Tailor-made XML Synopses

Jose de Aguiar MORAES FILHO, Theo HÄRDER
University of Kaiserslautern, 67663 Kaiserslautern, Germany

Abstract. We use a set of tailor-made techniques adjusted to XML particularities to summarize XML data, which then can be recursively evaluated for query optimization tasks. More specifically, we identify and deal with special cases for effectively using histograms for the summarization of structural aspects of XML documents and also cases for which histogram use is inappropriate. We perform comparative experiments using our native XML database management system called XTC.

Keywords. XML summarization, Histograms, Query processing

1. Motivation

The structure of XML documents brings new challenges on how to shrink tree representations in a way to be representative enough for the original document to support a cost-based XML query optimizer. XML documents may be, in practice, irregular with populated and broad subtrees and, at the same time, sparse and slim subtrees. Beside the structure part made up by elements and attributes, the content part (i.e., the tree leaves) normally contain text values which may embody varying distributions. XQuery and XPath [17, 18], including their predicates, may contain up to 8 axis relationships, where parent-child (/) and ancestor-descendant (//) are absolutely dominant in query workloads and prime candidates for optimization support. Predicates may be so rich that a single auxiliary structure (as complex as conceivable) would not solve all query optimization problems. Moreover, maintenance overhead would confine each practical approach. Thus, any summarized structure should be flexible enough for both: capturing such properties of XML data and supporting, as accurate as possible, cost-based XQuery/XPath query optimization.

We propose two synopses which provide good built-in support to be used in a query optimizer where the evaluation process take several physical operators (e.g., structural join (STJ) and holistic twig join (HTJ)) into account. In Section 2, we detail the synopses to summarize structural aspects of XML documents; our approaches are called Constant Element Frequencies under an Aggregated Parent and Level-Wide Element Summarization, CEF and LWES for short. Both of them leverage the use of histograms [6] to capture element distributions in an XML tree. They apply histograms to specific sets of elements in a tailored fashion by distinguishing situations where the use of histograms is profitable and those where histograms are useless. Furthermore, we propose a simple extension of histograms (called ABD, average bucket descriptor) to lower the estimation error when histograms are used in the XML domain. Section 3 presents a set of formulas to derive cardinality estimation for XML queries. These formulas used in our experiments could be exploited by an optimizer. We have implemented the proposed concepts in XTC [5] and proven their validity by comparative experiments (Section 4). Section 5 presents the related works. Finally, Section 6 concludes the paper.

2. The Proposed Approaches

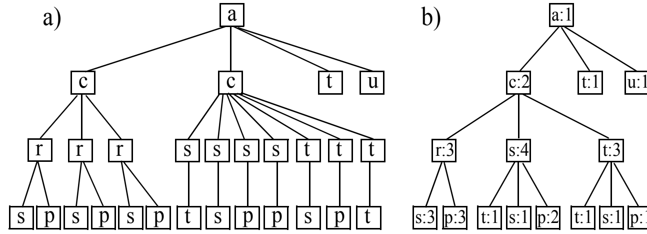


Figure 1 a) XML document and b) HNS structure

We can naively summarize an XML document by a general kind of path synopsis called Hierarchical Node Summarization (HNS–see Figure 1). It embodies a structural summary of all (sub-) paths of the document. Each node in an HNS defines a path class and stores the number of its path instances occurring in the document. The HNS construction is recursive and captures the frequencies of all elements under the same (aggregated) parent in a document. Building an HNS proceeds top-down, where the same element names under the aggregated parent are counted, and this information is represented by a node labeled with "*element:frequency*", where *element* is the element name and *frequency* is the number of occurrences counted. For example in Figure 1a, we have two elements *c* under parent *a* and, in turn, 3 elements *r* under a parent *c*: these elements are represented in the HNS of Figure 1b by an aggregated node *c* with frequency 2 (c:2) and, in turn, by an aggregated node *r* with frequency 3 (r:3). It turns out that such an HNS precisely preserves the frequency information of all (sub-)paths of the original document. If we need the frequency of a path, we just traverse this path in the HNS from the root and the frequency kept in the final element addressed by the path delivers this information, e.g., *a/c/r/p* and *a/c/s* yield 3 and 4, respectively.

HNS has strong positive points. First, it delivers high estimation accuracy for structural queries. Because the query *//c/t* matches two nodes in the given HNS, we can immediately return 4 as the number of qualified path instances (together with the fully specified paths). Furthermore, all path classes in a document are present in an HNS, which avoids false positive errors. Second, HNS estimates most XPath axes with very high accuracy. An estimation for *//c/s/following-sibling::t* returns 3, because, in HNS, a node *t* with frequency 3 has the same parent as *s*. Third, HNS is memory efficient for documents exhibiting a certain degree of uniformity.

