XSEED: Accurate and Fast Cardinality Estimation for XPath Queries

Ning Zhang M. Tamer Özsu Ashraf Aboulnaga Ihab F. Ilyas

School of Computer Science, University of Waterloo
{nzhang, tozsu, ashraf, ilyas}@uwaterloo.ca

Abstract

We propose XSEED*, a synopsis of path queries for cardinality estimation that is accurate, robust, efficient, and adaptive to memory budgets.* XSEED *starts from a very small kernel, and then incrementally updates information of the synopsis. With such an incremental construction, a synopsis structure can be dynamically configured to accommodate different memory budgets. Cardinality estimation based on* XSEED *can be performed very efficiently and accurately. Extensive experiments on both synthetic and real data sets show that even with less memory,* XSEED *could achieve accuracy that is an order of magnitude better than that of other synopsis structures. The cardinality estimation time is under 2% of the actual querying time for a wide range of queries in all test cases.*

1 Introduction

Cost-based optimization of XML queries requires the calculation of the cost of query operators. Usually the cost of an operator for a given path query is heavily dependent on the number of final results returned by the query in question, and the number of temporary results that are buffered for its sub-queries (see e.g., [13]). Therefore, accurate cardinality estimation is crucial for a cost-based optimizer.

The problem of cardinality estimation for a path query in XML distinguishes itself from the problem of cardinality estimation in relational database systems. One of the major differences is that a path query specifies *structural constraints* (a.k.a. tree patterns) in addition to value-based constraints. These structural constraints suggest a combined combinatorial and statistical solution. That is, we need to consider not only the statistical distribution of the values associated with each element, but also the structural relationships between different elements. Estimating cardinalities of queries involving value-based constraints has been extensively studied within the context of relational database systems, where histograms are used to compactly represent the distribution of values. Similar approaches have been proposed for XML queries [7]. In this paper, we focus on the structural part of this problem and propose a novel synopsis structure, called XSEED[1], to estimate the cardinality for path queries that only contain structural constraints. Although XSEED can be incorporated with the techniques developed for value-based constraints, the general problem is left for future work.

The XSEED synopsis is inspired by the previous work for estimating cardinalities of structural constraints [5, 4, 8]. These approaches, usually, first summarize an XML document into a compact graph structure called a *synopsis*. Vertices in the synopsis correspond to a set of nodes in the XML tree, and edges correspond to parent-child relationships. Together with statistical annotations on the vertices and/or edges, the synopsis is used as a guide to estimate the cardinality using a graph-based estimation algorithm. In this paper, we follow this general idea but develop a solution that meets multiple criteria. We consider the accuracy of the estimations, the types of queries and data sets that this synopsis can cover, the adaptivity of the synopsis to different memory budgets, the cost of the synopsis to be created and updated, and the estimation time comparing to the actual querying time. We believe that these are all important factors for a synopsis to be useful in practice.

None of the existing approaches considers all these criteria. For example, TreeSketch [8] focuses on the accuracy of the cardinality estimation. It starts off by building a bisimulation graph to capture the complete structural information in the tree (i.e., cardinality estimation can be 100% accurate for all types of queries). Then it relies on an optimization algorithm to reduce the bi-simulation graph to fit into the memory budget and still retain information as much as possible. Due to the NP-hardness of the optimization problem, the solutions are usually sub-optimal and the construction time could be prohibitive for large and complex data sets (e.g., it takes more than 13 hours to construct the synopsis for the 100MB XMark [9] data set

[1] XSEED stands for XML Synopsis based on Edge Encoded Digraph.

on a dedicated machine). Therefore, this synopsis is hardly affordable for a complex data set.

In contrast, XSEED takes the opposite approach: an XSEED structure is constructed by first building a very small *kernel* (usually a couple of KB for most data sets that we tested), and then by incrementally adding/deleting information to/from the synopsis. The kernel captures the coarse structural information in the data, and can be constructed easily. The purpose of the small kernel is not to make it optimal in terms of accuracy; it has to work for all types of queries and data sets, while, at the same time, having a number of desirable features such as the ease of construction and update, a small footprint, and the efficiency of the estimation algorithm. A unique feature of the XSEED kernel is that it recognizes and captures recursions in the XML documents. Recursive documents usually represent the most difficult cases for path query processing and cardinality estimation. None of the existing approaches address recursive documents and the effects of recursion over the accuracy of cardinality estimation. To the best of our knowledge, this is the first work to treat recursive documents and recursive queries.

Even with the small kernel, XSEED provides reasonably good accuracy in many test cases (see Section 6 for details). In some cases, XSEED performs an order of magnitude better than other synopses (e.g., TreeSketch) that use a larger memory budget. One of the reasons is that the kernel captures the recursion in the document, which is not captured by other techniques. Nevertheless, the high compression ratio of the kernel introduces information loss, which inevitably results in greater estimation errors in some cases. To remedy the accuracy deficiency for these cases, we introduce another layer of information, called *hyper-edge table (HET)*, on top of the kernel. The HET captures the special cases that are far from the assumptions that the kernel relies on. Our experiments show that even a small amount of this extra information can greatly improve the accuracy for many cases. The HET can be pre-computed in a similar or shorter time than other synopses, or it can be dynamically fed by a self-tuning optimizer using query feedback. This information can be easily maintained, i.e., it can be added to or deleted from the synopsis whenever the memory budget changes. When the underlying XML data change, the optimizer can choose to update the information eagerly or lazily. In this way, XSEED enjoys better accuracy as well as adaptivity.

Figure 1 depicts the process of constructing and maintaining the XSEED kernel and HET, and utilizing them to predict the cardinality. In the construction phase, the XML document is first parsed to generate the NoK XML storage structure [14], the path tree [1], and the XSEED kernel. The HET is constructed based on these three data structures if it is pre-computed. In the estimation phase, the optimizer calls the cardinality estimation module to predict the cardinality for an input query, with the knowledge acquired from the XSEED kernel and optionally from the HET. After the execution, the optimizer may feedback the actual cardinality or selectivity of the query to the HET, which might results in an update of the data structure.

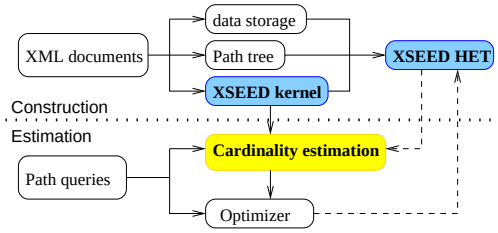


Figure 1: Cardinality estimation process using XSEED

Our contributions are the following:

- We design a novel synopsis structure, called XSEED, with the following properties:
  - The small kernel of XSEED captures the basic structural information, as well as recursions (if any), in the XML documents. The simplicity of the kernel makes the synopsis robust, space efficient, and easy to construct and update.
  - The HET of XSEED provides additional information about the tree structure. It enhances the accuracy of the synopsis and makes it adaptive to different memory budgets.
- We propose a novel and very efficient algorithm for traversing the synopsis structure to calculate the estimates. The algorithm is highly efficient and is well suited to be embedded in a cost-based optimizer.
- Extensive experiments with different types of queries on both synthetic and real data sets demonstrate that XSEED is accurate (an order of magnitude better than the state-of-the-art synopsis structure) and fast (less than 2% of actual running time for all test cases).

The rest of the paper is organized as follows: in Section 2, we introduce the basic definitions and preliminaries. In Section 3, we introduce the XSEED kernel. In Section 4, we present the cardinality estimation algorithm using the XSEED kernel. In Section 5, we introduce optimization techniques to improve XSEED accuracy. In Section 6, we report the experimental results. We compare our approach with related work in Section 7. Finally, we conclude in Section 8.

