Enhanced Statistics for Element-Centered XML Summaries

José de Aguiar Moraes Filho, Theo Härder, and Caetano Sauer

University of Kaiserslautern
P.O. Box 3049, D-67653, Kaiserslautern, Germany
{aguiar,haerder,csauer}@cs.uni-kl.de

Abstract. Element-centered XML summaries collect statistical information for document nodes and their axes relationships and aggregate them separately for each distinct element/attribute name. They have already partially proven their superiority in quality, space consumption, and evaluation performance. This kind of inversion seems to have more service capability than conventional approaches. Therefore, we refined and extended element-centered XML summaries to capture more statistical information and propose new estimation methods. We tested our ideas on a set of documents with largely varying characteristics.

1 Introduction

In recent years, many publications [1,2,3,4,5,6,7] proposed summarization methods providing statistics needed for XQuery/XPath optimization. Supporting only small and incomplete subsets of the rich XPath relationships, these methods often fail to cover all estimation requests (for up to 8 major axes relationships) of an optimizer. Furthermore, they widely differ in estimation accuracy delivered, storage space occupied, and memory footprint needed. Altogether, these deficiencies may heavily influence the process of query optimization. Therefore, no commonly agreed-upon solution is available so far.

We have developed an element-centered summary called EXsum [8] that focuses 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 gathers node by node structural information for every distinct element name in the document tree. Compared to competitor approaches, this novel way to summarize XML documents is more expressive and outperforms them in many optimization scenarios [8].

The base version of EXsum can be made extensible to enhance statistical information to be recorded on an XML document. Therefore, we have developed a suitable extension of EXsum which contributes to the state of the art. This extension captures more information from XML documents, especially the fan-in and fan-out of the axis relationships controlled, enabling new estimation procedures, especially in case of path expressions involving more than two steps or expressions with predicates. An empirical cross-comparison with competitor algorithms is provided based on a set of well-known XML documents.

2 Extending EXsum

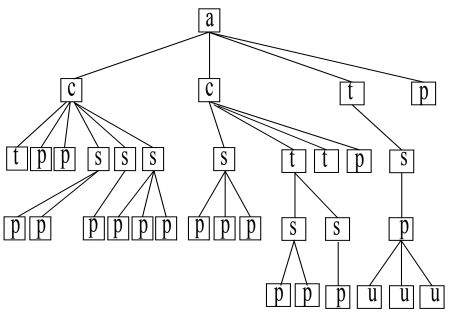

Fig. 1. A sample XML document

The original EXsum only captures the axis-related fan-out of all document nodes in a specific format ($[OC]$) by using a single counter per axis for each distinct element name. This axis-specific summarization of statistical data is called an ASPE spoke—*Axes Summary Per Element*. 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 number of counters: IC counters for the fan-in and OC counters for the fan-out (($[IC, OC]$)).

As a second observation, additional information called DPC (*Distinct Path Count*) is helpful to support some special estimation procedures. DPC counts the number of distinct path instances that satisfy a specific relationship, starting from the document root. In other words, if we have a relationship $s \rightarrow p$, we record the number of distinct rooted paths leading to s nodes involved in such a relationship. The tuples ($[IC, OC]$) stored for each element in the child and descendant spokes is upgraded to ($[IC, OC, DPC]$) to encompass DPC count.

Building Algorithm. To correctly count the node occurrences for EXsum, the plain building algorithm described in [8] must be modified. 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 more complex. 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 distinct element name 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 Fig. 1a. Furthermore, Fig. 2 illustrates 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 allocate an (empty) ERA list for a, currently the Top Of Stack (*TOS*) (Fig. 2a). In the next step, proceeding in document order, an element name c is located and pushed onto S. To control the allocation of ERA lists, we check if node y is the first occurrence under the subtree rooted

