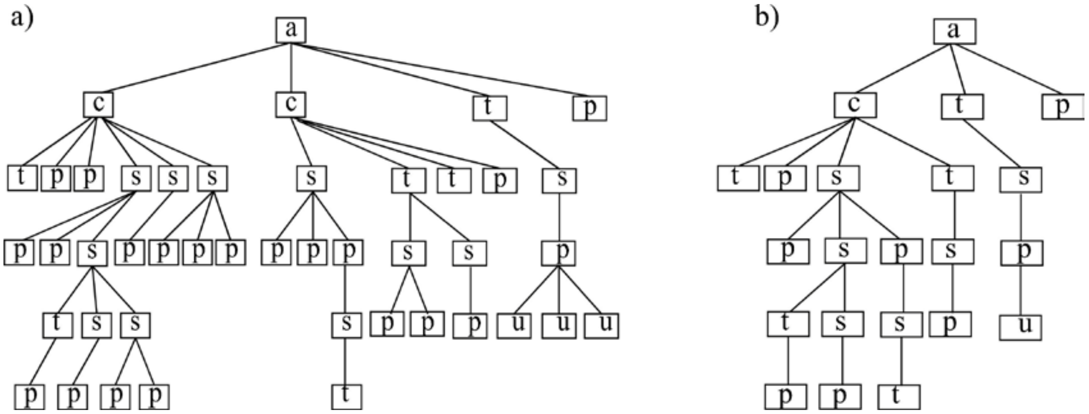

Figure 1. A sample XML document structure (a) and its path synopsis (b)

have a highly repetitive structure. Columns *max. depth* and *avg. depth* give some indicative hints on the variability of documents. For example, *swissprot* is considered to have quite a regular document structure, because its average depth is close to its maximal depth; in contrast, *treebank* has an extremely irregular tree structure. We consider these documents in our empirical experiments concerning XML summarization as detailed in Section 5.

Figure 1a depicts a sample document to be used as a running example, where many path instances only differ in the leaf values and the order they occur in the documents. Therefore, the structure part of them can be represented by a single unique path, called *path class* (see Figure 1b), to characterize all path instances having the same sequence of element names (but not their order) in the document. A *path synopsis* as a representation of all path classes serves as a query guide and a compact structural document view. To build such a path synopsis, all information needed is contained in a cyclic-free XML schema; otherwise, it can be constructed on the fly, while the document—sent by a client—is stored in the database. Typical path synopses have only a limited number of element names and path classes and can, therefore, be stored in a small memory-resident data structure. The way a path synopsis with its path classes represents the document tree is visualized in Figure 1b.

document name	description	size (MB)	#nodes inner/text	#E/A names	max.depth	avg.depth
dblp	Comp.Sc. Index	330.0	9,070,558 / 8,345,289	41	7	3.39
nasa	Astron. data	28.8	532,96 / 359,993	70	9	6.08
swissprot	Protein data	109.5	5,166,890 / 2,013,844	100	6	4.07
treebank	Wall Street J.	86.1	2,437,667 / 1,391,845	251	37	8.44
psd7003	Protein data	716	22,596,465 / 17,245,756	70	8	5.68

Table 1. Document characteristics

An element name occurring only once in the path synopsis is called *unique element name*, whereas element names appearing more than once in the path synopsis, but not in the same path class, are called *homonyms*. A path instance is said to be *recursive* when the same element name appears more than once in its path class.

In typical cases, documents contain varying degrees of homonyms, but most of its paths are recursion-free[2]. But in (rather) exceptional cases, we have to deal with *recursion* in a document, as exemplified by the paths *(a,c,s,s,s,p)* or *(a,c,s,p,s,t)* in Figure 1. Hence, some degree of recursion may be anticipated in specific document classes. Thus, we analyze recursiveness for reasons of generality and evaluate summarization structures that support documents exhibiting a limited[3] kind of structural recursion, too.

The concept of recursion level (RL) was introduced in [12] as a way to better represent recursion in XML documents and explained for the case where only a single element name could recur in a path. Given a rooted path in the XML tree, the maximal 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 the root to this node. Thus, given path *(a,c,s,s,t)*, the second *s* node has RL=1 and all other nodes have RL=0, whereas the PRL of this path is 1.

Recursion can also occur in query expressions, making the estimation even more difficult (and often more imprecise). For recursive path expressions, we follow the definition in [12]. 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.

3. A Brief look at EXsum

For comprehension, we repeat the main aspects of EXsum [2]—a method for element-centered XML summarization—,which was proposed in two versions: one targeting non-recursive documents and another for recursive ones, in which the RL information is included in the structure. The latter can be considered a more general structure subsuming the non-recursive version because, for a non-recursive document, all RL information can be present with RL=0. Therefore, we discuss the general form of EXsum that has been used as base for our extension.

An EXsum structure can be considered as a set of ASPE (axes summary per element) nodes where each of them, in turn, represents a distinct element/attribute name. An ASPE node is a compound of an element/attribute name, the number of related node occurrences (*occ*) in the document, and a varying number of "spokes"—a suitable ASPE node visualization resembles a "spoked wheel"—, each primarily representing the related element distributions for a specific axis. In this way, one gets not only a simple and flat summary structure, but also preserves estimation efficiency, since navigation costs are low and do not depend on document structure or size. Moreover, axis relationships are captured, enabling this class to effectively support path expression estimations. Note that information for the child (parent) axis is kept separately from that of the descendant (ancestor) axis to enable higher

[2] *dblp* has 41 element names where 32 are homonyms resulting in 146 path synopsis nodes. 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.

[3] Highly recursive XML documents such as *treebank* (see Table 1) are exotic outliers and not frequent in practice; therefore, they do not deserve first-class citizenship.

flexibility for axis-wise estimation of location steps. Figure 2 visualizes the statistics of 2 (out of 6) ASPE nodes collected on our sample document. The ASPE for *s*, e.g., shows that *occ(s)* is 11 and that all nodes *s* together have a child relationship to 16 nodes *p*, 1 node *s* at RL=1, 2 nodes *s* at RL=2, and 2 nodes *t*.