However, HNS has also negative points. The number of HNS nodes may be high for deeply-structured documents and, because the HNS tree has to be traversed for some query axes, the number of nodes may negatively impact the estimation process. For deeply-structured documents, HNS may consume an enormous amount of storage space which could impede cardinality estimation and, in turn, the entire query optimization.

Attempting to avoid weaknesses and limitations and, at same time, keeping the positive properties of HNS, we develop a framework to enable tailor-made HNS structures based on two approaches. Both CEF and LWES, approximate all paths in XML documents without any pruning and use histograms to estimate structural information.

2.1. CEF

The observation that child sets having the same element name as parent (e.g., *s* and *p* under *r* in Figure 1a) frequently exhibit a similar element distribution led to the develop-

ment of the CEF method. It assumes a certain stable repetition (reasonably uniform distribution) of such patterns of parent-child sets. Hence, this property serves to save storage space. The resulting CEF can be considered as a tree consisting of the inner HNS nodes and specific compacting structures. The HNS leaf nodes and their distribution could be expressed by bitlists (in case of uniform frequencies) or histograms attached to these inner HNS nodes. In case storage saving is not possible, uncompressed *element:frequency* lists may be used.

Let us take an intuitive look on how a CEF can be built (see Figure 2). Starting from an HNS, we can compress the XML document as follows. As a general rule, each set of leaf nodes in the HNS is represented in CEF by its parent node and a histogram which captures the frequency distribution of this set. For instance, we take the *set C* whose parent node is ***s:4*** and represent it in the CEF structure by a histogram together the parent node itself. This rule could recursively be applied in such a way that the final CEF structure consists of only inner nodes and histograms. Of course, special cases may occur in which histograms are inappropriate (see, e.g., *set A* in Figure 2); we deal with them in Section 2.3. The main argument to favor CEF construction is to reduce the number of nodes in the resulting CEF synopsis which can yield a faster search time when CEF is used in the optimization process.

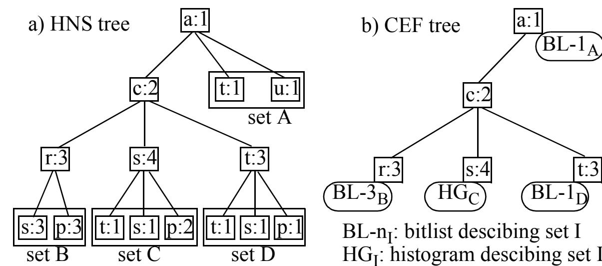


Figure 2 Deriving a CEF tree from an HNS tree

2.2. LWES

A different way to compress a HNS tree is to capture the distribution of the elements level by level. For example, Figure 3a shows three nodes with element name *s* in level 4, one *s:3* under *r*, another *s:1* under *s*, and again an *s:1* under *t*; let us call them node occurrences of s. LWES represents all such occurrences of *s* at the same level by using a single histogram. In a similar way, this rule is applied to all other HNS nodes, that is, *p* and *t* (italicized in Figure 3b at level 4). Moreover, LWES maintains a list of parent pointers which has a twofold goal: it properly captures the hierarchy (parent-child relationships) of the document and it helps to distinguish each HNS node occurrence of an element. Both are exploited during the cardinality estimation process.

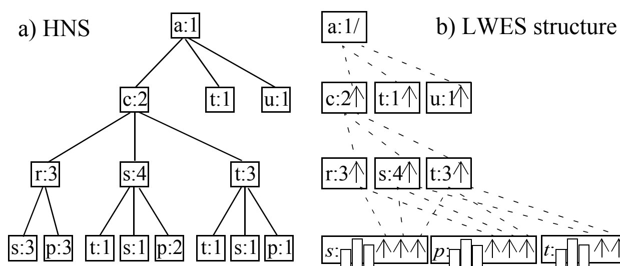


Figure 3 Example of an LWES synopsis (cut-out)

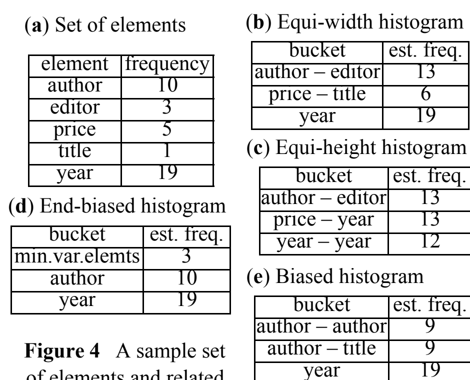
The LWES approach is an alternative solution which tries to deal with recursivity in XML documents such as *treebank*, but it may also be beneficial for others, e.g., *dblp.* LWES has at least one advantage over CEF. An element name (node) at a level is repre-

sented by only a single histogram in LWES. CEF must possibly use more than one histogram to represent an element at a level. There are, also in LWES, special cases in which histograms are inappropriate and we deal with them in next section.