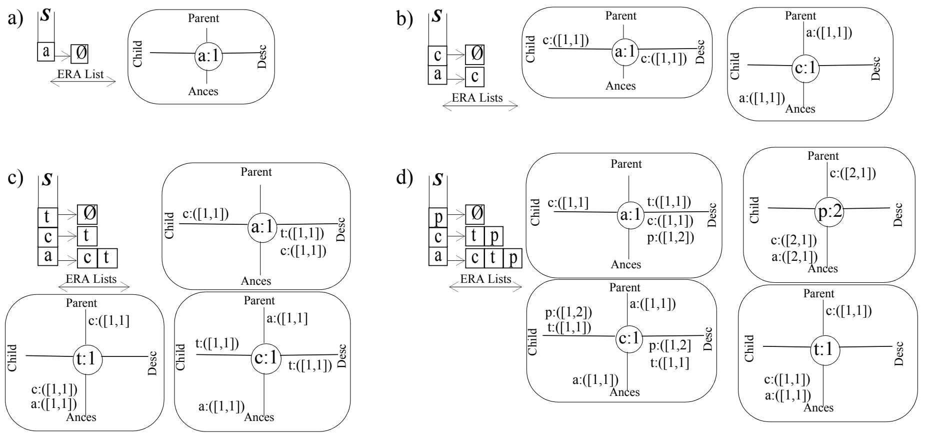

Fig. 2. States of EXsum and stack S for the initial building steps

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 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 IC/OC counters are adjusted 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 check returns $True$ and we include c in the ERA list of a. Thus, a c with ([1,1]) is included in the child spoke of ASPE(a). Accordingly, ASPE(c) has an a with ([1,1]) in the parent spoke indicating only one c and one a a in this subtree.

Additionally, we add a c with ([1,1]) in the descendant spoke of ASPE(a) and an a with ([1,1]) in the ancestor spoke of ASPE(c) (Fig. 2b). The main reason to do so is to be compliant with the axis definitions in the XPath specification. Hence, 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 ($S = [a, c, t]$) (Fig. 2c). 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 ([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), and for the descendant spoke of ASPE(a). As t has no children, t is popped out from S. 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 at this point are shown in Fig. 2d.

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. 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 ([1,2]). Conversely, 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 ([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.

Calculating DPC. To compute DPC, we need to maintain the set of all distinct rooted paths for each relationship. We have designed a procedure which processes every rooted path occurrence. 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.

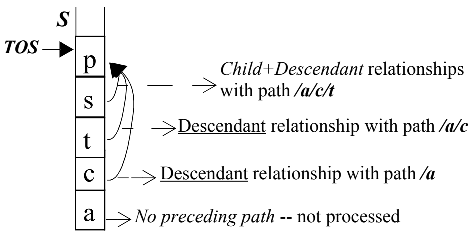
S
TOS
p
s
t
c
a
Child+Descendant relationships with path /a/c/t
Descendant relationship with path /a/c
Descendant relationship with path /a
No preceding path -- not processed

Fig. 3. Computing DPC

To explain how this it works, we take a practical example. Consider the path (a, c, t, s, p) in the document in Fig. 1. Fig. 3 illustrates which relationships and paths are computed. The TOS element in this case is p. 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, 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. For the particular case of the relationship between TOS and the element right before it, the path is also added to the child set. 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 p, denoted as a pair $(c; p)$, and add an occurrence of the path $/a$. The child set will be left untouched, as c is not at position TOS-1. Because sets are used, no duplicate elements will be added, and only distinct paths will populate them.

When going to the t element, the path to be added is $/a/c$. Reaching the fourth s, we need to add the path $/a/c/t$ to the relationship $(s; p)$. Moreover, it will also be added to the child set, as the element is positioned right before the TOS p. After completion of the document scan, EXsum building is finished; the resulting structure, including DPC counters, is shown in Fig. 4.

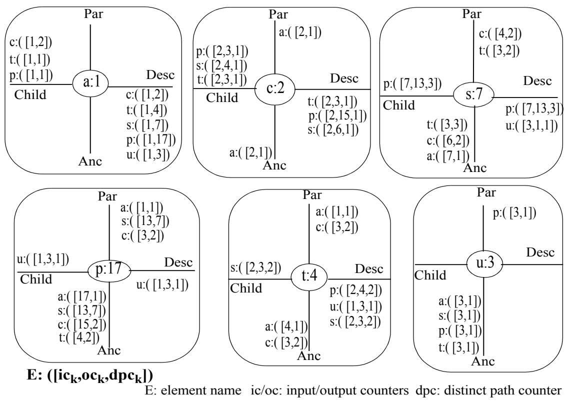

Fig. 4. EXsum for our sample document

Estimation Procedures. Because exact calculations cannot be performed by EXsum when n-step (n>2) path expressions come into play, [8] proposed the use of *interpolation* as an estimation procedure. Based on EXsum extensibility, we realize new procedures to compensate the decrease of accuracy in these cases. For the following explanations, we refer to the EXsum structure depicted in Fig. 4.

DPC Division. The DPC 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 there is only one path leading to $c \rightarrow s$. For the next step ($/p$), we find three distinct paths reaching $s \rightarrow p$: (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 $//s//p$ would deliver 3 ((a, c), (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. Thus, the estimate for $//s//p$ is $occ(//s//p) = 7/3$.