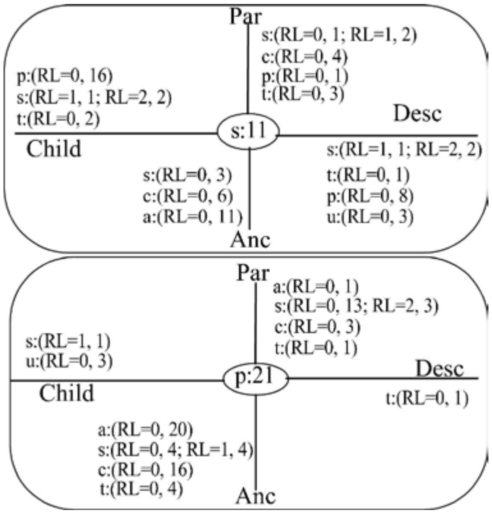

Figure 2. ASPE nodes

For path expressions with two location steps, the spoke referenced by the second step is followed in the ASPE node addressed by the first one. As an example, *//s/p* delivers 16. The construction principle of EXsum guarantees that such expressions on recursion-free documents always return accurate cardinalities. When recursion and two location steps are involved, EXsum can only compute approximate cardinalities, in general. Consider the estimation of *//s//s*, where the estimation procedure follows the child and descendant spokes in ASPE(s) and adds the values over all RLs of s, yielding an overestimation of *occ(//s//s)*=6, whereas *occ(//s//s)*=3 would deliver the accurate cardinality.

For n-step query expressions (n>2), EXsum has to rely on appropriate heuristics and, therefore, cannot always guarantee accurate results. ASPE nodes do not capture complete root-to-leaf paths of a document, instead they keep axis relationships between pairs of element names and record their distribution on the basis of element names. For this reason, n-step expressions, e.g., *//x/y//z*, are decomposed in overlapping two-step fractions and need a kind of interpolation to combine their results.

To evaluate the partial expressions *//x/y* and *y//p*, we access ASPE(*x*) and ASPE(*y*) (whose values are equivalent to *occ(//x)* and *occ(//y)*, respectively). 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. 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≤C2* always holds. This interpolation could be applied step by step, providing a heuristic for n-step path expressions. If more accurate information is present (e.g., by mining entire paths), it can be used instead. As an example, the cardinality of *//s/s/p* is estimated 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 16. With the interpolation factor of ASPE(*s*)=11, the estimation is 3/11*16.

As a first observation, EXsum captures, by means of the ASPE spokes, the axis-related fan-out for each element/RL combination using only a single counter. Computing fan-in and fan-out for every axis relationship may give us more opportunities to explore refined estimation methods. For this reason, we double the counters (see Figure 3): IC counters for fan-in and OC counters for fan-out.

4. Extending EXsum

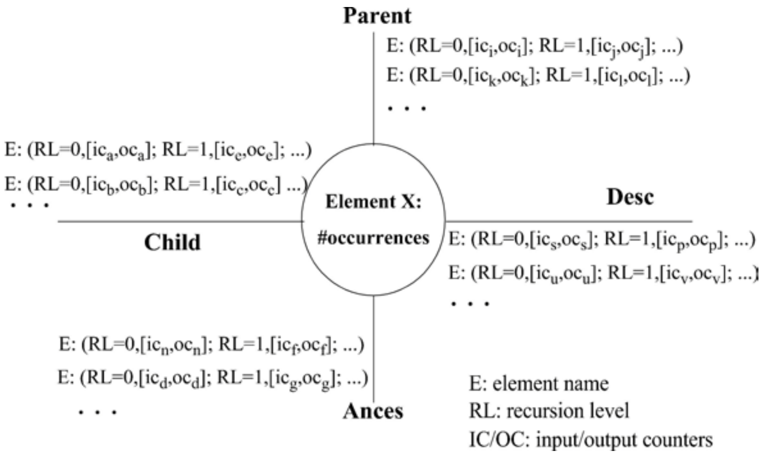
Parent
E: (RL=0,[ic_i,oc_i]; RL=1,[ic_j,oc_j]; ...)
E: (RL=0,[ic_k,oc_k]; RL=1,[ic_l,oc_l]; ...)
. . .
E: (RL=0,[ic_a,oc_a]; RL=1,[ic_e,oc_e]; ...)
E: (RL=0,[ic_b,oc_b]; RL=1,[ic_c,oc_c] ...)
. . .
Child
Element X:
#occurrences
Desc
E: (RL=0,[ic_s,oc_s]; RL=1,[ic_p,oc_p]; ...)
E: (RL=0,[ic_u,oc_u]; RL=1,[ic_v,oc_v]; ...)
. . .
E: (RL=0,[ic_n,oc_n]; RL=1,[ic_f,oc_f]; ...)
E: (RL=0,[ic_d,oc_d]; RL=1,[ic_g,oc_g]; ...)
. . .
Ances
E: element name
RL: recursion level
IC/OC: input/output counters

Figure 3. Extended EXsum node format

To illustrate these new counters, consider a *parent-child relationship* between elements *s* and *p* in recursion-free (i.e., RL=0) path instances in the document of the Figure 1. In the extended form of EXsum, these relationships are recorded as follows: ASPE(*s*) has a child spoke where a *p* exists having (RL=0,[IC=7, OC=13]). This means that, for the child relationship *s*→*p*, we find in the document 7 nodes *s* being parents of *p* nodes and 13 nodes *p* being children of *s* nodes. Conversely, in the parent spoke of ASPE(*p*), an entry for *s* exists with (RL=0,[IC=13, OC=7]) indicating that for the parent relationship *s*←*p* the same number of nodes is counted in the reverse direction. Note that IC and OC counters are somewhat replicated across ASPE nodes. However, this feature enables estimates of arbitrary long path expressions.

As a second observation, an additional information called DPC (Distinct Path Count) is helpful to support some special estimation procedures (see Section 4.2). DPC counts the number of distinct path instances that reach a specific relationship (e.g. *s*→*p*), starting from the document root. In other words, if we have a relationship *s*→*p*, we record the number of distinct rooted paths leading to *s* nodes involved in such a relationship. The triples *(RL=x,[IC,OC])* (see Figure 3) stored for each element in the child and descendant spokes are upgraded to a 4-tuple with *(RL=x,[IC,OC,DPC])*, encompassing DPC information.

4.1. Building Algorithm

To enable correct node counting by the extended EXsum, the plain building algorithm (described in [2]) must be modified as shown in Algorithm 1. As for the plain EXsum format, the counting of axes occurrences is done for each element using a stack *S*. Counter calculation is straightforward for forward axes (descendant and child), we simply add 1 to the respective counter in the corresponding ASPE node, every time we find a descendant/child element in the stack. Relationship counting in reverse axes (parent and ancestor) is, however, a bit complex. To correctly count element occurrences in reverse axes, we use an auxiliary list called *Element in Reverse Axis* (ERA). It maintains, for each element *x* in *S*, a list of all distinct nodes that were pushed onto *S* after *x* or, in other words, all distinct nodes under the subtree rooted by *x*. This means that every time an element is pushed onto *S*, the list of each element currently in *S* is updated. Another use of ERA lists is to update IC/OC counters in every ASPE node involved in the computation. We exemplify the extended EXsum building process using the document in Figure 1a and indicate in Algorithm 1 where each step is executed. Moreover, Figure 4 shows the initial building steps of EXsum.