2 Preliminaries

We assume familiarity with the XML data model and path expressions[2]. We start by giving an example to illustrate the basic ideas, and briefly review concepts whenever necessary. Throughout the paper, we use a n-tuple

[2]The formal definitions can be found in [3].

$(u_1, u_2, \ldots, u_n)$ to denote a path $u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_n$ in an XML tree or a synopsis structure, and use $|p|$ to denote the cardinality of a path expression p (the number of XML elements returned by p).

Example 1 The following DTD describes the structure of an article document.

```
<!ELEMENT article (title, authors, chapter*)>
<!ELEMENT chapter (title, para*, sect*)>
<!ELEMENT sect    (title?, para*, sect*)>
```

By common practice, element names can be mapped to an alphabet consisting of compact labels. For example, the following mapping f maps the element names in the above DTD to the alphabet $\{\mathtt{a}, \mathtt{t}, \mathtt{u}, \mathtt{c}, \mathtt{p}, \mathtt{s}\}$:

```
f(article)=a    f(title)=t    f(authors)=u
f(chapter)=c    f(para)=p     f(sect)=s
```

An example XML tree instance conforming to this DTD and the above element name mapping is depicted in Figure 2(a). To avoid possible confusion, we use a framed character, e.g., $\boxed{\mathtt{a}}$, to represent the abbreviated XML tree node label whenever possible. □

2.1 Recursion

An interesting property of the XML document is that it could be *recursive*, i.e., an element could be directly or indirectly nested in an element with the same name. For example, a `sect` element could contain another `sect` subelement. In the XML tree, recursion represents itself as multiple occurrences of the same label in a rooted path.

Definition 1 (Recursion Levels) Given a rooted path in the XML tree, the maximum number of occurrences of any label minus 1 is the *path recursion level* (PRL). The recursion level of a node in the XML tree is defined to be the PRL of the path from root to this node. The *document recursion level* (DRL) is defined to be the maximum PRL over all rooted paths in the XML tree. □

For example, the recursion level of the path $(\mathtt{a}, \mathtt{c}, \mathtt{s}, \mathtt{p})$ in Figure 2(a) is 0 since each label only occurs once in the path, and the recursion level of path $(\mathtt{a}, \mathtt{c}, \mathtt{s}, \mathtt{s}, \mathtt{s}, \mathtt{p})$ is 2 since there are three `s` nodes in the path.

Recursion could also exist in a path expression. Recall that a path expression consists of a list of location steps, each of which consists of an axis, a NodeTest, and zero or more predicates. Each predicate could be another path expression. When matching with the nodes in an XML tree, the NodeTests specify the tag name constraints, and the axes specify the structural constraints. We classify path queries into three classes: *simple path* expressions that are linear paths containing /-axes only, *branching path* expressions that include branching predicates but also only have /-axes, and *complex path* expressions that contain branching predicates and/or //-axes.

Definition 2 (Recursive Path Expression) A path expression is *recursive* with respect to an XML document if an element in the document could be matched to more than one NodeTest in the expression. □

For example, a path expression `//s//s` on the XML tree in Figure 2(a) is recursive since an $\boxed{\mathtt{s}}$ node at recursion level greater than zero could be matched to both NodeTests. It is straightforward to see that simple and branching path expressions consisting of only /-axis cannot be recursive. Recursive path queries always contain //-axes, and they usually present themselves on recursive documents. However, it is also possible to have recursive path queries on non-recursive documents, when the queries contain the sub-expression `//*//*`. Similarly, we define the *query recursion level* (QRL) of a path expression as the maximum number of occurrences of the same NodeTests with //-axis along any rooted path in the query tree. In general, recursive documents are the hardest documents to summarize, and recursive queries are the hardest queries to evaluate and to estimate.

2.2 Structural Summaries

A structural summary is a graph that summarizes the nodes and edges in the XML tree. Preferably, the summary graph should preserve all the structural relations and capture the statistical properties in the XML tree. There are a number of proposed summary structures. In the following, we only introduce the label-split graph [6], which is the basis of the XSEED kernel.

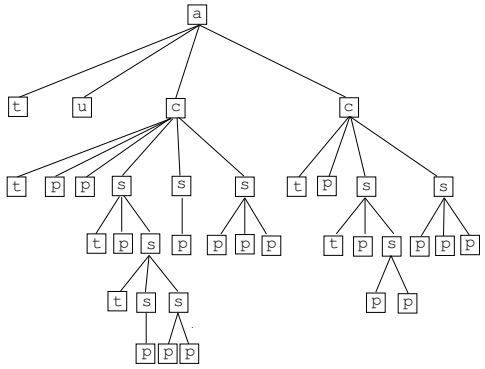
Definition 3 (Label-split Graph) Given an XML tree $T(V_t, E_t)$, a label-split graph $G(V_s, E_s)$ can be uniquely derived from a mapping $f : V_t \rightarrow V_s$ as follows:

- For every $u \in V_t$, there is a $f(u) \in V_s$.
- A node $u \in V_t$ is mapped to $f(u) \in V_s$ if and only if their labels are the same.
- For every pair of nodes $u, v \in V_t$, if $(u, v) \in E_t$, then there is a directed edge $(f(u), f(v)) \in E_s$.
- No other vertices and edges are present in $G(V_s, E_s)$. □

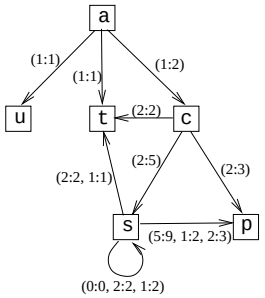
Figure 2(b), without the edge labels, depicts the label-split graph of the XML document shown in Figure 2(a). The label-split graph preserves the node label and edge relation in the XML tree, but not the cardinality of the relations.

3 Basic Synopsis Structures—XSEED kernel

Definition 4 (XSEED Kernel) The XSEED kernel for an XML tree is an edge-labeled label-split graph. Each edge $e = (u, v)$ in the graph is labeled with a vector of integer pairs $(p_0{:}c_0, p_1{:}c_1, \ldots, p_n{:}c_n)$. The i-th integer pair $(p_i{:}c_i)$, referred as $e[i]$, indicates that, at recursion level i, there are



(a) An example XML tree



(b) The XSEED kernel

Figure 2: An example XML tree and its XSEED kernel

a total of p_i elements mapped to the synopsis vertex u and c_i elements mapped to the synopsis vertex v. The p_i and c_i are called *parent-count* (referred as $e[i][\texttt{P\_CNT}]$) and *child-count* (referred as $e[i][\texttt{C\_CNT}]$), respectively. □

Example 2 The XSEED kernel shown in Figure 2(b) is constructed from the XML tree in Figure 2(a). In the XML tree, there is one a node and it has two c children. Thus, the edge (a, c) of XSEED kernel is labeled with integer pair (1:2). Out of these two c nodes in the XML tree, there are five s child nodes. Therefore, the edge (c, s) in the kernel is labeled with (2:5). Out of the five s nodes, two of them have one s child each (for a totally two s nodes having two s children). Since the two s child nodes are at recursion level 1, the integer pair at position 1 of the label of (s, s) is 2:2. Since the recursion level could not be 0 for any path having an edge $(\texttt{s}, \texttt{s})$, the integer pair at position 0 for this edge is 0:0. Furthermore, one of the two s nodes at recursion level 1 has two s children, which makes the integer pair at position 2 of the edge label (s, s) 1:2. □

The following observations of XSEED kernel are important for cardinality estimation algorithm given in Section 4.

