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

Cardinality estimation is a crucial part of a cost-based optimizer. Many research efforts have been focused on XML synopsis structures of path queries for cardinality estimation in recent years. In ideal situations, a synopsis should provide accurate estimates for different types of queries over a wide variety of data sets, consume a small amount of memory while being able to adjust as memory budgets change, and be easy to construct and update. None of the existing synopsis proposals satisfy all of the above requirements. In this paper, we propose a novel synopsis, XSEED, *that is accurate, robust, efficient, and adaptive to memory budgets. We construct an* XSEED *structure starting from a very small kernel, then incrementally update information of the synopsis. With such an incremental construction, a synopsis structure can be dynamically configured to accommodate different memory budgets. It can also handle updates to underlying XML documents, and be self-tuning by incorporating query feedback. Cardinality estimation based on* XSEED *can be performed very efficiently and accurately, with our techniques of small footprint and novel recursion handling. Extensive experiments*[1] *on both synthetic and real data sets are conducted, and our results show that even with less memory, the accuracy of* XSEED *could achieve 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

XML is rapidly becoming a key technology for data exchange and data integration on the Internet. This has increased the need for efficient execution of XML queries. Cost-based optimization 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., [15]). Therefore, accurate cardinality estimation is crucial for a cost-based optimizer to be able to make the right decision.

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 [6, 9]. In this paper, we focus on the structural part of the this problem and propose a novel synopsis structure, called XSEED[2] 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 [7, 4, 6, 10]. These approaches, usually, first summarize an XML document into a compact graph structure called *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

[1]The full set of testing queries can be found at `http://db.uwaterloo.ca/~ddbms/publications/xml/XSeed_workload.tgz`.

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

this paper, we follow this general idea but develop a solution that meets multiple criteria. That is, we consider not only the accuracy of the estimations, but also 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 consider all these dimensions. For example, TreeSketch [10], a synopsis for cardinality estimation, focuses on the accuracy of the cardinality estimation. It starts off by building up a bi-simulation 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 one day to construct the synopsis for the 100MB XMark [11] data set 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 lies in the data, and can be constructed easily. The purpose of the small kernel is not to make it optimal in terms of accuracy. Rather, 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 captures recursions that exist in the XML documents. Recursive documents usually represent the most difficult cases for path query processing and cardinality estimation. None of the existing approaches has studied recursive documents and the effects of recursion over the accuracy of cardinality estimation. To the best of our knowledge, this paper 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 even 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

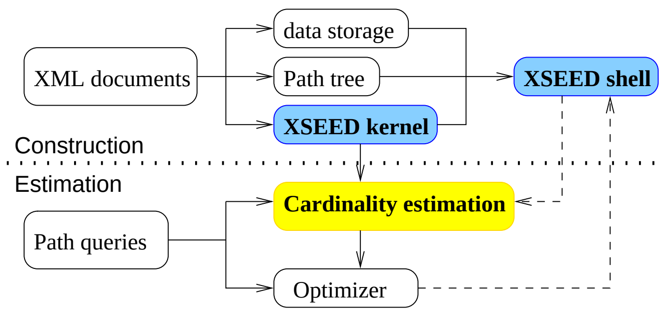


Figure 1: Cardinality estimation process using XSEED

inevitably results in greater estimation errors in some cases. To remedy the accuracy deficiency for these cases, we introduce another layer of information, called *shell*, on top of the kernel. The shell 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 shell can be pre-computed in a similar or shorter time than other synopses, or it can be dynamically fed by a self-tuning optimizer if the query feedback mechanism is enabled. This information can be easily maintained, i.e., it can be added/deleted to/from the synopsis whenever the memory budget changes. When the underlying XML data is changed, the optimizer can choose to update the information eagerly or lazily. In this way, XSEED enjoys better accuracy and adaptivity as well.

Figure 1 depicts the process of constructing and maintaining the XSEED kernel and shell, and utilizing them to predict the cardinality. In the construction phase, the XML document is first parsed to generate the NoK XML storage structure [16], the path tree [1], and the XSEED kernel. The shell is constructed based on these three data structures if it is opt to 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 XSEED shell. After the execution, the optimizer may feedback the actual cardinality of the query to the XSEED shell, which might results in an update of the data structure.

Our contributions are the following:

- We design a novel synopsis structure, called XSEED, with the following properties:
  - The tiny 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 shell of XSEED provides additional informa-

tion of 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 model.
- 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 for the rest of the paper. In Section 3, we introduce the main idea of constructing the basic part of XSEED kernel from an XML document. In Section 4, we present the algorithm for estimating cardinality on the XSEED kernel, and analysis its complexity. In Section 5, we introduce the cases where the XSEED kernel makes estimation errors and how to acquire additional knowledge to compensate them. In Section 6, we report the experimental results on the accuracy of the synopsis on different data sets and workloads under different memory budgets. The running time of the estimation algorithm is also reported. We compare our approach with related work in Section 7. Finally, we conclude in Section 8 by a summary and final remarks.

2 Preliminaries

In this section, we give the basic definitions related to the XSEED synopsis structure. Due to space limitations, we omit the formal definitions of XML data model and path expressions[3]. Rather 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 $(u_1, u_2, \ldots, u_n)$ 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.