When the document root is visited, its name *a* is pushed onto *S*. In addition, ASPE(*a*) is allocated and all axes information that can be evaluated in this situation is recorded. In this case, we add 1 in ASPE(*a*) as the current number of *a* occurrences in the document and

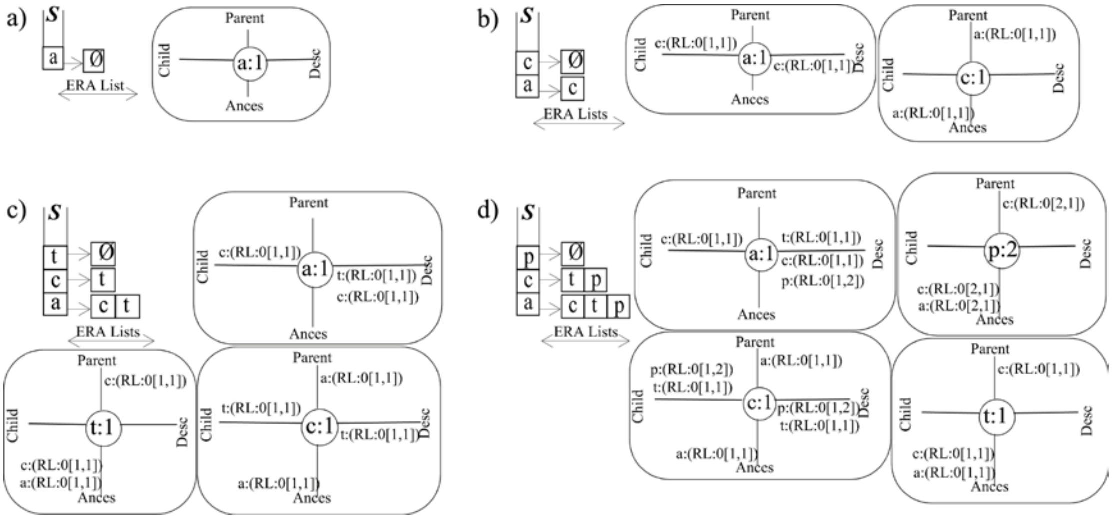

Figure 4. States of EXsum and stack *S* for the initial building steps

allocate an (empty) ERA list for *a*, currently the Top Of Stack (*TOS*) (lines 4 and 6-10 of Algorithm 1 and Figure 1a). In the next step—proceeding left-most depth-first, i.e., in document order—a node with element name *c* is located and pushed onto *S*. To control the allocation of ERA lists, the function *isFirstOccurrence(x,y)* checks if node *y* is the first occurrence under the subtree rooted by node *x*. To perform this check, we must look for node *y* in the ERA list of *x*. If no occurrence is found, we must register *y* and return *True*. Because ASPE(*c*) is not present, it is created and the related axes information is added to *a* and *c* as follows. The algorithm needs to adjust IC/OC counters in ASPE(*a*) and in ASPE(*c*). As it is the first time that an element *c* appears under (a subtree rooted by) *a*, the function *isFirstOccurrence* returns *True* and includes *c* in the ERA list of *a* (line 4). Thus, a *c* with (RL:0,[1,1]) is included in the child spoke of ASPE(*a*). Accordingly, ASPE(*c*) has an *a* with (RL:0,[1,1]) in the parent spoke, which indicates that there is only one *c* and one a *a* in this subtree. As the path *(a,c)* is recursion-free, procedure *Compute_RL* gives RL=0 (line 5).

Additionally, we add a *c* with (RL:0,[1,1]) in the descendant spoke of ASPE(*a*) and an *a* with (RL:0,[1,1]) in the ancestor spoke of ASPE(*c*) (Figure 4b). The main reason to do so is to be compliant with the axis definitions in the XPath specification [10][4]. Therefore, EXsum counts child (parent) and descendant (ancestor) relationships together in the descendant (ancestor) spoke and separately inserts child (parent) relationships only in the child (parent) spoke.

Continuing the document traversal, a node with element name *t* is now visited (*S=[a,c,t]*) (Figure 4c). 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. ERA lists of *a* and *c* now include a *t* and, again, both lists report that it is the first *t* encountered. Thus, an *a* with (RL:0,[1,1]) appears in the ancestor spoke of ASPE(*t*). The same *t*-counters exist for the child spoke of ASPE(*c*), parent and ancestor spokes of ASPE(*t*) (lines 6 and 9), and for the descendant spoke of ASPE(*a*) (line 5). As *t* has no children, an *End_Element* event is signaled and *t* is popped out from *S*.

[4] Here, a descendant (resp. ancestor) axis relationship is defined as *"as the transitive closure of the child (parent) axis; it contains the descendants of the context node (the children (parents), the children (parents) of the children (parents), and so on)"*

Algorithm 1: A new path to be added to EXsum structure is processed

```
1  Procedure $Build\_Synopsis$ begin
2      n ← stack.size();
3      for $i = 1$ to $n - 1$ do
4          setOppositeCount ← $isFirstOccurrence(i, n)$; /* maintain ERA lists */
5          $Compute\_RL$;
6          add descendant spoke from $stack[i]$ to $stack[n]$ with recursion level $descRecLevel$;
7          add ancestor spoke from $stack[n]$ to $stack[i]$ with recursion level $ancRecLevel$;
8          if $i = n - 1$ then
9              add child spoke from $stack[i]$ to $stack[n]$ with recursion level $descRecLevel$;
10             add parent spoke from $stack[n]$ to $stack[i]$ with recursion level $ancRecLevel$;
11     $computeDPC(stack)$/* count DPC */
12 end
13 Procedure $Compute\_RL$ begin
14     ancRecLevel ← 0;
15     for $j = 1$ to $i - 1$ do
16         if $stack[j] = stack[i]$ then
17             ancRecLevel++;
18     descRecLevel ← 0;
19     for $j = i$ to $n - 1$ do
20         if $stack[j] = stack[n]$ then
21             descRecLevel++;
22 end
23 Procedure $computeDPC(stack)$ begin
24     n ← stack.size();
25     if $n > 2$ then
26         for $i = 2$ to $n - 1$ do
27             $recLevel$ ← 0;
28             for $i = 1$ to $i - 1$ do
29                 if $j = i$ then
30                     $recLevel$ ++
31             add the sub-path $stack[1 .. i - 1]$ with RL=$recLevel$ to the descendant set of
               $(stack[i]; stack[n])$;
32             if $i = n - 1$ then
33                 add the sub-path $stack[1 .. i - 1]$ with RL=$recLevel$ to the child set of
                   $(stack[i]; stack[n])$;
34 end
```

Then, reaching the fourth element *p*, *S* and the counters are adjusted in the parent and ancestor spokes of ASPE(*p*), child and parent spokes of ASPE(*c*), and descendant spoke of ASPE(*a*). The states of EXsum and stack *S* up to this point are illustrated in Figure 4d.