Observation 1: For every path $(u_1, u_2, \ldots, u_n)$ in the XML tree, there is a corresponding path $(v_i, v_2, \ldots, v_n)$ in the kernel, where the label of v_i is the same as the label of u_i. Furthermore, for each edge (v_i, v_{i+1}), the number of integer pairs in the label is greater than the recursion level of the path $(u_1, \ldots, u_{i+1})$. For example, the path $(\texttt{a}, \texttt{c}, \texttt{s}, \texttt{s}, \texttt{s}, \texttt{p})$ in Figure 2(a) has a corresponding path $(\texttt{a}, \texttt{c}, \texttt{s}, \texttt{s}, \texttt{s}, \texttt{p})$ in the XSEED kernel in Figure 2(b). Moreover, the number of integer pairs in the label vector prevents a path with recursion level larger than 2, e.g., $(\texttt{a}, \texttt{c}, \texttt{s}, \texttt{s}, \texttt{s}, \texttt{s}, \texttt{p})$, from being derived from the synopsis.

Observation 2: For every node u in the XML tree, if its children have m distinct labels (not necessarily different from u's label), then the corresponding vertex v in the kernel has at least m out-edges, where the labels of the destination nodes match the labels of the children of u. This observation directly follows from the first observation. For example, the children of c nodes in the XML tree in Figure 2(a) have three different labels, thus the c vertex in the XSEED kernel in Figure 2(b) has three out-edges.

Observation 3: For any edge (u, v) in the kernel, the sum of the child-counts over all recursive levels i and greater is exactly the total number of elements that should be returned by the path expression $q//\texttt{u}//\texttt{v}$, whose recursion level is i and where q is a path expression that exists in the kernel. As an example, the number of results of expression $//\texttt{s}//\texttt{s}//\texttt{p}$ on the XML tree in Figure 2(a) is 5, which is exactly the sum of the child-counts of the label associated with edge $(\texttt{s}, \texttt{p})$ at recursion level 1 and 2.

The first observation guarantees that the synopsis preserves the complete information of the simple paths in the XML tree. However, some simple rooted paths that can be derived from the synopsis may not exist in the XML tree. That is, the kernel may contain false positives for a simple path query. The second observation guarantees that, for any branching path query, if it has a match in the XML tree, it also has a match in the synopsis. Again, false positives for branching path queries are also possible. The third observation connects the recursion levels in the data and in the query. This is useful in answering complex queries containing //-axes.

Kernel construction. The XSEED kernel can be generated while parsing the XML document. The pseudo-code in Algorithm 1 can be implemented using a SAX event-driven XML parser.

The $path\_stk$ in line 1 is a stack of vertices (and other information) representing the path while traversing in the kernel. Each stack entry ($\langle u, out\_edges \rangle$ in line 9) is a binary tuple, in which the first item indicates which vertex in the kernel corresponds to the current XML element, and the second item keeps a set of (e, l) pairs, in which e is an outedge of u, and l is the recursion level of the rooted path ended with the edge e. These pairs are used to increment

Algorithm 1 Constructing the XSEED Kernel

CONSTRUCT-KERNEL(S : Synopsis, X : XMLDoc)

```
1  path_stk ← empty stack;
2  rl_cnt ← empty counter stacks;
3  while the parser generates more event x from X
4    do if x is an opening tag event then
5         v ← GET-VERTEX(S, x);
6         if path_stk is empty then
7           rl_cnt.push(v);
8           path_stk .push(⟨v, ∅⟩);
9         else ⟨u, out_edges⟩ ← path_stk.pop();
10          e ← GET-EDGE(S, u, v);
11          l ← rl_cnt.push(v);
12          e[l][C_CNT] ← e[l][C_CNT] + 1;
13          out_edges ← out_edges ∪ (e, l);
14          path_stk.push(⟨u, out_edges⟩);
15          path_stk.push(⟨v, ∅⟩);
16     elseif x is a closing tag event then
17        ⟨v, out_edges⟩ ← path_stk.pop();
18        for each pair (e, l) ∈ out_edges
19          do e[l][P_CNT] ← e[l][P_CNT] + 1;
20        rl_cnt .pop(v);
```

the parent-count in the case of a close tag event (line 19).

The rl_cnt in line 2 is a data structure, we called "counter stacks", which efficiently calculates the recursion level of a path in expected $O(1)$. When traversing the XML tree, the vertices in the XSEED kernel are pushed onto and popped from rl_cnt as in a stack (line 7, 11, and 20). The key idea of the data structure to guarantee the efficiency is to partition the items into different stacks based on their number of occurrences. A hash table is kept to give the number of occurrences for any item pushed onto the counter stacks. Whenever an item is pushed onto the rl_cnt, the hash table is checked, the counter is incremented, and the item is pushed onto the corresponding stack maintained in the data structure. When an item is popped from rl_cnt, its occurrence is looked up in the hash table, popped from the corresponding stack, and the occurrence counter in the hash table is decremented. The recursion level of the whole path is indicated by the number of non-empty stacks minus 1. As an example, after pushing the sequence of `(a, b, b, c, c, b)` the data structure is shown in Figure 3. `a` and `b` are pushed onto counter stack 1 since their occurrences are 0 before inserting. When the second `b` is pushed, the counter of `b` is already 1, thus the new `b` is pushed to stack 2. Similarly, the following `c`, `c`, and `b` are pushed to the stack 1, 2 and 3, respectively. This data structure guarantees efficient calculation of recursion levels and is also used in the cardinality estimation algorithm introduced in Section 4.

The functions GET-VERTEX and GET-EDGE (lines 5 and 10) search the kernel and return the vertex or edge indicated by the parameters. If the vertex or edge is not in the graph then it is created.

Synopsis update. When the underlying XML document is updated, i.e., some elements are added or deleted, the kernel can be incrementally updated. The basic idea is to compute, for each subtree that is added or deleted, the kernel structure for the subtree. Then it can be added or subtracted from the original kernel using efficient graph merging or subtracting algorithm. Due to space limitations, the details are left to the full version of this paper [15].

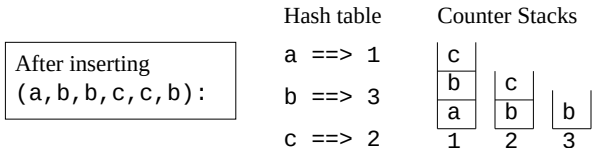


Figure 3: Counter stacks for efficient recursion level calculation

4 XSEED-based Cardinality Estimation

Before introducing the estimation algorithm, we define the following notions that are crucial to understand how cardinalities are estimated.

Definition 5 (Forward and Backward Selectivity) For any rooted path $p_{n+1} = (\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_n, \mathtt{v}_{n+1})$ in the XSEED kernel $G(V_s, E_s)$, denote $e_{(i,i+1)}$ as the edge (v_i, v_{i+1}), p_i as the sub-path $(\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_i)$, and r_i as the recursion level of p_i, then the *forward selectivity* ($fsel$) and *backward selectivity* ($bsel$) of path p_{n+1} are defined as:

$$fsel(p_{n+1}) = \frac{|/\mathtt{v}_1/\mathtt{v}_2/\cdots/\mathtt{v}_n/\mathtt{v}_{n+1}|}{S_{n+1}},$$

$$bsel(p_{n+1}) = \frac{|/\mathtt{v}_1/\mathtt{v}_2/\cdots/\mathtt{v}_n[\mathtt{v}_{n+1}]|}{|/\mathtt{v}_1/\mathtt{v}_2/\cdots/\mathtt{v}_n|},$$

where S_{n+1} is the sum of child-counts at the recursion level r_{n+1} over all in-edges of vertex $\mathtt{v}_{n+1}$, i.e.,

$$S_{n+1} = \sum e_{(i,n+1)}[r_{n+1}][\mathtt{C\_CNT}], \quad \forall e_{(i,n+1)} \in E_s. \quad \square$$

Intuitively, forward selectivity is the proportion of $\mathtt{v}_{n+1}$ nodes that are contributed by the path $(\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_n)$, and backward selectivity captures the proportion of $\mathtt{v}_n$ nodes under the path $(\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_{n-1})$ that have a child $\mathtt{v}_{n+1}$.