Previous Step Cardinality Division. The *Prev.Step* method uses the *occ*(*currentstep*) gathered from the OC counter 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 expression estimate is calculated.

This method introduces a strict dependency between the estimations of each step. For example, for estimating $//t/s/p$, we take three location steps $//t$, $/s$ and $/p$. The first one yields $occ(t) = 4$. For the second step, we probe the child spoke of ASPE(t) for an s and take its OC value, i.e., 3. Then, $occ(/s) = 4/3 = 1.5$. For the last step $/p$, we take the OC value of p in the child spoke of ASPE(s), i.e., 13. Thus, $occ(/p) = 13/1.5$ which is the estimated cardinality of the expression.

3 Empirical Evaluation

To assess the practical value of the EXsum extensions, we have systematically implemented and incorporated our ideas and competing summaries in our native XML database management system called XTC [9]. As competitor approaches, we have chosen XSeed [7] and LWES [2], whose parameter settings were adjusted as follows. For the XSeed kernel, we have set the search pruning parameter to 50 for the documents. For LWES, end-biased histograms were continuously applied to all levels of the summary structure.

We have used the following documents: *dblp* (330MB), *nasa* (25.8MB), *swissprot* (109.5MB) and *psd*7003 (716MB). For each document, we have generated query workloads containing three basic query types: queries with *simplechild* and *descendant* path steps and those with predicates. In the case of EXsum and LWES, *parent* and *ancestor* queries are also evaluated, as they provide support for them. The workloads were processed on a computer equipped with an Intel Core 2 Duo processor running at 2.2 GHz and 3 GB of RAM memory, the Linux operating system, and Java 6. The XTC server process was running on the same machine.

To accurately compare the estimation quality of EXsum and its competitors in all our experiments, we have used an error metric called Normalized Root Mean Square Error (NRMSE) 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 estimation times and sizing, i.e., storage size and memory footprint needed for cardinality estimation of query expressions.

Table 1. Estimation times (in msec)

Simple child queries			
Doc.	EXsum	LWES	XSeed
dblp	2.85	3.18	13.21
nasa	3.55	3.30	11.60
swissprot	2.93	2.80	17.83
pds7003	3.86	3.15	3.28

Parent and ancestor queries		
Doc.	EXsum	LWES
dblp	4.39	7.00
nasa	4.42	4.50
swissprot	5.48	7.34
pds7003	4.00	3.34

Descendant queries			
Doc.	EXsum	LWES	XSeed
dblp	3.18	3.12	26.12
nasa	2.75	2.93	7.19
swissprot	2.95	3.20	20.00
pds7003	4.04	3.53	7.96

Queries with predicates		
Doc.	EXsum	XSeed
dblp	4.92	7.63
nasa	5.60	10.20
swissprot	11.80	24.84
pds7003	13.86	15.75

Timing Analysis. Estimation time refers to the time needed to deliver the cardinality estimations for a query addressing a given document, i.e., 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 1 shows the estimation times classified by query types. As the timing differences among the EXsum's estimation procedures is negligible, we have reported in Table 1 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 is slightly slower for most queries, both of them consume prohibitive times for the estimation of queries with descendant axes. This makes us to infer that XSeed might provide unacceptable times if deeply structured documents come into play.

Table 2. Sizing Analysis

Storage (in KB)				
	EXsum			
Doc.	DPC	other	LWES	XSeed
dblp	7	6	2	7
nasa	9	9	2	7
swissprot	14	13	4	15
pds7003	7	7	2	6

Memory Footprint (in KB)				
Doc.	DPC	other	LWES	XSeed
# 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
pds7003	0.60	0.57	2	6
# location steps = maximum depth				
dblp	1.13	1.08	2	7
nasa	1.17	1.11	2	7
swissprot	0.82	0.78	4	15
pds7003	0.79	0.76	2	6

Sizing Analysis. The storage amount listed in Table 2 characterizes the size of a summary. LWES presents the most compact storage. XSeed is slightly more compact than EXsum for the compared documents.