The correct counting of elements in the reverse axes is highlighted when the process visits the fifth element (the second *p*, *S=[a,c,p]*). Here, ASPE(*p*) is already allocated and we have *p* in the ERA lists of *c* and *a*. Thus, it is not the first occurrence of *p* under the subtrees rooted by *c* and *a* (line 4, *setOppositeCount=false*). Therefore, we add 1 for ASPE(*p*) that now counts 2 and add also 1 for *p* in the child spoke of ASPE(*c*) and in the descendant spoke of ASPE(*a*) which now contain (RL:0,[1,2]). As *setOppositeCount=false*, we do not add 1 for the OC counters of *a* and *c* in parent and ancestor spokes of ASPE(*p*), i.e, they keep (RL:0,[2,1]). This mirrors the document structure in which there is one *c* as parent of two *p* nodes and, consequently, one *a* as ancestor of two *p* nodes. Hence, after a subtree is entirely traversed, we have obtained the correct values of the corresponding IC/OC counters.

4.1.1. Dealing with Recursion: The calculation of RLs is performed in the building algorithm (line 5 and procedure *Compute_RL* in Algorithm 1). For each axis relationship in every ASPE spoke, we calculate RL and, for each RL, the IC/OC counters. EXsum is, in its general format (see Figure 3), designed as a recursion-aware summary. For this purpose, we consider two kinds of recursion: forward-path recursion and reverse-path recursion (see right hand of Figure 5).

Forward-path recursion is considered when navigating downwards through the path, from the document root element to current element. Represented by child and descendant spokes in ASPE nodes, the cardinality information captured is used to support the estimation of such axes in recursive path expressions. In turn, the reverse-path recursion is computed in the opposite direction, i.e., from the current element to the document root. Similarly, reverse-path recursion is exploited in path expressions dealing with parent and ancestor axes.

The first appearance of a recursive path in the document of Figure 1 occurs when the document scan reaches the tenth element *t* with the path *(a,c,s,s,t)*. For this path, we have inserted t:RL=1,[1,1] in the child and descendant spokes of ASPE(*s*). Counters a:RL=1,[1,1] and c:RL=1,[1,1] are also added in the ancestor spoke of ASPE(*s*). We detail the building of ASPE nodes with the RL calculation using the twelfth element *s*, whose path is *(a,c,s,s,s)* and whose state of *S* has *s* as *TOS*, as depicted in Figure 5.

Consider that the document scan has reached the twelfth element, where the incremental changes to EXsum have to be found for the *TOS* element *s*. First of all, we must add 1 to ASPE(*s*), because a new node *s* is processed. Thus, ASPE(*s*)=3 is obtained. To update EXsum with the axis relationships of the *TOS s*, we have to consider the RLs in forward-path direction for the elements in *s*: a:(RL=0); c:(RL=0); s:(RL=0), which have *TOS* as a descendant; and s:(RL=1), for which *TOS* is a child, and also the RLs in the reverse path for the elements in *s*: a:(RL=2); c:(RL=2); s:(RL=1), which are ancestors of *TOS*; and s:(RL=0), which is the parent of *TOS*.

When calculating forward-path recursion, the first elements in *s* (*a*, *c*, and *s*) follow the same building process as in the recursion-free case, because there is no recursion in path *(a,c,s)*. Thus, up to the first *s* in the stack, ASPE nodes and their spokes are built as explained in the previous section. Recursion comes into play with the second *s* in stack *S*.

Summarization of forward-path recursions runs as follows. We add 1 to the OC counter of *s* with RL=0 in the descendant spoke of ASPE(*a*). In the same way, 1 must be added to the OC counter of *s* with RL=0 in the descendant spoke of ASPE(*c*) and ASPE(*s*). Furthermore, we must insert a new RL record in the descendant and child spokes of ASPE(*s*) for *s* with (RL=1,[1,1]).

For reverse-path recursions, we add two new RL records to the ancestor spoke of ASPE(*s*) for *a* and *c* with (RL=2,[1,1]). Additionally, we add a new RL record in the ancestor spoke of ASPE(*s*) for *s* with (RL=1,[1,1]). Finally, we increment both IC and OC counters of *s* with RL=0 in the parent and ancestor spokes of ASPE(*s*).

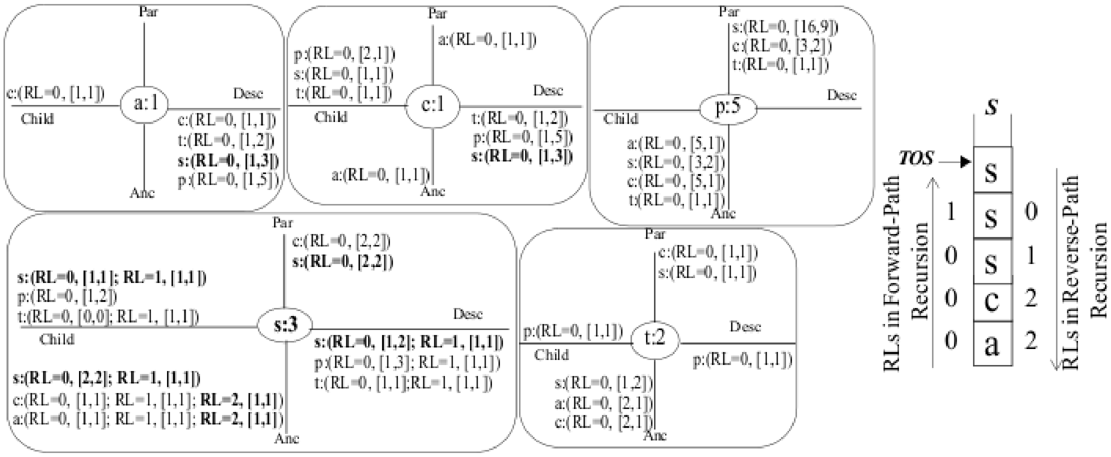

Figure 5. EXsum state after processing the 12th element

4.1.2. Calculating DPC: To compute the DPC, we need to maintain the set of all distinct rooted paths for each relationship. We can implement this in two ways. First, given a relationship *s→p*, we can traverse the path synopsis seeking for the specific rooted paths we need and repeat this traversal for each relationship computed. If the path synopsis exists, this traversal has a time complexity of *O(n.log(n))*, where *n* is the number of nodes of the path synopsis.