In Definition 5, if we assume that the probability of $\mathtt{v}_n$ having a child $\mathtt{v}_{n+1}$ is independent of $\mathtt{v}_n$'s ancestors, we can approximate $bsel$ as:

$$bsel(p_{n+1}) \approx \frac{e_{(n,n+1)}[r_{n+1}][\mathtt{P\_CNT}]}{S_n},$$

where S_n is defined similar to S_{n+1}. This approximated $bsel$ is the proportion of $\mathtt{v}_n$ under *any* path that have a child $\mathtt{v}_{n+1}$. Combining the definition and the approximation, the cardinality of the branching path $p_n[\mathtt{v}_{n+1}]$ can be estimated using the cardinality of the simple path p_n as follows:

$$|p_n[\mathtt{v}_{n+1}]| = |p_n| \times bsel(p_{n+1}) \approx |p_n| \times \frac{e_{(n,n+1)}[r_{n+1}][\mathtt{P\_CNT}]}{S_n}.$$

More generally, given a branching path expression $p = /\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}[\mathtt{v_{n+1}}]\cdots[\mathtt{v_{n+m}}]$, let $q = /\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}$, and assuming the $bsel$ of $q/\mathtt{v_{n+i}}$ is independent of the $bsel$ of $q/\mathtt{v_{n+j}}$ for any $i, j \in [1, m]$, then the cardinality of p is estimated as:

$$\begin{aligned} |q/[\mathtt{v_{n+1}}]\cdots[\mathtt{v_{n+m}}]| &\approx |q| \times bsel(q/\mathtt{v_{n+1}}) \times \cdots \\ &\quad \times bsel(q/\mathtt{v_{n+m}}) \\ &= |q| \times absel(p), \end{aligned}$$

where $absel(p)$ denotes the *aggregated* $bsel$s (products) of the rooted paths that end with a predicate query tree node. Since the $bsel$ of any simple path can be approximated using the XSEED kernel, the problem is reduced to how to estimate the cardinality of a simple path query.

For the simple path query $/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}/\mathtt{v_{n+1}}$ in Definition 5, if we again assume the probability of $\mathtt{v}_i$ having a child $\mathtt{v}_{i+1}$ is independent of $\mathtt{v}_i$'s ancestors, we can approximate the cardinality of $/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}/\mathtt{v_{n+1}}$ as:

$$|/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}/\mathtt{v_{n+1}}| \approx e_{(n,n+1)}[r_{n+1}][\mathtt{C\_CNT}] \times fsel(p_n).$$

Intuitively, the estimated cardinality of $/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}/\mathtt{v_{n+1}}$ is the number of $\mathtt{v}_{n+1}$ that are contributed by $\mathtt{v}_n$ times the proportion of $\mathtt{v}_n$ that are contributed by the path $/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_{n-1}}$. Based on this, $fsel$ can be estimated as:

$$fsel(p_{n+1}) \approx \frac{e_{(n,n+1)}[r_{n+1}][\mathtt{C\_CNT}] \times fsel(p_n)}{S_{n+1}}.$$

Since $fsel$ is defined recursively, we should calculate $fsel(p_{n+1})$ bottom-up, starting with $fsel(p_1)$, and then $fsel(p_2)$ and so on. At the same time, the estimated cardinalities of all sub-expressions are also calculated.

Example 3 Suppose we want to estimate the cardinality of query `/a/c/s/s/t` on the kernel shown in Figure 2(b). The following table shows the vertices in a path while traversing the kernel, the estimated cardinality, forward selectivity, and backward selectivity.

vertex	cardinality	*fsel*	*bsel*
a	1	1	1
c	2	1	1
s	5	1	1
s	2	1	0.4
t	1	1	0.5

The first row in the table refers to the path consisting of the single root node a; the second row refers to the path of (a, c) in the kernel, and so on. In particular the cardinality of the last row indicates the estimated cardinality of the path expression `/a/c/s/s/t`.

When traversing the first vertex a, we set the cardinality, $fsel$, and $bsel$ at their initial values of 1. When traversing the second vertex c, the cardinality is approximated as $|/\mathtt{a/c}| = e_{(\mathtt{a},\mathtt{c})}[0][\mathtt{C\_CNT}] \times fsel(\mathtt{a}) = 2 \times 1 = 2$, since the recursion level of path $(\mathtt{a}, \mathtt{c})$ is 0. $fsel(\mathtt{a}, \mathtt{c})$ is estimated as $\frac{|/\mathtt{a/c}|}{S_{(\mathtt{a},\mathtt{c})}} = \frac{2}{2} = 1$, where $S_{\mathtt{a},\mathtt{c}}$ is the sum of child-counts of all in-edges of c at recursion level specified by path $(\mathtt{a}, \mathtt{c})$. $bsel(\mathtt{a}, \mathtt{c})$ is estimated as $\frac{e_{(\mathtt{a},\mathtt{c})}[0][\mathtt{P\_CNT}]}{S_{(\mathtt{a})}} = \frac{1}{1} = 1$. When traversing a new vertex, the same calculations will take the results associated with the old vertices and the edge labels in the XSEED kernel as input, and produce the cardinality, $fsel$, and $bsel$ for the new vertex as output. □

The cardinality of a simple path query can be estimated as above; if we want to estimate the cardinality of a branching query or a complex path query consisting of //-axes and wildcards (*), we need to develop a matching algorithm to match the pattern tree specified by the expression to the kernel. In fact, the XSEED estimation algorithm defines a *traveler* (Algorithm 2) and a *matcher* (Algorithm 3). The matcher calls the traveler, through the function call NEXT-EVENT, to traverse the XSEED kernel in depth-first order. The rooted path is maintained while traveling. Whenever a vertex is visited, the traveler generates an *open event*, which includes the information about the label of the vertex, the DeweyID of this vertex, the estimated cardinality, the forward selectivity, and the backward selectivity of the current path. When finishing the visit of a vertex (due to some criterion introduced later), a *close event* is generated. In the end, an end-of-stream (EOS) event is generated when the whole graph is traversed. The matcher accepts this stream of events and maintains a set of internal states to match the tree pattern specified by the path expression.