Table 2 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 by the number of location steps in a query during the estimation. Except for EXsum, all other methods 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 the 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. Child/Desc. Error(%)

Doc.	Prev.Step	DPC	LWES	XSeed
dblp	6.06	13.86	14.49	15.51
nasa	291.49	29.32	3.45	3.36
swissprot	202.55	0.00	12.10	10.01
pds7003	0.00	0.00	0.00	0.00

Estimation Quality. In addition to the results presented, we also compared the estimation quality of our methods against LWES and XSeed. The comparison with the original interpolation method for EXsum has gained similar results; we have omitted it due to space restrictions. DPC is only applicable to child/descendant queries, whereas Prev.Step can be applied to all queries compared. XSeed does not support parent/ancestor queries and LWES does not support queries with predicates. Therefore, we cannot make a true Cartesian comparison.

We analyzed the accuracy of simple child and descendant queries in Table 3. We can see that EXsum itself delivers good estimations, except for cases where homonyms are scattered across the document (*nasa*, *swissprot*). In this case, Prev.Step cannot contribute with quality results. In general, however, DPC gains the best results for

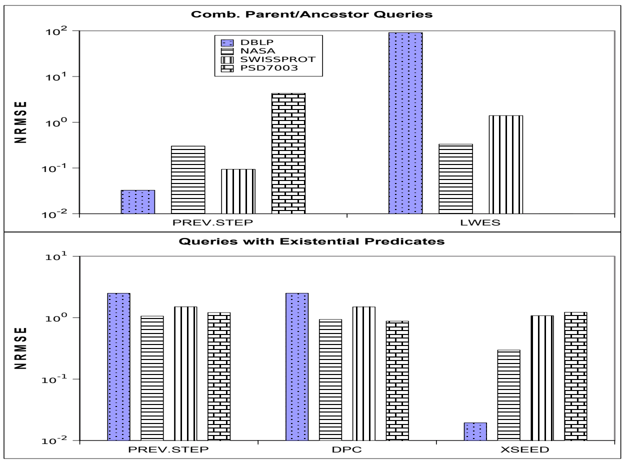

Fig. 5. Par./Anc. queries and predicates

the majority of cases and Prev.Step can provide high quality estimations on documents whose degree of structural variability is very low (*dblp* and *psd*7003). Furthermore, we have investigated the estimation quality for queries with parent and ancestor axes and queries with predicates (see Fig. 5). For the former, Prev.Step delivered high-quality estimations in most cases, comparable to or even better than LWES. 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 most of the cases and especially good ones for *dblp*.

4 Conclusion

In this paper, we have extended EXsum, an element-centered summary, to capture more statistical information on XML documents and, using this information, to support the estimation of a richer set of path expressions than possible with the base method. We have made a set of experiments to quantitatively evaluate our proposal against approaches published in the literature. Evaluating the new methods proposed for EXsum, DPC has delivered quality results for estimating structural path expressions, whereas the Prev.Step method has reported high errors concerning cardinality estimation for queries with child/descendant axis. For queries with parent/ancestor axes, Prev.Step has presented the best results. This empirical observation may lead us to infer that using only a single method to estimate all possible path expressions is not recommendable. More research is therefore needed to prove this finding.

References

1. Aboulnaga, A., Alameldeen, A.R., Naughton, J.F.: Estimating the selectivity of xml path expressions for internet scale applications. In: Proc. VLDB Conference, pp. 591–600 (2001)
2. Aguiar Moraes Filho, J.d., Härder, T.: Tailor-made xml synopses. In: Proc. BalticDB&IS Conference, pp. 25–36 (2008)
3. Freire, J., Haritsa, J.R., Ramanath, M., Roy, P., Siméon, J.: Statix: making xml count. In: SIGMOD Conference, pp. 181–191 (2002)
4. Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Parr, R.: Xpathlearner: An on-line self-tuning markov histogram for xml path selectivity estimation. In: Proc. VLDB Conference, pp. 442–453 (2002)
5. Polyzotis, N., Garofalakis, M.N.: Xsketch synopses for xml data graphs. ACM Trans. Database Syst. 31(3), 1014–1063 (2006)
6. Wang, W., Jiang, H., Lu, H., Yu, J.X.: Bloom histogram: Path selectivity estimation for xml data with updates. In: Proc. VLDB Conference, pp. 240–251 (2004)
7. Zhang, N., Özsu, M.T., Aboulnaga, A., Ilyas, I.F.: XSeed: Accurate and fast cardinality estimation for xpath queries. In: Proc. ICDE Conference, p. 61 (2006)
8. Aguiar Moraes Filho, J.d., Härder, T.: EXsum—an xml summarization framework. In: Proc. IDEAS Symposium, pp. 139–148 (2008)
9. Haustein, M.P., Härder, T.: An efficient infrastructure for native transactional xml processing. Data Knowl. Eng. 61(3), 500–523 (2007)