2.3. Tailored XML Compression

The element names occurring in an HNS tree are not continuous and have no natural ordering. Instead, we have to deal with non-ordered discrete data spaces. Therefore, parametric distributions [10] can not be applied. However non-parametric estimation techniques, e.g., histograms, may be suitable for the compression of *element:frequency* lists. Various forms of histograms [6]—all observing the standard assumptions of *uniform element/value distribution* and *element independence*—were proposed so far; we sketch the most important ones. According to [6], a histogram on a set X is constructed by partitioning the data distribution of X's elements into $\beta (\beta \geq 1)$ mutually disjoint subsets called *buckets* and by approximating frequencies and values in each bucket in some common fashion, normally by averaging frequencies. This definition allows a degree of freedom in which we can do both: (*i*) adapt the histogram definition to our needs, and (*ii*) use several types of histograms proposed in the literature.

There are, of course, several types of histograms. Four of these well-known types are: Equi-width (*EW*), Equi-height (*EH*) [12], End-biased (*EB*) [8] and Biased histograms [14]. To illustrate these histograms, consider a set of *NE* elements as depicted with *NE* = 5 in Figure 4a where each element is annotated with its frequency (freq.), i.e., its number of occurrences. This set could represent a complete subtree or a set of elements at a specific level of a document and has to be mapped onto β buckets (β ≤ *NE*). In our illustrations, we use three buckets to represent such a set. While keeping the alphabetic order, an *EW* histogram (Figure 4b) groups *NE*/β elements together with their sum of frequencies in a bucket (with the leftover in the last bucket). Each bucket is then labeled with a start element and an end element, where the start element is the first entry in the bucket, and the end element is the last entry in the bucket. If a bucket holds only one entry, it will have equal start and end elements. In contrast, *EH* computes the sum *S* of the individual element frequencies and sets *S/NE* as the “equal height”. With this criterion, the entries of the original set are partitioned in an order-preserving way into buckets (Figure 4c). If the frequency contribution of the end element is not fully contained in the bucket frequency (est. freq.), this element will appear as start element in the subsequent bucket thereby spanning two (or more) buckets (see *price – year* and *year – year* buckets in Figure 4c). The biased histogram types try to emphasize particular elements while they approximate the remaining elements. Some degrees of freedom are conceivable, e.g., emphasizing elements with highest or highest/lowest frequencies or averaging elements with minimum variance. In our example in Figure 4d, *EB* selects *NE-(β-1)* elements which exhibit the minimum



(**a**) Set of elements

element	frequency
author	10
editor	3
price	5
title	1
year	19

(**b**) Equi-width histogram

bucket	est. freq.
author – editor	13
price – title	6
year	19

(**c**) Equi-height histogram

bucket	est. freq.
author – editor	13
price – year	13
year – year	12

(**d**) End-biased histogram

bucket	est. freq.
min.var.elemts	3
author	10
year	19

(**e**) Biased histogram

bucket	est. freq.
author – author	9
author – title	9
year	19

Figure 4 A sample set of elements and related histograms

variance and represents them by a single bucket with their frequency average. The remaining β-1 elements are represented by individual (singleton) buckets. Here, the *EB* histogram isolates the elements *author* and *year*, and averages the remaining elements (min. var. elemts) in a bucket. A Biased histogram (Figure 4e) isolates the element with the highest frequency (*year*) and approximates the remaining elements in an *EH* way.

The direct and straightforward application of histograms may not be appropriate in all cases for XML data. In fact, some special situations exist in which histograms cannot contribute to further compression. These cases are described and dealt with as follows. **Case *1***: all elements in a set have the same frequency. In this case, we do not need a histogram at all. We only store a bitlist representing the position of these elements in the set. Hence, for *set A* (*set B* and *set D*) in Figure 2, adding frequency (e.g. 1 for *set A*) is sufficient to compute all selectivities in the set. **Case *2***: as a special case of 1, a set has only one element; hence, insertion of a node representing this element together with its frequency in LWES/CEF is sufficient. **Case *3***: a set has two elements with varying frequencies. We could approximate such a situation by averaging frequencies. However, if frequencies contain a large variance, the use of averages does not well reflect the actual distribution. Therefore, we store such sets as uncompressed *element:frequency* lists.