Algorithm 2 is a simplified pseudo-code for the traveler algorithm. When traversing the graph, the algorithm maintains a global variable $pathTrace$, which is a stack of *footprint* (line 4). A footprint is a tuple including the current vertex, the estimated cardinality of the current path, the forward selectivity of the path, the backward selectivity of the path, the index of the child to be visited next, and the hash value for the current path. If the next vertex to be visited is the root of the synopsis, an open event with initial values are generated, otherwise the NEXT-EVENT function calls the VISIT-NEXT-CHILD function to move to the next vertex in depth-first order. The latter function calls the END-TRAVELING function to check whether the traversal should terminate (this is necessary for a synopsis containing cycles). Whether to stop the traversal is dependent on the estimated cardinality calculated in the EST function. In the EST function, the cardinality, forward selectivity, and backward selectivity are calculated as described earlier. If the estimated cardinality is less than or equal to some threshold ($CARD\_THRESHOLD$), the END-TRAVELING function returns `true`, otherwise it returns `false`. The OPEN-EVENT function accepts the vertex, the estimated cardinality, the forward selectivity, and

Algorithm 2 Synopsis Traveler

```
NEXT-EVENT()
1  if pathTrace is empty then
2      if no last event then                    ▷ current vertex is the root
3          h ← hash value of curV;
4          fp ← ⟨curV, 1, 1.0, 1.0, 0, h⟩;
5          pathTrace.push(fp);
6          evt ← OPEN-EVENT(v, card, fsel, bsel);
7      else evt ← EOS-EVENT();
8  else evt ← VISIT-NEXT-CHILD();

VISIT-NEXT-CHILD()
1  ⟨u, card, fsel, bsel, chdcnt, hsh⟩ ← pathTrace.top();
2  kids ← children of curV;
3  while size of kids is greater than chdcnt
4    do v ← kids[chdcnt];
5       if ¬END-TRAVELING(v, chdcnt) then
6           curV ← v;
7           ⟨v, card, fsel, bsel, hsh⟩ ← pathTrace.top();
8           evt ← OPEN-EVENT(v, card, fsel, bsel);
9           return evt;
10      increment chdcnt in pathTrace.top() by 1;
11 evt ← CLOSE-EVENT(u);
12 return evt;

END-TRAVELING(v : SynopsisVertex, chdCnt : int)
1  old_rl ← the recursion level of current path without v;
2  rl ← the recursion level of current path and v;
3  ⟨stop, card, fsel, bsel, n_h⟩ ← EST(v, rl, old_rl);
4  if stop then
5      return true;
6  fp ← ⟨v, card, fsel, bsel, 0, n_h⟩;
7  pathTrace.push(fp);
8  return false;

EST(v : SynopsisVertex, rl : int, old_rl : int)
1  ⟨u, card, fsel, bsel, chdcnt, hsh⟩ ← pathTrace.top();
2  e ← GET-EDGE(u, v);
3  if rl < e.label.size() then
4      n_card ← e[rl][C_CNT] * fsel;
5      sum_cCount ← TOTAL-CHILDREN(u, old_rl);
6      n_bsel ← e[rl][P_CNT]/ sum_cCount;
7  else n_card ← 0;
8  sum_cCount ← TOTAL-CHILDREN(v, rl);
9  n_fsel ← n_card / sum_cCount;
10 if n_card ≤ CARD_THRESHOLD then
11     stop ← true;
12 else stop ← false;
13 return ⟨stop, n_card, n_fsel, n_bsel, n_hsh⟩;
```

the backward selectivity as input, and generates an event including the input parameters and the DeweyID as output. The DeweyID of the event is maintained by the OPEN-EVENT and CLOSE-EVENT functions and is not shown in Algorithm 2.

If we treat the sequence of open and close events as open and close tags of XML elements, with the cardinality and selectivities as attributes, the traveler generates the following XML document from the XSEED kernel in Figure 2(b):

```
<a dID="1." card="1" fsel="1" bsel="1">
 <t dID="1.1." card="1" fsel="0.2" bsel="1"/>
```

Algorithm 3 Synopsis Matcher

```
CARD-EST(K : Kernel, qroot : QueryTreeNode)
1  frtSet ← {qroot};
2  frtStk .push(frtSet);
3  est ← 0;
4  evt ← NEXT-EVENT();
5  while evt is not an end-of-stream (EOS) event
6    do if evt is an open event then
7           frtSet ← frtStk .top();
8           new_fset ← ∅;
9           for each query tree node q ∈ frtSet
10            do if q.label = evt.label ∨ q.label = "*" then
11                   insert q's children into new_fset;
12                   insert evt into q's output queue;
13              if q.axis = "//" then
14                   insert q into new_fset;
15          frtStk.push(new_fset);
16      else if evt is a close event then
17              qroot.rmUnmatched();
18              if qroot.isTotalMatch() then
19                   est ← est +OUTPUT(evt.dID, qroot);
20              else if evt is matched to qroot then
21                   qroot.rmDescOfSelf(evt.dID);
22              frtStk.pop();
23      evt ← NEXT-EVENT();
24 return est;

OUTPUT(dID : DeweyID, qroot : QueryTreeNode)
1  Q ← rstQTN.outQ;
2  est ← 0;
3  absel ← AGGREGATED-BSEL(qroot);
4  for each evt ∈ Q
5    do est ← est + evt.card * absel;
6  Q.clear();
7  rstQTN . rmDescOfSelfSubTree(dID);
8  return est;
```

```
 <u dID="1.2." card="1" fsel="1" bsel="1"/>
 <c dID="1.3." card="2" fsel="1" bsel="1">
  <t dID="1.3.1." card="2" fsel="0.4" bsel="1"/>
  <p dID="1.3.2." card="3" fsel="0.25" bsel="1"/>
  <s dID="1.3.3." card="5" fsel="1" bsel="1">
   <t dID="1.3.3.1." card="2" fsel="0.4" bsel="0.4"/>
   <p dID="1.3.3.2." card="9" fsel="0.75" bsel="1"/>
   <s dID="1.3.3.3." card="2" fsel="1" bsel="0.4">
    <t dID="1.3.3.3.1." card="1" fsel="1" bsel="0.5"/>
    <p dID="1.3.3.3.2." card="2" fsel="1" bsel="0.5"/>
    <s dID="1.3.3.3.3." card="2" fsel="1" bsel="0.5">
     <p dID="1.3.3.3.3.1." card="3" fsel="1" bsel="1"/>
</s> </s> </s> </c> </a>
```

The tree corresponding to this XML document is dynamically generated and does not need to be stored. Since it captures all the simple paths that can be generated from the kernel, we call it the *expanded path tree* (EPT). In a highly recursive document (e.g., Treebank), the EPT could be even larger than the original XML document. This is because a single path with a high recursion level will result in generating other non-existing paths during the traversal. In this case, we need to set a higher $CARD\_THRESHOLD$ to limit the traversal. As demonstrated by our experiments, this heuristic greatly reduces the size of the EPT without causing much error.

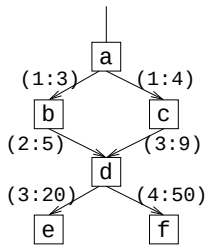


Figure 4: Example of ancestor independence assumption breaks

Algorithm 3 shows the pseudo-code for matching a query tree rooted at $qroot$ with the EPT generated from the kernel K. The algorithm maintains a stack of *frontier sets*, which is a set of query tree nodes (QTN) for the current path in the traversal. The QTNs in the frontier set are the candidates that can be matched with the incoming event. Initially the stack contains a frontier set consisting of the $qroot$ itself. Whenever a QTN in the frontier set is matched with an open event, the children of the QTN are inserted into a new frontier set (line 11). Meanwhile, the matched event is buffered into the output queue of the QTN as a candidate match (line 12). In addition to the children of the QTN that matches with the event, the new frontier set should also include all QTNs whose axis is "//" (line 14). After that, the new frontier set is ready to be pushed onto the stack for matching with the incoming open events if any.