In the absence of a path synopsis for the document, we have designed the self-contained procedure *computeDPC* (lines 23-34 and called in line 11 of Algorithm 1). It processes every rooted path occurrence, which is represented by the *Stack* parameter provided by a call of the *buildSynopsis* procedure. For every given pair of related nodes, we maintain two sets, one for the child (child set) and the other for the descendant (descendant set) relationship.

To explain how this algorithm works, we take a practical example. Consider the path *(a,c,s,s,t)* in the recursive document in Figure 1. The *TOS* element in this case is *t*. The procedure starts by assigning the size of the path to *n*, which is 4 in this case. Then, we check for the value of *n*. The procedure is only executed for values of *n* greater than 2 (line 25), because a path with 2 nodes contains only one child relationship and, therefore, no preceding distinct paths. Then, for every node *i* in the path before the *TOS*, we add an occurrence of the sub-path that leads from root to the descendant relationship between *i* and the *TOS* (line 31). The proper RL value is calculated by counting the occurrences of *i* in this sub-path (lines 26-30). For the particular case of the relationship between *TOS* and the element right before it, the path is also added to the child set (line 33). So, in the given example, the procedure starts with element *c* (position 2). Then, we take the descendant set of the relationship from *c* to *t*, denoted as a pair *(c;t)*, and add an occurrence of the path */a* with RL=0. The child set will be left untouched, as *c* is not at position *TOS*-1. Because sets are

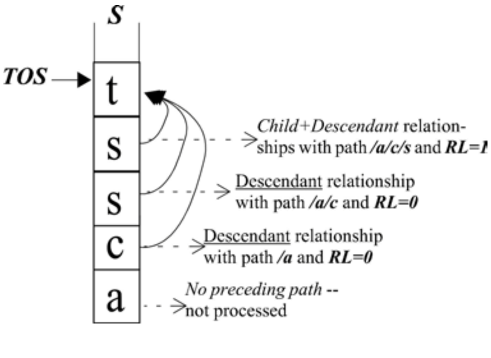
S
TOS
t
s
s
c
a
Child+Descendant relation-ships with path /a/c/s and RL=1
Descendant relationship with path /a/c and RL=0
Descendant relationship with path /a and RL=0
No preceding path -- not processed

Figure 6. Computing DPC

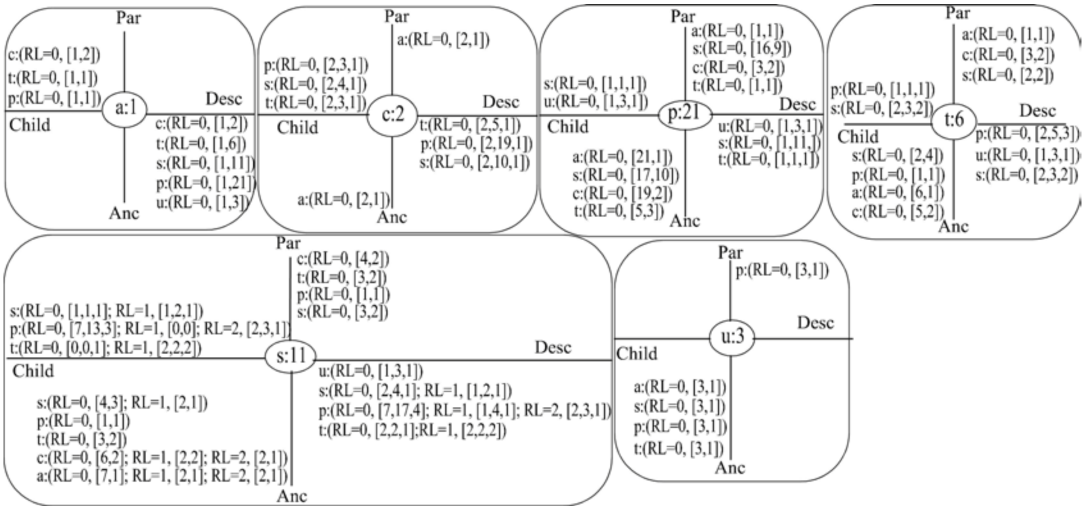

Figure 7. EXsum for our sample document

used, no duplicate elements will be added, and only distinct paths will populate them. When going to the first *s* element, the path to be added is */a/c*, also with RL=0. Reaching the second *s*, we need to add the path */a/c/s* to the relationship *(s;t)*. This time, because one occurrence of *s* is found in the preceding path, it will be inserted with RL=1. Moreover, it will also be added to the child set, as the element is positioned right before the *TOS t*. Figure 6 illustrates which relationships and paths are computed. After completion of the document scan, the EXsum building is finished; the resulting structure, including DPC counters, is shown in Figure 7.

4.2. Heuristics to Support Estimation of Longer Path Expressions

Because exact calculations cannot be performed by EXsum when n-step (n>2) path expressions come into play, [2] proposed the use of *interpolation* as an estimation procedure. Here, we introduce new heuristics to compensate the decrease of accuracy in these cases. For the following explanations, we refer to the EXsum structure depicted in Figure 7 and, without loss of generality, exemplify the heuristics with recursion-free queries.

4.2.1. DPC Division: The DPC (Distinct Path Count) Division procedure relies on the uniform distribution assumption of document paths leading to a location *step* captured by the DPC counter in the EXsum structure. The idea is to divide the *occ(step)* cardinality by the related count.

Consider a path expression */a/c/s/p*. To estimate step */s*, *occ(/s)* is given as follows. In ASPE(*c*), we search the child spoke for an *s* and find the OC and DPC counters. The estimation of *occ(/s)=OCcounter / DPCcounter* delivers 4/1=4 as step estimation. This means that (coincidently) there is only one path leading to *c→s*. For the next step (*/p*), we find three distinct paths reaching *s→p*: *(a,c)*, *(a,t)* and *(a,c,t)*. With an OC counter value 13 of *p* in the child spoke of ASPE(*s*), *occ(/p)=13/3=4.3*, which is the estimated cardinality of the expression.

DPC is also available for descendant steps. DPC for the step *s//p* would deliver 4 (*(a,c)*, *(a,c,s,s)*, *(a,t)* and *(a,c,t)*), because it corresponds to the number of paths leading to *s* nodes which have at least one *p* in its subtree. Note that, for the same pair relationship x→y, the DPC counter in the descendant spoke is always greater or equal than that in the child spoke, because, regarding EXsum, child steps are a subset of descendant steps. Thus, the estimate for *//s//p* is *occ(//s//p)=17/4*

4.2.2. IC Counter Division: This procedure makes an estimation for each location step by dividing the OC counter by the IC counter found in ASPE spokes. It assumes that the occurrences of a given axis relationship are uniformly distributed throughout each distinct path. This method becomes equivalent to the DPC Division method if each distinct path has only one occurrence in the document tree.