An important issue is how the different histogram types influence cardinality estimation and, in turn, the optimization precision of query types. A query encompassing XPath (XQuery) axes as parent, child, ancestor and descendant can be viewed as an exact-match query. For example, retrieving all names of book authors, */book/author/name*, could be translated to the exact match of the child *name* (among others) whose parent is *author*. In this case, most histograms support such a query type, but EB has one of the best accuracy results for exact match in general [7]. On the other hand, queries containing preceding (following) axes could be evaluated in a similar way as range queries. For example, getting all book titles where the related books have a publisher, *//publisher/following-sibling::title*, retrieves the "next siblings" of *title* after *publisher*. For this case, Biased and EH histograms can be useful, although [7] shows that EB histograms provide also good estimations. The application of histograms in our approaches can be even tailored to specific parts of a document driven by the query workload. In this case, if we know that a typical query workload addressing a specific fragment of a document (obtained, e.g., from a query feedback mechanism) mostly contains exact-match queries, we can apply an EB histogram for this fragment, making the cardinality estimation more accurate and, possibly, generating better query plans (QEP).

Another issue, regarding histograms application, is whether or not we should make extensions for some histogram types. For example, the use of *EB* histograms in the CEF approach may lead to undesired situations when evaluating a query axis, e.g., the descendant axis. When the estimator process probes an element against a path instance in a descendant axis, two possibilities can happen: if the element is in a singleton bucket, no error occurs. Otherwise, the average bucket value is returned as the estimation which can yield a false-positive error. This may happen because there is no explicit representation of elements in the average bucket. Depending on the path expression, this error may be propagated and may finally result in a lower estimation accuracy. To overcome this situation, we propose a new extended *EB* histogram, called EB-ADB (Average Bucket Descriptor). We thus add to *EB* histograms a descriptor that represents all elements in the average bucket (ABD). The implementation of ABD is quite simple: a compressed bit array where '1' in a given array position represents an element within

the range of elements belonging to an average bucket. With ABD, we reduce the false-positive error in CEF and, of course, its propagation, because we can now identify exactly whether or not an element is in an *EB* histogram. For LWES, this problem does not occur, because the parent pointers work as descriptors for the average bucket. The building algorithms of CEF and LWES have a complexity of only *O(n)*, where *n* is the number of HNS nodes. For space restrictions, we have omitted them.

3. Reference Queries

So far, we have discussed how path synopses can be enriched with various forms of summary information and how this information, in turn, can be effectively compressed by histograms of different types. Now we will outline for which query expressions and in which way these summaries can be exploited, that is, how selectivities for path expressions can be estimated to enable the query optimizer to derive the 'best' or, more realistic, sufficiently good QEPs.

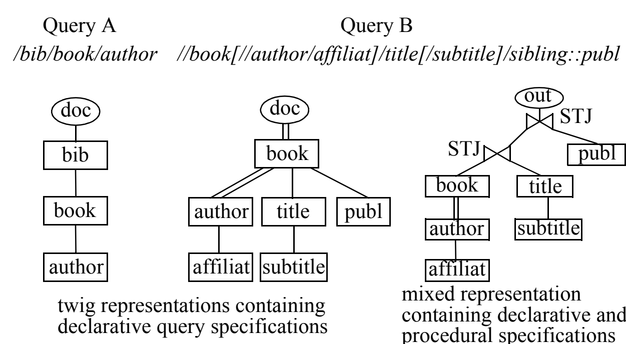


Figure 5 Various forms of query representation patterns

To facilitate our discussion, look at Query A in Figure 5, */bib/book/author*. The terms *bib*, *book,* and *author* are node tests in XPath terminology; and */* is called path step. In this sense, an XPath/XQuery expression consists of a number of node tests connected by path steps evaluated during query processing. Furthermore, various forms of path predicates, represented by brackets, may occur as depicted in Query B of Figure 5. Predicates check for the existence of one or more elements in expressions inside brackets. Both, queries A and B can be internally represented by a set of structural join operators (STJ), by a twig which can be executed using a holistic twig join operator (HTJ) [2], or by a set of twigs representing parts of the query, linked by structural joins. In any case, the optimizer should estimate the cost of each possible QEP for a query. Here, cardinality estimation plays a key role. For example, for Query A using structural joins, we can have at least two possible ways of evaluation: *(bib* STJ *(book* STJ *author))* and *((bib* STJ *book)* STJ author*)*. Furthermore, each structural join could be executed using hash, nested-loop or merge join operators [11]. More specifically, if we estimate that the *author* selectivity is less than the *book* selectivity, the STJ operator in the inner parenthesis should look up first *author*. Otherwise, STJ should look up *book*. The same reasoning is valid for query predicates if they exist.

In the following sections, we sketch formulas for cardinality estimation of path expressions. The gist of our estimation method is to work with two concepts: context (C_n) node test and target (T_n) node test. C_n can represent an XQuery/XPath bind variable, while T_n can represent a child, descendant, parent, or ancestor node in a query. Any part of a query expression can be characterized by these two concepts. For example, in */bib/book/author*, three C_n (*document root* for */bib*; *bib* for */book*; and *book* for */*

author) and three T_n (*bib* for */bib*; *book* for */book*; and *author* for */author*) occur. For our estimation methods, the document root node cardinality is 1.