Whenever a close event is seen, the matcher first cleans up the unmatched events in the output queue associated with each QTN (line 17). The call $qroot.rmUnmatched()$ checks the output queue of each QTN under $qroot$. If some buffered event does not have all its children QTN matched, these events are removed from the output queue. After the cleanup, if the top of the output queue of $qroot$ indicates a total match, the estimated cardinality is calculated (line 19). Otherwise, if $qroot$ is not a total match, the partial results should be removed from the $qroot$. Finally, the stack for the frontier set is popped indicating that the current frontier set is finished matching.

In the OUTPUT function, we need to sum the cardinalities of all the events cached in the resulting QTN. If there are predicates, the function AGGREGATED-BSEL calculates the product of backward selectivities of all events matched with predicate QTNs. After the summation, the output queue of the resulting QTN and all its descendant QTNs should be cleaned up.

5 Optimization for Accuracy—HET

The whole idea of the cardinality estimation using XSEED kernel is to first *compress* the XML tree, in the construction phase, into a graph structure that contains small amount of statistical annotations, and then, in the estimation phase, *decompress* the graph into a tree (EPT) based on the independence assumption (explained in detail later). The accuracy of cardinality estimation, therefore, depends upon how well the independence assumption hold on a particular XML document. Intuitively, the independence assumption refers to whether u having a child v is independent of the fact whether u having a particular parent/ancestor or other children. To capture the cases that are far from the independence assumption, we need to collect and keep additional information.

There are two cases where the estimation algorithm relies on the independence assumption. The first case happens when there are multiple in-edges and out-edges to a vertex v. The probability of v having a child, say w, is independent of which node is the parent of v. This assumption ignores the possible correlations between ancestor and descendants. This case is best illustrated by the following example.

Example 4 Given the XSEED kernel depicted in Figure 4, we want to estimate the cardinality of `b/d/e`. Since vertex `d` in the graph has two in-edges incident to `b` and `c`, the estimation algorithm assumes that the total number of `e` nodes (20) from `d` nodes are independent of whether `d`'s parents are `b` nodes or `c` nodes. Under this assumption, the cardinality of `b/d/e` is the cardinality of `d/e` times the proportion of `d` nodes that are contributed by `b` nodes, namely the forward selectivity of the path $p =$ `b/d/e`:

$$\begin{aligned} |p| &= |\texttt{d/e}| \times \mathit{fsel}(\texttt{b/d/e}) \\ &= e_{(\texttt{d},\texttt{e})}[0][\texttt{C\_CNT}] \times \frac{e_{(\texttt{b},\texttt{d})}[0][\texttt{C\_CNT}]}{e_{(\texttt{b},\texttt{d})}[0][\texttt{C\_CNT}] + e_{(\texttt{c},\texttt{d})}[0][\texttt{C\_CNT}]} \\ &= 20 \times \frac{5}{14} \approx 7.14. \end{aligned}$$

The estimate of $|\texttt{b/d/e}|$ is not 100% accurate, due to the ancestor independence assumption. □

The second type of independence assumption is in the case of branching path queries. If a vertex u in the kernel has two children v and w, the independence assumption assumes that the number of u nodes that have a child node v is independent of whether or not u also has a child w, ignoring the possible correlations between two siblings.

Example 5 Consider the XSEED kernel in Figure 4, and the path expression `b/d[f]/e`. Based on the independence assumption, the cardinality of the path expression `b/d[f]/e` is the cardinality of `b/d/e` times the proportion of `d` elements that have a `f` child, namely the backward selectivity of `f` in the path $p =$ `b/d/f`:

$$\begin{aligned} |p| &= |\texttt{b/d/e}| \times \mathit{bsel}(\texttt{b/d/f}) \\ &= |\texttt{b/d/e}| \times \frac{e_{(\texttt{d},\texttt{f})}[0][\texttt{P\_CNT}]}{e_{(\texttt{b},\texttt{d})}[0][\texttt{C\_CNT}] + e_{(\texttt{c},\texttt{d})}[0][\texttt{C\_CNT}]} \\ &= 20 \times \frac{5}{14} \times \frac{4}{14} \approx 2.04. \end{aligned}$$

Again, this estimate is not 100% accurate. □

A simple solution to this problem is to keep in what we call the *hyper-edge table* (HET), the actual cardinalities of the simple paths (e.g., `b/d/e`) and the "correlated backward selectivity" of the branching paths (e.g., the backward selectivity of `f` correlated with its sibling `e` under the path `b/d` in Example 5) when they induce large errors, so that we do not need to estimate it. In principle, the HET serves the same role as a histogram in relational database systems.

HET Construction. The HET can be pre-computed or populated by the optimizer through query feedback. While constructing the HET through query feedback is relatively straightforward, there are two issues related to pre-computation: (1) although we can estimate the cardinality using the XSEED kernel, we need an efficient way to evaluate the actual cardinalities to calculate the errors; and (2) the number of simple paths is usually small, but the number of branching paths is exponential in the number of simple paths. Thus, we need a heuristics to select a subset of the branching paths to evaluate.

To solve the first issue, we generate the path tree [1] while parsing the XML document (see Figure 1). The path tree captures the set of all possible simple paths in the XML tree. While constructing the path tree, we associate each node with the cardinality and backward selectivity of the rooted simple path leading to this node. Therefore, the actual cardinality of a simple path can be computed efficiently by traversing the path tree. To evaluate the actual cardinality of a branching path, we use the Next-of-Kin (NoK) operator [14], which performs tree pattern matching while scanning the data storage (see Figure 1) once, and returns the actual cardinality of a branching path.

To solve the second issue, we introduce two thresholds that effectively control the number of candidate branching paths. The first threshold is the maximum number of branching predicates MBP in the candidate path expressions. This threshold is the most effective one, since if we only consider the case where branching are restricted at the leaf level, the number of candidate branching paths could be $\sum_{i=1}^{n}\sum_{j=1}^{\min(f_i, MBP+1)}\binom{f_i}{j}$, where n is the number of nodes in the path tree, and f_i is the fan-out of node v_i in the path tree. MBP should be set to a very small number, say 2 (in which case it is called a *2BP HET*), in order to obtain reasonable number of candidate paths. To further reduce the candidates, we setup another threshold ($BSEL\_THRESHOLD$) for the backward selectivity of the path tree node to be examined, i.e., if $bsel(v) < BSEL\_THRESHOLD$, we evaluate the actual backward selectivity of the branching paths that have v as a predicate; otherwise v is omitted.

Accordingly, the construction of the hyper-edge table is straightforward: for every node v in the path tree, the estimated cardinality and actual cardinality are calculated. The path is put into a priority queue keyed by the absolute estimation error. If $bsel(v) < BSEL\_THRESHOLD$, all branching paths (only at the leaf level) with this node as one of the predicates are enumerated, and the paths are put into the priority queue. To limit the memory consumption of the hyper-edge table, we use a hashed integer (32 bits) to represent the string of path expression. When the hash function is reasonably good, the number of collisions is negligible. The hashed integer serves as a key to the actual cardinality and the correlated backward selectivity of the path expression. Table 1 is an example HET for the XSEED kernel in Figure 4, where actual hyper-edges rather than hashed values are shown.

hyper-edges	cardinality	correlated $bsel$
`/a/b/d/e`	14	0.1
`/a/c/d/e`	6	0.14
`/a/b/d/f`	21	0.25
`/a/c/d/f`	29	0.52
`d[e]/f`	4	0.35

Table 1: Hyper-Edge Table

We manage HET simply: we keep all the hyper-edges sorted in descending order of their errors on secondary storage and only keep the top k entries which have the largest errors in main memory to fill the memory budget. In our experiments, an 1BP hyper-edge table does not take a lot of disk space (less than 500,000 entries in the most complex Treebank data set and less than 1,000 entries for all the other tested data sets), but 2BP and 3BP could be very large for complex data sets.