Thus, to estimate the expression *//c/s/p*, we have, for *occ(/p)*, OC=17 and IC=7. The procedure estimates *occ(//c/s/p)=17/7*. IC counter division method is intended to bring accurate results for nodes that appear occasionally and in isolated positions, like some nodes in *treebank*. As an example, the estimate of *//c/t/s* results in *occ(//c/t/s)*=3/2=1.5 and the actual result is 2.

4.2.3. Node Frequency Division: Another heuristics for the path step estimation is to divide the value of the OC counter by *occ(x)* of the context node *x*. This is similar to the IC counter division method, except that it considers, for a step *a/b*, being *a* the context node, all occurrences of *a* in the document, i.e., *occ(a)*, without considering any relationship to *b* nodes.

The accuracy of this method should be, in the best case, equal to the one achieved by the IC Counter Division. By applying this procedure on the expression */a/c//t*, we have *occ(c//t)*=5 and ASPE(*c*)=2. The estimation gives us *occ(c//t)* / ASPE(*c*)=2.5

4.2.4. Previous Step Cardinality Division: This method uses two factors: the *occ(currentstep)* gathered from the OC counter in ASPE spoke and the estimation result of the previous step in an expression. Dividing both numbers, the procedure yields the estimation for the current step. By iterating this calculation throughout all location steps of a path expression, the estimation of the expression is calculated.

This method introduces a strict dependency between the estimations of each step, forcing a sequential execution, which could be a disadvantage for certain document paths. On the other hand, it only depends on the OC counter. For example, for estimating *//t/s/p*, we take three location steps *//t*, */s* and */p*. The first one yields *occ(t)*=6. For the second step, we probe the child spoke of ASPE(*t*) for an *s* and take its OC value, i.e., 3. Then, *occ(/s)*=6/3=2. For the last step */p*, we take the OC value of *p* in the child spoke of ASPE(*s*), i.e., 13. Thus, *occ(/p)*=13/2=6.5. Hence, *occ(//t/s/p)*=6.5.

5. Empirical Evaluation

To assess the practical value of the EXsum extensions, it is necessary to systematically evaluate and cross-compare them against competing methods under representative empirical workloads and in an identical environment. For this reason, we have implemented and incorporated our ideas and competing summaries in our native XML database management system called XTC [2]. As competitor approaches, we have chosen XSeed and LWES, whose parameter settings were adjusted as follows. For the XSeed kernel, we have set the search pruning parameter to 100 for *treebank*, 50 for *dblp*, and 20 for the other documents. For LWES, EB histograms were continuously applied to all levels of the summary structure.

For each document in Table 1, we have generated query workloads containing three basic query types: queries with *simplechild* and *descendant* path steps and those with predicates. Descendant queries may have multiple descendant steps. In the case of EXsum and LWES, *parent* and *ancestor* queries are also evaluated, as they provide support for them. We have cross-compared the approaches regarding timing, sizing, and estimation quality.

The test workloads were processed on a computer equipped with an Intel Core 2 Duo processor chip running at 2.2 GHz and 3 GB of DDR-2 RAM memory, the GNU/Linux operating system (version 2.6.27), and the Java 6 virtual machine (version 10) of Sun. The XTC server process was running on the same machine.

5.1. Timing Analysis

Estimation time refers to the time needed to deliver the cardinality estimations for a query addressing a given document, that is, the time the estimation process needs to get the query expression, to access the summary (possibly more than once), and to report the estimate to the optimizer. Here, we report averages of the times needed for the queries in a workload.

Table 2 shows the estimation times classified by query types. For EXsum, the timing difference among the various estimation procedures is negligible. Thus, we have reported in Table 2 just the worst results depicted in column *EXsum*. Obviously, EXsum delivers superior results for all document and query types; hence, its impact on the overall optimization process is very low. While LWES is comparable and XSeed slightly slower for most queries, both of them consume prohibitive times in deeply structured documents, especially for the estimation of descendant axes; this seems to be unacceptable for practical use.

Table 2. Estimation times (in msec)

Document	EXsum	LWES	XSeed
Simple child queries			
dblp	2.85	3.18	13.21
nasa	3.55	3.30	11.60
swissprot	2.93	2.80	17.83
treebank	3.72	5.15	7,413.00
psd7003	3.86	3.15	3.28
Descendant queries			
dblp	3.18	3.12	26.12
nasa	2.75	2.93	7.19
swissprot	2.95	3.20	20.00
treebank	3.21	27,391.00	8,588.00
psd7003	4.04	3.53	7.96

Document	EXsum	LWES
Parent and ancestor queries		
dblp	4.39	7.00
nasa	4.42	4.50
swissprot	5.48	7.34
treebank	5.09	10.88
psd7003	4.00	3.34
Queries with predicates		
dblp	4.92	7.63
nasa	5.60	10.20
swissprot	11.80	24.84
treebank	7.29	6,705.20
psd7003	13.86	15.75

5.2. Sizing Analysis

The storage amount listed in Table 3 characterizes the net size of a summary and only includes the bytes necessary to store the summary on disk. The gross size may be influenced by a specific implementation and confuse a direct comparison. XSeed presents the most compact storage. LWES is more compact than EXsum for non-recursive documents.

Table 3 also compares the memory footprint for various estimation situations on all summaries/documents. We have computed the average memory size needed to estimate cardinalities for queries with two characteristics: queries whose number of location steps, whatever axes included, are equal to the document's average depth (rounded up to next integer value), and queries whose number of location steps is equal to the maximum document depth. These cases enable us to infer whether a summary needs to be entirely or only partially loaded into memory, i.e., whether or not the memory consumption of a summary is bounded to the number of location steps in a query during the estimation. Except for EXsum, all other summaries require the entire structure in memory to perform cardinality estimations. EXsum, in contrast, only loads the referenced ASPE nodes and is, therefore, the summary with lowest memory footprint and related disk IO. Thus, although the use of EXsum implies higher storage space consumption, the estimation process may compensate it by lower memory use and IO overhead.