CEF and LWES support the evaluation of several kinds of path expressions. However, in this paper, we only focus on two frequent and, therefore, important types of queries: Simple Path Expressions (**SP**) and Predicate Path Expressions (**PP**). Both of them encompass queries presented in Figure 5, e.g., Query A relates to SP and Query B relates to PP. To define the expressions at a more abstract level, we derive the following definitions. Simple Path Expressions (**SP**) are XPath expressions in one of the following formats: ***(a)*** $/v_1/.../v_{n-1}/v_n$, and ***(b)*** $/v_1/.../v_{n-1}//v_n$. Predicate Path Expressions (**PP**) are XPath expressions whose formats include: ***(a)*** $/v_1/.../v_{n-1}[./v_n]$; ***(b)*** $/v_1[.//v_2].../v_n$; ***(c)*** $/v_1[./v_2$ *AND* $./v_3$ *...]/.../*v_n, *and* ***(d)*** $/v_1/[./v_2$ *OR* $./v_3...]/.../v_n$. In SP and PP expressions, $v_1...v_n$ are node tests, and / (//) are path steps representing parent-child (ancestor-descendant) relationships. Each predicate can be formed by one or more components. Each predicate component can, in turn, be a complete SP expression and possibly linked by logical connectors (AND/OR). For example, the predicate [.//author AND ./affiliat] has two components: .//author and ./affiliat, both linked by an AND connector. We allow existential predicates in every part of a PP.

Given a path step S, C_n/T_n, we estimate the cardinality of T_n, denoted ECard ($/T_n$), by $\text{ECard}(/T_n) = \text{GetTarget}(T_n)$. *GetTarget* (T_n) is a function that searches CEF/LWES for T_n under C_n to get the estimate. If histograms exist for C_n, the function returns the histogram estimations. Consider Query B in Figure 5. We could recursively apply the *ECard* formula (and formulas in the next sections) to estimate, for example, the cardinality of the twig. Furthermore, we could use such formulas to estimate the selectivity of each predicate and decide, e.g., on the join order of the STJ operators.

3.1. Estimating Cardinality of SP

The selectivity for a path step, denoted by Sel_{child}(S), can be estimated as $Sel_{child}(C_n/T_n) = (\text{ECard}(T_n))/(\text{ECard}(C_n))$. Sel_{child} could be used by a query optimizer to derive the selectivity estimation of each path step and then to decide the join order used to evaluate a QEP with, for example, structural join operators. We can estimate the cardinality of an SP expression with format **a** as $SP_{Card}(v_1/\ldots/v_n) = \text{ECard}(v_n)$, which means that the cardinality of an SP expression is just the cardinality of its last step. Using the concepts of SP_{Card} and Sel_{child}, the estimated selectivity of an SP expression with format **a** can be obtained through the following: $SP_{sel}(v_1/\ldots/v_{n-1}/v_n) = (\text{ECard}(v_n))/(SP_{Card}(v_1/\ldots/v_{n-1}))$. Dividing the cardinality of the last path step by the SP cardinality, SP_{sel} captures the contribution of (elements in) the last step with regard to the whole path. The concepts SP_{sel} and SP_{Card} are useful when a set of HTJ is applied to evaluate a query. For example, refer to Figure 5 (query B, right hand). SP_{sel} could be used to drive the join order of the twigs: *book/author/affiliat*, */title/subtitle*, and */year*. In fact, SP_{sel} could be applied to all twigs to estimate which of them is the most selective. The concepts Sel_{child} and SP_{sel} are useful, because we can exploit SPs as predicate components. Therefore, we need such concepts to calculate predicate selectivities. We consider the above formulas as core formulas, because they can be recursively used to derive estimations in a query evaluation process.

3.2. Estimating Selectivity of PP

Given path step S, $C_n[./T_n]$, with context node C_n and target node T_n in a predicate expression, the selectivity estimate, Sel_ρ, for a predicate expression with the format **a** is calculated as $Sel_\rho(v_1/\ldots/v_{n-1}/[./v_n]) = SP_{sel}(v_1/\ldots/v_{n-1}) \times Sel_{child}(./v_n)$. Sel_ρ simply states that the more selective a predicate is, the more selective the whole expression is. Of course, this is a classical assumption and it is further used in predicate expressions linked by logical operators. Let $\rho_{AND} = (\rho_1, \rho_2, \ldots, \rho_n)$ be a set of predicates using logical AND (^) connectors linking predicates ρ_i, and each ρ_i is an SP expression. The selectivity $Sel_{\rho AND}$ is estimated as

$$Sel_{\rho_{AND}}(v_1/\ldots/v_i/[./v_{i+1} \wedge \ldots \wedge ./v_{i+n}]) = SP_{sel}(v_1/\ldots/v_i) \times \prod_{1 \le k \le n} Sel_{child}(./v_{i+k})$$

The idea behind predicate selectivity computation is to estimate the weight of a predicate w.r.t. the entire path expression. For AND-connected predicates, the resulting selectivity is computed the product, whereas OR-connected predicates are calculated by the sum.