Cardinality estimation. If the HET is available, we need to modify the traveler and matcher algorithms to exploit the extra information. The following changes apply to the 1BP HET. For 2BP and 3BP HET we need to change the AGGREGATED-BSEL as well, which we omit it here due to lack of space. In the traveler algorithm, we need to modify the lines 2 to 7 in function EST as follows:

```
1  if HET is available then
2      n_hsh ← incHash(hsh, v);
3      if n_hsh is in HET then
4          ⟨n_card, n_bsel⟩ ← HET.lookup(n_hsh);
5  else
6      e ← GET-EDGE(u, v);
7      if rl < e.label.size() then
8          n_card ← e[rl][C_CNT] ∗ fsel;
9          sum_cCount ← TOTAL-CHILDREN(u, old_rl);
10         n_bsel ← e[rl][P_CNT]/ sum_cCount;
11     else n_card ← 0;
```

This snippet of code guarantees that the actual cardinalities of simple paths are retrieved from the HET. The $incHash$ function incrementally computes the hash value of a path: given an old hash value for the path up to the new vertex and the new vertex to be added, the function returns the hash value for the path including the new vertex.

The matcher also needs to be modified to retrieve the correlated backward selectivity from the HET. The following should be inserted after line 11 in function CARD-EST:

```
1  if HET is available and q is a predicate QTN then
2      p ← q's parent QTN;
3      r ← p's non-predicate child QTN;
4      hsh ← incHash("p[q]/r");
5      if hsh is in HET then
6          ⟨card, bsel⟩ ← HET.lookup(hsh);
7          evt.bsel ← bsel;
```

In this code, the correlated backward selectivity of q and its non-predicate sibling QTN is checked. The parameter to the $incHash$ function is the string representation of the branching path $p[q]/r$.

6 Experimental results

We first evaluate the performance of the synopsis in terms of the following: (1) compression ratio of the synopsis on different types of data sets, and (2) accuracy of cardinality estimation for different types of queries.

To evaluate the combined effects of the above two properties, we compare accuracy with different space budgets against a state-of-the-art synopsis structure, TreeSketch [8]. TreeSketch is considered the best synopsis in terms of accuracy for branching path queries, and it subsumes XSketch [6] for structural-only summarization.

Another aspect of the experiments is to investigate the efficiency of the cost estimation function using the synopsis. We report the running time of the estimation algorithm for different types of queries. The ratios of the estimation times and the actual query processing times are also reported.

These experiments are performed on a dedicated machine with 2GHz Pentium 4 CPU and 1GB memory. The synopsis construction and cardinality estimation are implemented in C++. The TreeSketch code is obtained from its developers. The reported running times for the estimation algorithms are the averages of five runs.

6.1 Data sets and workload

We tested synthetic and real data sets with different characteristics: simple without recursion (DBLP[3], SwissProt[4], and TPC-H[4]), complex with small degree of recursion (XMark [9], NASA[4], and XBench TC/MD [12]), and complex with high degree of recursion (Treebank[4]). In this paper, we report DBLP, XMark10 and XMark100 (XMark with 10MB and 100MB of sizes, respectively), and Treebank.05 (randomly chosen 5% of Treebank) as well as the full Treebank as representative data sets for the three categories. The trends for the other data sets are similar. The basic statistics about the data sets are listed in Table 2.

We divided the workload into three categories: simple path (SP), branching path (BP), and complex path (CP). BP and CP queries represent the simplest branching queries with 0 or 1 predicate for each step. For each data set, we generate all possible SP queries, and $1,000$ random BP and CP queries. To test the effectiveness of HET with different MBP configurations, we also generate workload that have up to 2 branching predicates (2BP and 2CP) and 3 branching predicates (3BP and 3CP) in each step. The randomly generated queries are non-trivial. A sample CP query looks like `//regions/australia /item[shipping]/location`. The full test workload can be found in the full paper [15].

6.2 Construction time

For each data set, we measure the time to construct the kernel and HET separately. As described in Section 5, branching paths are estimated only for those path tree nodes whose backward selectivity is less than $BSEL\_THRESHOLD$. We use 0.1 as the threshold for all the data sets except Treebank, for which it is set to 0.001.

The construction time for XSEED and TreeSketch are given in Table 2. In this table, "DNF" indicates that the construction did not finish in 24 hours. The construction time for XSEED consists of the kernel construction time and the 1BP HET construction time (first and second part, respectively). The total construction time is the sum of these two numbers. As shown in the table, XSEED kernel construction time is negligible for all data sets, and the HET construction time is reasonable; overall they are much smaller than TreeSketch.

6.3 Accuracy of the synopsis

To evaluate the accuracy of XSEED synopsis, we again compare with TreeSketch on different types of queries (SP, BP, and CP). We calculated several error metrics but report here two[5], Root-Mean-Squared Error (RMSE) and Normalized RMSE (NRMSE), to evaluate the quality of the estimations. The RMSE is defined as $\sqrt{(\sum_{i=1}^{n}(e_i - a_i)^2)/n}$, where e_i and a_i are the estimated and actual result sizes, respectively, for the i-th query in the workload. The RMSE measures the average error over all queries in the workload. The NRMSE is adopted from [13] and is defined as RMSE/$\bar{a}$, where $\bar{a} = (\sum_{i=1}^{n} a_i)/n$. NRMSE is a measure of the average error per unit of accurate result size.

Since TreeSketch synopses cannot be constructed in our time limit on Treebank, we only list, in Table 3, the error metrics on the DBLP, XMark10, XMark100, and Treebank.05. These data sets represent all three data categories: simple, complex with small degree of recursion, and complex with high degree of recursion. The workload

[3] Available for download at `http://dblp.uni-trier.de/xml`

[4] Available for download at `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

[5] The Coefficient of Determination (R-squared) and Order Preserving Degree (OPD) are also calculated, but the values are very close to the perfect score for almost all datasets, so we omitted here.

Data sets	Data characteristics			XSEED kernel size	Synopsis construction time (mins)	
	total size	# of nodes	avg/max rec. level		XSEED	TreeSketch
DBLP	169 MB	4022548	0 / 1	2.8KB	$0.24 + 27$	11
XMark10	11 MB	167865	0.04 / 1	2.7KB	$0.01 + 0.27$	31
XMark100	116 MB	1666315	0.04 / 1	2.7KB	$0.1 + 2.7$	815
Treebank.05	3.4 MB	121332	1.3 / 8	24.2KB	$0.008 + 52$	839
Treebank	86 MB	2437666	1.3 / 10	72.7KB	$0.168 + 261$	DNF

Table 2: Characteristics of experimental data sets, their XSEED kernel size, and construction times

is the combined SP, BP, and CP queries. We tested both programs using 25KB and 50KB memory budgets, as well as testing XSEED kernel without HET, thus reducing the memory requirement. For the DBLP and XMark data sets, XSEED only uses 20KB and 25KB memory respectively for the total of kernel and HET, thus their error metrics on 25KB and 50KB are the same. Even without help from the HET, the XSEED kernel outperforms TreeSketch with 50KB memory budget on the XMark and Treebank.05 data sets. The reason is that the TreeSketch synopsis does not recognize recursions in the document, so even though it uses much more memory, the performance is not as good as the recursion-aware XSEED synopsis. When the document is not recursive, TreeSketch has better performance than the bare XSEED kernel. However, spending a small amount of memory on the HET greatly improves performance. The RMSE for XSEED with 25KB (i.e., a small HET) is almost half of the RMSE for TreeSketch with 50KB memory.