Table 3. Sizing Analysis

Document	EXsum DPC	EXsum other	LWES	XSeed
Storage (in KB)				
dblp	7	6	2	7
nasa	9	9	2	7
swissprot	14	13	4	15
treebank	168	158	3,339	160
psd7003	7	7	2	6
Memory Footprint (in KB)				
# location steps = ceil(average depth)				
dblp	0.65	0.62	2	7
nasa	0.91	0.84	2	7
swissprot	0.68	0.65	4	15
treebank	6.03	5.66	3,339	160
psd7003	0.60	0.57	2	6
# location steps = maximal depth				
dblp	1.13	1.08	2	7
nasa	1.17	1.11	2	7
swissprot	0.82	0.78	4	15
treebank	24.80	23.28	3,339	160
psd7003	0.79	0.76	2	6

5.3. Estimation Quality

To compare the estimation quality of EXsum and competitors, we have used an error metric called Normalized Root Mean Square Error (NRMSE). NRMSE is given by the formula $\sqrt{\sum_{i=1}^{n}(e_i - a_i)^2}/(\sum_{i=1}^{n}(a_i)/n)$ where *n* is the number of queries in the workload, *e* the estimated result size, and *a* the actual result size. NRMSE measures the average error per unit of the accurate result. Furthermore, we analyze timing including estimation and build times, and sizing (i.e., storage size and memory footprint) needed for cardinality estimation of query expressions.

In addition to the results presented, we also compared our estimation methods against “Interpolation” and “No Estim.”. The former is the original EXsum estimation method proposed, the latter is the simple probing of the corresponding ASPE nodes with related spokes, but without using estimation procedures. “No Estim.” gives us hints on how good EXsum represents structural properties of XML documents. We analyzed the accuracy of simple-child and descendant queries in Table 4.

Table 4. Comb. NRMSE error for child/desc. queries (%)

Doc.	No Estim.	Input Cnt	Prev.Step	Node Freq.	DPC	Interpol.	LWES	XSeed
dblp	13.89	244.36	6.06	244.36	13.86	0.91	14.49	0.91
nasa	32.98	228.65	291.49	228.67	29.32	3.35	3.45	3.36
swissprot	0.00	267.07	202.55	267.07	0.00	0.00	12.10	0.01
treebank	866.51	587.43	>1,000	587.02	591.64	429.67	361.25	441.68
psd7003	0.00	133.35	0.00	133.35	0.00	0.00	0.00	0.00

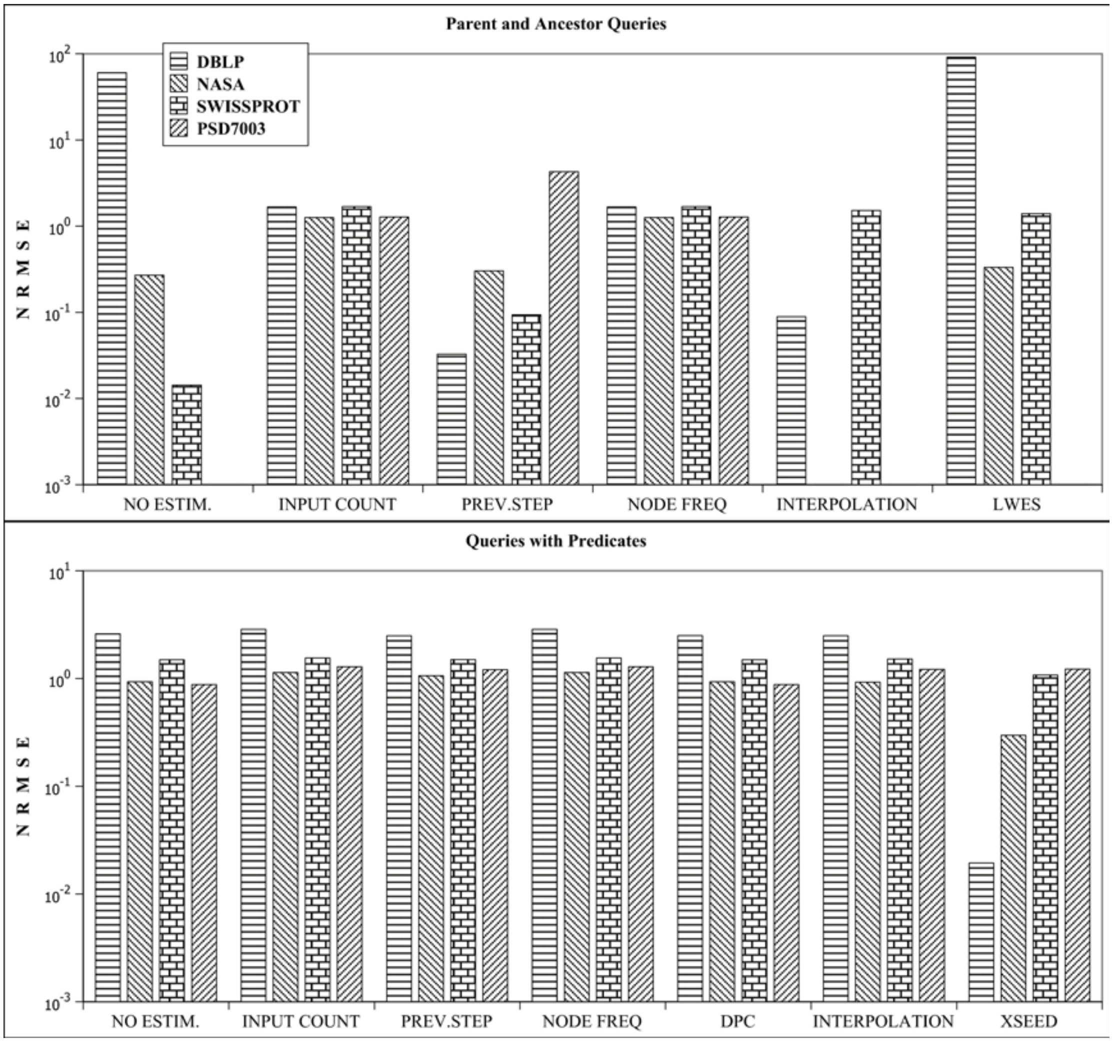

Figure 8. Parent/ancestor and predicate queries

We can see that EXsum itself delivers good estimations, except in cases where a high degree of recursion is present (*treebank*) or where subtrees (or nodes) of homonyms are scattered across the document. In general, however, "DPC" and "Interpolation" present the best results in the majority of the cases, and "Input Count", "Prev. Step", and "Node Freq." yield the worst results, with the exception that "Prev. Step" can provide high quality estimations on documents whose degree of structural variability is very low (*dblp* and *psd7003*).