3.3. Estimating Descendant Axes

Given a path step S, $C_n//T_n$, with a context node C_n and a descendant target node T_n in an SP expression, we may have several target nodes T_n reachable from C_n. Let $\tau = (t_1, t_2, \ldots, t_n)$ be a set of target T_n nodes reachable from C_n, the estimated cardinality of a descendant axis expression is $\mathrm{ECard}_{desc}(C_n//T_n) = \sum_{1 \le i \le n} \mathrm{GetTarget}(t_i)$, where $t_i \in \tau$. Using $ECard_{desc}$, the cardinality and selectivity estimation formulas for descendant axes are similar to those in Section 3.1, and we omit them because of space restrictions.

4. Empirical Evaluation

To determine the practical use of our proposals, we have performed a number empirical experiments using a well-known set of XML documents listed in Table 1 and whose results are analyzed regarding timing, sizing and accuracy. Regarding estimation accuracy, we build, for each document, a query workload relating to query types in Section 3 and compute the cardinality estimation according to formulas presented contrasting actual and estimate values and ploting the results in Section 4.2.

Table 1: Characteristics of documents considered

Doc-name	Descrip-tion	Size in MB	#nodes (inner / text)	#voc. names	max. depth	avg. depth	Observation
uniprot	Protein data	1,820	81,983,492 / 53,502,972	89	7	4.53	huge size, quite regular, non-recursive
dblp	Comp. Sc. index	330.0	9,070,558 / 8,345,289	41	7	3.39	middle size, less regular, non-recursive
psd-7003	Protein data	716.0	22,596,465 / 17,245,756	70	8	5.68	big size, quite 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 J.	86.1	2,437,667 / 1,391,845	251	37	8.44	middle size, completely irregular, highly recursive

In Table 1, column *#nodes* represents the total number of nodes in a document according to the DOM specification [17]. Column *#voc. names* indicates that the num-

ber 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 *uniprot* 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, some documents are less regular and have in some cases huge sizes (*uniprot*).

We compare our solutions (CEF/LWES) against Markov Tables (MT) [1] and XSeed [18]. For Markov Tables, we have implemented two MT's compression strategies: Suffix-*(MTSuf*) using 30 entries in MT and No-*(MTNo*) using 90 entries. Moreover, we have also implemented an uncompressed MT(MTUcomp). For all MT implementations, we have used the pruning parameter *m*=2, indicating that paths down to length 2 are not represented in MT. XSeed uses a graph to capture XML document paths and allows false positives in the estimation results. It uses a pruned search on this graph to estimate path cardinalities.

We have performed our experiments using a 2-GHz Pentium Centrino Duo processor with 1 GB main memory, running under Windows XP SP2. We have implemented and integrated all approaches into XTC which is written in Java 6. Throughout the performance measurements, we use 512 MB of main memory for the Java Virtual Machine and 16 MB for the XTC buffer. For CEF/LWES, we analyze the behavior of two histograms types: *End-biased* (*EB*) and *Equi-height* (*EH*). The reasons for choosing such types are simple. *EB* histograms typically exhibit the least approximation error, whereas *EH* histograms are commonly used in real DBMS configurations. Because of limitations of EW histograms [10], we do not experiment with them.

XSeed and MT have low building times (typically some seconds). But the times needed for CEF and LWES construction are in the sub-second range and, therefore, absolutely negligible. If we take into account the whole synopsis building process including document scan time, it typically takes just a few minutes. Even for a deeply-structured document such as *treebank*, we have computed LWES in just about 0.8 seconds and XSeed in approximately 0.18 minutes. When subtrees are inserted into the document, the corresponding summarization structures CEF and LWES can be directly updated and need no reconstruction. These findings demonstrate that building and maintenance algorithms are good and scalable. We do not comment on them any further.

4.1. Sizing Analysis

Memory space consumption demonstrates that in all tested documents, except *treebank*, we can reach 4–6 orders of magnitude less than document size. For treebank, we take 2 orders of magnitude. For example, in almost all documents, the sizes vary between 0,001% and 0,055% of the document sizes (Figure 6). Even for *treebank*, the sizes for the summarization structures consume 0.4% (XSeed) to 2.3% (CEF) of the document size. MTSuf* structures exhibit the least space consumption for highly recursive documents, outperforming thus LWES/CEF for such documents. Variations of CEF sizes are due to histogram type and irregularity degree of XML documents. An interesting result illustrating the influence of irregularity is shown by the CEF sizes of *nasa* and *uniprot*. *Nasa* is a 25MB document whereas *uniprot* has 1.8GB. However, the former has at most a CEF size of 0.78KB whereas the latter has only slightly more (0.95 KB). Being much larger than *nasa,* it illustrates that *uniprot* has a much more regular tree structure.