There is only one case—BP queries on DBLP (see Figure 5)—where TreeSketch outperforms XSEED even with the help of HET. In this case, XSEED errors are caused by the correlations between siblings that are not captured by the HET. For example, the query `/dblp/article[pages]/publisher` causes a large error on XSEED. The reason is that the backward selectivity (0.8) of `pages` under `/dblp/article` is above the default *BSEL_THRESHOLD* (0.1), so the hyper-edge `article[pages]/publisher` was omitted in the HET construction step, thus the correlation between `pages` and `publisher` is not captured. It is possible to use better heuristics to address this problem, although we have not investigated this in this paper.

We also tested the accuracy of different types of workload (1BP, 2BP and 3BP) on HETs with different MBP (maximum branching predicates) settings. Our observation is that 1BP HET is usually the best tradeoff between construction time and accuracy. Figure 6 shows the HET construction times (on the right y-axis) using different MBP settings for the DBLP dataset, and the estimation errors (on the left y-axis) for each setting on the 2BP workload. The error is reduced significantly (66%) going from no HET to 1BP HET, but the reduction in errror from 1BP HET to 2BP HET diminishes to 8%. On the other hand, the construction time of 2BP HET is about 10 times that of 1BP HET.

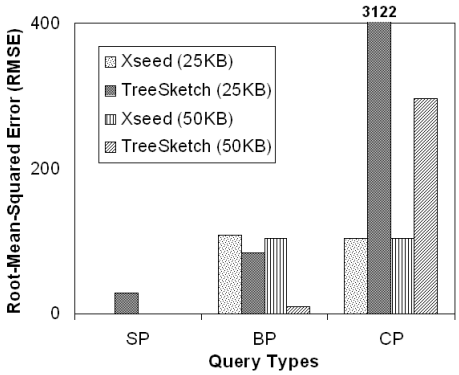


Figure 5: Estimation errors for different query types on DBLP

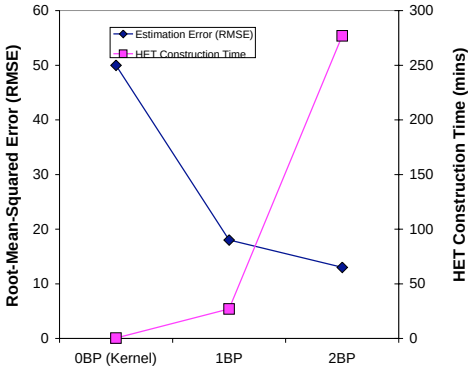


Figure 6: Different MBP settings on DBLP

6.4 Efficiency of cardinality estimation algorithm

To evaluate the efficiency of the cardinality estimation algorithm, we listed the ratio of the time spent on estimating the cardinality and the time spent on actually evaluating the path expression. The path expression evaluator we used is the NoK operator [14] extended to support //-axes.

The efficiency of the cardinality estimation algorithm depends on how many tree nodes there are in the expanded path tree (EPT) that can be generated from traversing the XSEED kernel. For DBLP, XMark10 and XMark100 data sets, the generated EPT is very small—0.0035%, 0.036%, and 0.05% of the original XML tree, respectively. As mentioned previously, the EPT could be large for highly recursive documents such as Treebank.05 and Treebank. To limit the size of EPT, as mentioned earlier, we establish a threshold for the estimated cardinality of the next vertex to visit. In these experiments, the threshold is set to 20 (i.e.,

Program settings		DBLP		XMark10		XMark100		Treebank.05	
		RMSE	NRMSE	RMSE	NRMSE	RMSE	NRMSE	RMSE	NRMSE
XSEED kernel		1960.5	15.4%	39.6	15.1%	276.15	5.06%	22.7	169%
25KB mem	XSEED	103	0.81%	3.737	1.43%	256.3258	4.71%	22.7	169%
	TreeSketch	221.5	1.67%	62.738	23.7%	638.1908	11.7%	229.5823	877.14%
50KB mem	XSEED	103	0.81%	3.737	1.43%	256.3258	4.71%	12.82	95.61%
	TreeSketch	203.09	1.59%	58.3946	22.09%	635.5347	11.65%	227.1157	867.71%

Table 3: Error metrics for XSEED andTreeSketch

if the estimated cardinality of the next vertex in depth-first order is less than 20, it will not be visited), and the ratio of EPT size to XML tree size is 6.9% and 5.5%.

The average ratios of the estimation time to the actual query running time on DBLP, XMark10, XMark100, Treebank.05, and Treebank are 0.018%, 0.57%, 0.0916%, 2%, and 1.5%. The ratios for XMark10 and XMark100 differ significantly because their XSEED kernels are very similar, but the size of the XML documents differs by a factor of 10.

7 Related work

There are many approaches dealing with cardinality estimation for path queries (e.g., [5, 4, 1, 6, 11, 2, 8, 10]). Some of them [5, 1, 11, 10] focus only on a subset of the possible path expressions, e.g., simple paths (linear chain of steps that are connected by /-axis) or linear paths containing //-axes. Moreover, none of them directly addresses recursive data sets, and only [5] and [10] support incremental maintenance of the synopsis structures.

TreeSketch [8], an extension to XSketch [6], can estimate the cardinality of branching path queries very accurately in many cases. However, it does not perform as well on recursive data sets. Also, due to the complexity of the construction process, TreeSketch is not practical for structure-rich data such as Treebank. XSEED has similarities to TreeSketch, but the major difference is that XSEED preserves structural information in two layers of granularity (kernel and HET); while TreeSketch tries to preserve this information in a complex and unified structure.

The hyper-edge table has been inspired by previous proposals [2, 10]. In [2], the actual statistics of previous queries are recorderd into a table and reused later. In [10], a Bloom Filter is used to compactly store cardinality information about simple paths. In this paper, we use one hash value for that purpose, since practice shows that a good hash function produces very few collisions for thousands of paths.

8 Conclusion

In this paper, we propose a compact synopsis structure to estimate the cardinalities of path queries. To the best of our knowledge, our approach is the first to support accurate estimation for all types of queries and data, incremental update of the synopsis when the underlying XML document is changed, dynamic reconfiguration of the synopsis according to the memory budget, and the ability to exploit query feedback. The simplicity and flexibility of XSEED make it well suited for implementation in a real DBMS optimizer.

Acknowledgement: We thank Neoklis Polyzotis and Minos Garofalakis for providing us with the TreeSketch code.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, 2001.

[2] A. Aboulnaga and J. F. Naughton. Building XML Statistics for the Hidden Web. In *CIKM*, 2003.

[3] D. Chamberlin. XQuery: An XML Query Language. *IBM Systems Journal*, 41(40), 2002.

[4] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *ICDE*, 2001.

[5] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, 1997.

[6] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph Structured XML Databases. In *SIGMOD*, 2002.

[7] N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *VLDB*, 2002.

[8] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML Query Answers. In *SIGMOD*, 2004.

[9] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Tech. Report INS-R0103, CWI, 2001.

[10] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *VLDB*, 2004.

[11] Y. Wu, J. M. Patel, and H. Jagadish. Estimating Answer Sizes for XML Queries. In *EDBT*, 2002.

[12] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *ICDE*, 2004.

[13] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *VLDB*, 2005.

[14] N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *ICDE*, 2004.

[15] N. Zhang, M. T. Özsu, A. Aboulnaga, and I. F. Ilyas. XSeed: Accurate and Fast Cardinality Estimation for XPath Queries. Technical report CS-2005-22, University of Waterloo, 2005.