Furthermore, we have investigated the estimation quality for queries with parent and ancestor axes and queries with predicates (see Figure 8). For the former, "Prev. Step" and "Interpolation" delivered high-quality estimations in most of cases, comparable or even better than LWES. Except for *dblp*, "No Estim." also produced good estimations. Queries with predicates have obtained low estimation quality (an NRMSE reaching 100%). In this case, XSeed has a tendency to yield slightly better results in the most of the cases and especially good ones for *dblp*. A particular study on estimation quality of the approaches on *treebank* is

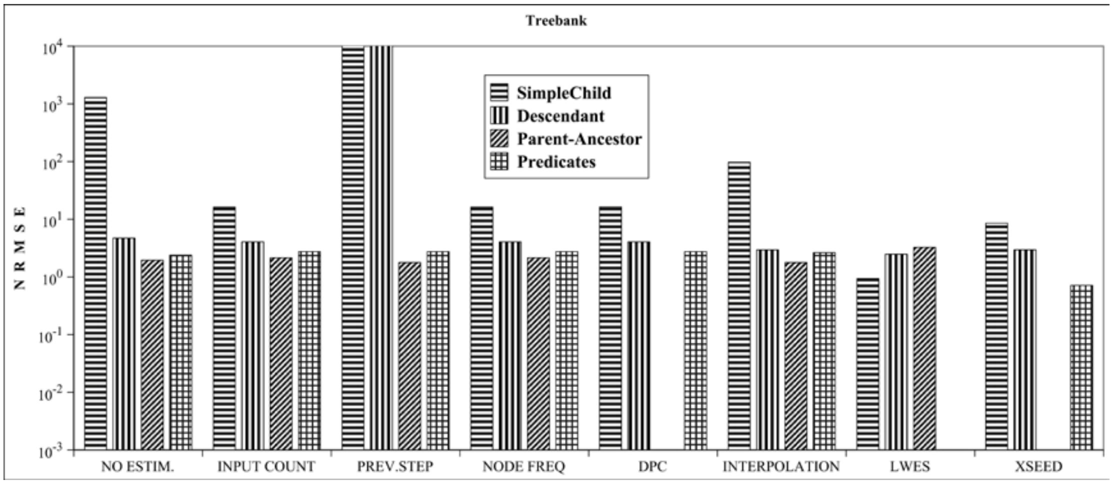

Figure 9. Estimation quality on *treebank*

given in Figure 9. These results tell us that the summarization of highly recursive documents remains a challenge for all compared summaries.

6. Conclusion

In this paper, we have extended EXsum, an element-centered summary, to capture more statistical information on XML documents and, using this information, support more estimation procedures than originally proposed. We have made experiments to quantitatively evaluate our proposal against approaches published in the literature. Two properties of XML documents directly influence the quality of summaries and, consequently, their estimation results: recursion and homonym trees (or nodes) which frequently exhibit varying numbers of occurrences in different parts of a document. While recursion gives room for improvements, because all approaches compared have presented very low estimation quality, homonym trees seem to be properly handled in the approaches compared. However, the consequences of a varying degree of interplay between recursion and homonyms are set to be a future research.

Evaluating the four new estimation methods proposed for EXsum, two of them have produced quality estimation results, whereas the remaining methods have reported high errors for query cardinality estimation. Nevertheless, the structure of EXsum itself yields quality estimations and, in some cases, accurate (NRMSE=0) ones, meaning that EXsum can capture the most important characteristics of a document.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton, “Estimating the selectivity of XML path expressions for internet scale applications”, In Proc. VLDB Conference, 2001, pp. 591-600.

[2] J. d. Aguiar Moraes Filho and T. Härder, “EXsum—an XML summarization framework”, In Proc. IDEAS Symposium, 2008, pp. 139-148.

[3] J. d. Aguiar Moraes Filho and T. Härder, “Tailor-made XML synopses”, In Proc. BalticDB&IS Conference, 2008, pp. 25-36.

[4] J. d. Aguiar Moraes Filho, T. Härder, and C. Sauer, “Enhanced statistics for element-centered XML summaries", In Proc. Database Theory and Application (DTA), LNCS, Jeju Island, Korea, 2009.

[5] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon, “Statix: making XML count”, In SIGMOD Conference, 2002, pp. 181-191.

[6] T. Härder, C. Mathis, and K. Schmidt, "Comparison of complete and elementless native storage of XML documents", In IDEAS Symposium. IEEE Computer Society, 2007, pp. 102-113.

[7] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr, "Xpathlearner: An on-line selftuning markov histogram for XML path selectivity estimation", In Proc. VLDB Conference, 2002, pp. 442-453.

[8] N. Polyzotis and M. N. Garofalakis, "Structure and value synopses for XML data graphs", In Proc. VLDB Conference, 2002, pp. 466-477.

[9] N. Polyzotis and M. N. Garofalakis, "Xsketch synopses for XML data graphs", ACM Trans. Database Syst., 31(3):1014-1063, 2006.

[10] W3C, XML path language (XPath) 2.0|W3C recommendation 23 january 2007. http://www.w3.org/TR/xpath20/, 2007.

[11] W. Wang, H. Jiang, H. Lu, and J. X. Yu, "Bloom histogram: Path selectivity estimation for XML data with updates", In Proc. VLDB Conference, 2004, pp. 240-251.

[12] N. Zhang, M. T. Özsu, A. Aboulnaga, and I. F. Ilyas, "XSeed: Accurate and fast cardinality estimation for XPath queries", In Proc. ICDE Conference, 2006, pp. 61.

Authors

José de Aguiar Moraes Filho is a PhD candidate at the University of Kaiserslautern, Germany. He has received his Master degree in Applied Computer Science at the University of Fortaleza (UNIFOR), Brazil, in 2004. His research interests include XML query processing and optimization, mobile databases, and data integration.

Theo Härder obtained his Ph.D. degree in Computer Science from the TU Darmstadt in 1975. In 1976, he spent a post-doctoral year at the IBM Research Lab in San Jose and joined the project System R. In 1978, he was associate professor for Computer Science at the TU Darmstadt. As a full professor, he is leading the research group DBIS at the TU Kaiserslautern since 1980. He is the recipient of the Konrad Zuse Medal (2001) and the Alwin Walther Medal (2004) and obtained the Honorary Doctoral Degree from the Computer Science Dept. of the University of Oldenburg in 2002. Theo Härder's research interests are in many areas of database and information systems in particular, relational DBMS architecture, XML databases, transaction systems, and information integration. He is author/coauthor of 7 textbooks and more than 200 scientific contributions with > 140 peer-reviewed conference papers and > 60 journal publications.

His professional services include numerous positions as chairman of the special interest group "Databases and Information Systems" of the German Informatics Society, conference/program chairs and program committee member, editor-in-chief of Computer Science—Research and Development (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM). He served as a DFG (German Research Foundation) expert and was chairman of the Center for Computed-based Engineering Systems at the University of Kaiserslautern, member of two joint collaborative DFG research projects DFG (SFB 124, SFB 501), and co-coordinator of the National DFG Research Program "Object Bases for Experts".

Caetano Sauer has received his Bachelor in Computer Science from the University of Kaiserslautern and is currently a Master candidate at the same institution. He currently works as a Research Assistant for the XTC project. Research interests include Database Systems, XML Query Processing, Web Services, and Natural Language Processing.