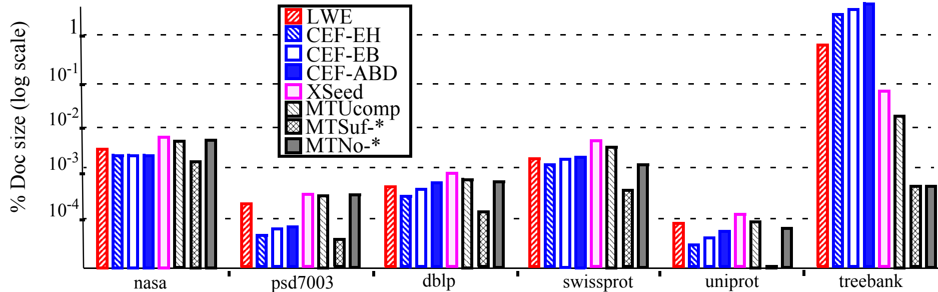


Figure 6 Synopses size analysis

4.2. Estimation Accuracy

To estimate the accuracy of all approaches and to facilitate cross-comparison, we apply the two metrics used for XSeed and MT [1, 18]: Relative Error (RE) and Normalized Root Mean Square Error (NRMSE). RE is given by the following formula: $E = |e - a| / a$, where *e* is the estimated value and *a* is the actual value. NRMSE measures the average error per unit of the accurate result and is defined in [18]. For this purpose, we have divided the query workload into three classes: SP-Child and SP-Desc relate to formats **a** and **b** of SPs, respectively, whereas PP-pred encompasses PP formats (see Section 3). For each class, we have calculated cardinality and selectivity, thereby contrasting estimates and actual values, and applied the metrics to all document considered. For the estimation of cardinality/selectivity, we executed the combined query workload on all documents. For plotting the results, we discarded the best estimations, i.e., estimations very close to the actual values. We thus plot the rest of them, including, of course, the worst cases.

In Figure 7, we show a comparison between CEF using *EB* and *EH* histograms, and LWES. CEF-*EB* produces, in general, higher errors than *EH* ones. This confirms the problem with *EB* histograms in XML summarization mentioned in Section 2.3. Thus, when we apply our proposed histogram extension (ABD), we reach much better accuracy. Clearly, this is a trade-off situation between storage space and accuracy. Therefore, we have to check whether or not the increase on size justifies the benefit of lowering the false-positive error. In our experiments, we obtain, by using CEF ABD, an average increase of only 7.4% on CEF sizes. Hence, with such a small increase, we reach very good estimation results. Because CEF trees built with *EH* and *EB* histograms do not allow to derive good estimation results, we use only CEF trees with *EB*-ABD histograms (CEF ABD) in further comparative benchmarks.

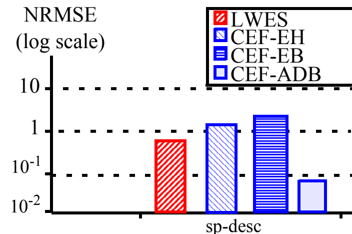


Figure 7 Cardinality estimation error CEF x LWES

In Figure 8, we compare CEF and LWES against MT and XSeed. Our proposals outperform the others in almost all cases. XSeed does not behave well for non-recursive XML documents. For example, in *uniprot*, XSeed has an estimation error similar to MTSuf* (about 14% NRMSE error), which is twice as worse than that of LWES (7%). MT approaches tend to underestimate queries with long paths, even if these queries are posed on non-recursive documents, as *swissprot* and *psd7003*.

Although the XSeed kernel is designed for highly recursive documents, it does not provide good estimates, compared to LWES. The specific reason is the attempt to minimize false positives by using a pruned search method in the XSeed estimation algorithm which is controlled by a tuning parameter. Clearly, if this parameter is large, XSeed [18] tends to give better results, obviously, at the cost of increased evaluation overhead. In [18], the authors reported an NRMSE error of 169% (kernel) for only a 4MB treebank document. We have summarized the XSeed estimation errors on an entire (86MB) *treebank* document and found that, applying the same pruned search, the error is even higher (see Figure 8). In contrast, LWES has an average error of <0,02% for the entire *treebank* document.

Although still small for memory-resident use (see Section 4.1.), one may argue that it is difficult to keep a large LWES synopsis in memory. In this case, a possible strategy is to load parts of the structure on demand. For example, we may load LWES level by level according to query optimizer use. Such a strategy would keep only the top-most levels in memory, say the first 6 levels, and would fetch nodes from deeper levels on demand. In fact, LWES lends itself to evaluation based on such fragments. Because most of the queries only encompass the top-most levels, optimization heuristics could additionally be successfully applied.

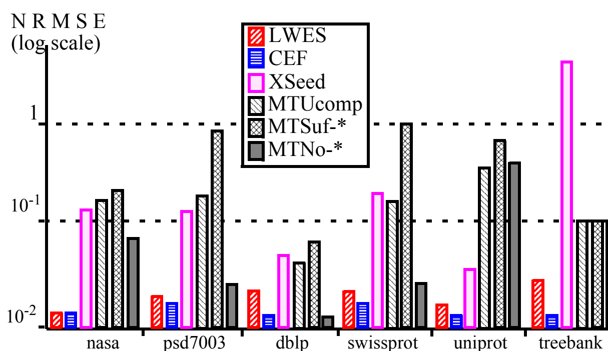


Figure 8 Cardinality estimation error

5. Related Work

Markov Tables [1] and XPathLearner [9] use a tree-based summarized structure. They also prune paths at a certain level; usually, they keep paths only down to level 2 and use a statistical model to estimate longer paths. Bloom Histograms [16] capture structural information on entire root-to-leaf paths. Thus, they cannot deal with path predicates addressing specific nodes. StatiX [3] also uses histograms to describe node distributions. However, it depends on XML schema information and is targeted to map XML-to-relational. XSKETCH [14] and XSeed [19] are graph-based structures for modelling structural information. Both use the concept of graph bi-similarity to summarize XML elements which means that some information is lost in the summarization process. Furthermore, [19] is specifically designed for highly recursive XML documents and uses an optional tabular structure (called HET) to estimate XPath queries. This tabular structure is built by searching the graph structure using a query feedback mechanism. However, this tabular structure is limited by a threshold confining the number of predicates to be evaluated for a query.

6. Conclusion

In this paper, we proposed the use of techniques adaptable to the particularities of XML data. We leverage histograms as one of the ways to further compress summarization structures of XML documents. First, we used them on node sets (CEF) at each sub-tree and then extended this idea to entire levels (LWES). Additionally, we coped with cases in which histograms are useless together with ways to shrink the XML tree. Our proposals are at least comparable to competitive approaches. Only CEF is less suitable for the mapping of deeply-structured XML documents, such as *treebank*, although it is the technique of choice for (even huge) documents exhibiting a certain degree of uniformity. On the other hand, LWES scales well for all types of documents. Although [18] needs less storage, LWES is more adequate for cardinality estimation, because it has a far lower error rate and enables load on demand for synopses too large to be kept in memory. The maintenance effort of CEF/LWES structures is another positive point. We have detected a problem with *EB* histograms and proposed an effective solution for it, called ABD, whose overhead is of only 7% on synopsis sizes.

Motivated by our results, we plan to extend our research work in two directions. First, we want to explore the summarization of text values in XML documents (together with structural summarization). Second, we plan experiments with these approaches in real cost-based XML query optimization scenarios.

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., 2001, pp. 591-600.

[2] Bruno, N., Koudas, N., and Srivastava, D. Holistic Twig Joins: Optimal XML Pattern Match. In Proc. ACM SIGMOD Conf., 2002. pp.310-321.

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

[4] Härder, T., Mathis, C., and Schmidt, K. Comparison of Complete and Elementless Native Storage of XML Documents, in: Proc. IDEAS 2007, Banff, Canada, 2007, pp. 102-113.

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

[6] Ioannidis, Y. The History of Histograms (abridged). In Proc. VLDB Conf., 2003, pp. 19-30.

[7] Ioannidis, Y., Poosala, V. Histogram-based Solutions to Diverse Database Estimation Problems. IEEE Data Engineering Bulletin, September, 2005, pp.10-18.

[8] Ioannidis, Y., Poosala, V. Balancing histogram optimality and practicality for query result size estimation. In Proc. ACM SIGMOD Conf., 1995, pp. 233-244.

[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., 2002, pp.442-453.

[10] Mannino, M. V., Chu P., and Sager, T.: Statistical Profile Estimation in Database Systems. ACM Comp. Surveys 20:3 1988, pp. 191-221.

[11] Mathis, C., Härder, T. Hash-Based Structural Join Algorithms. In Proc. EDBT Workshops (DATAX), 2006, pp. 136-149.

[12] Piatetski-Shapiro, G., and Connel, C. Accurate Estimation of the Number of Tuples Satisfying a Condition. In Proc. ACM SIGMOD Conf., 1984, pp. 256-276.

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

[14] Poosala, V., Ioannidis, Y., Haas, Peter J., and Shekita, E. J. Improved Histograms for Selectivity Estimation of Range Predicates. In Proc. ACM SIGMOD Conf., 1993, pp. 294-305.

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

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

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

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