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 $\{\mathtt{a}, \mathtt{t}, \mathtt{u}, \mathtt{c}, \mathtt{p}, \mathtt{s}\}$:

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

```
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 throughout the rest of the paper. □

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. We define the recursion levels with respect to a path, node, and document as follows:

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. □

As an example, the recursion level of the path $(\mathtt{a}, \mathtt{c}, \mathtt{s}, \mathtt{p})$ in Figure 2(a) is 0 since there is no duplicated nodes 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.

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 two or more NodeTests 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 one could be matched to both the NodeTests. It is straightforward to see that 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 and recursive queries are the hardest documents to summarize and the hardest queries to evaluate and estimate.

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 many possible levels of abstractions depending on the desired tradeoff between the space constraints and information preservation. For example, we can choose to only preserve the basic properties such as the node label and edge relation, or we can preserve the cardinality of the edge relation as well. There are a number of proposed structures. In the following, we only introduce the label-split graph [8], which is the basis of XSEED.

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 is the parent of v in 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. The XSEED, as described in the following section, preserves more information.

3 Basic Synopsis Structures—XSEED kernel

In this section, we first give an overview of the basic synopsis structure–the XSEED kernel. An efficient algorithm that constructs the kernel from parsing the XML document is also presented. We then introduce the cardinality estimation process using the kernel.

3.1 Overview

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 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 $e[i]$[P_CNT]) and *child-count* (referred as $e[i]$[C_CNT]), respectively. □

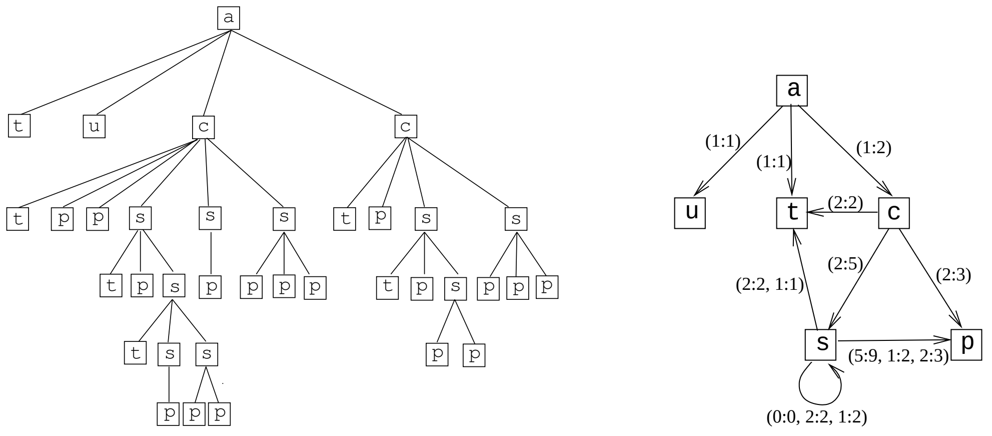
Example 2 The XSEED kernel shown in Figure 2(b) is constructed from the XML tree in Figure 2(a). In the XML tree, there is totally one $\boxed{\texttt{a}}$ node and it has two $\boxed{\texttt{c}}$ children. Correspondingly in the XSEED kernel, the edge of (a, c) is labeled with integer pair (1:2). Out of these two $\boxed{\texttt{c}}$ nodes in the XML tree, there are five $\boxed{\texttt{s}}$ child nodes. Therefore, the edge (c, s) in the kernel is labeled with (2:5). Out of the five $\boxed{\texttt{s}}$ nodes, two of them have one $\boxed{\texttt{s}}$ node each (for a totally two $\boxed{\texttt{s}}$ nodes having two $\boxed{\texttt{s}}$ children). Since the two $\boxed{\texttt{s}}$ 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 $(\texttt{s}, \texttt{s})$, therefore the integer pair at position 0 for the edge(s, s) is 0:0. Furthermore, one of the two $\boxed{\texttt{s}}$ nodes at recursion level 1 has two $\boxed{\texttt{s}}$ children, which makes the integer pair at position 2 of the edge label (s, s) 1:2. □

With this simple structure, the algorithm introduced in Section 4 can estimate the cardinality of all types of queries, including recursive queries, and queries containing wildcards (*). This algorithm is based on the following observations.

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). Furthermore, the number of integer pairs in the label vector prevents a path with recursion level larger than 2, e.g., $(\texttt{a}, \texttt{c}, \texttt{s}, \texttt{s}, \texttt{s}, \texttt{s}, \texttt{p})$, being derived from the synopsis.

Observation 2: For every node u in the XML tree, if its children have m distinct labels (not necessarily be 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 $\boxed{\texttt{c}}$ nodes in the XML tree in Figure 2(a) have three different labels, thus the $\boxed{\texttt{c}}$ 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 level i and greater is exactly the total number of elements that should be returned by the path expression p`//u//v`, where p is a path expression and the recursion level of p`//u//v` is i. As an example, the number of results of expression `//s//s//p` on the XML tree in Figure 2(a) is 5,



(a) An example XML tree (b) The XSEED kernel

Figure 2: An example XML tree and its XSEED synopsis kernel

which is exactly the sum of the child-counts of the label associated with edge (s, 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.

3.2 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 of the algorithm is a stack of vertices (and other information) representing the path while traversing in the kernel. Each stack entry ($\langle u, vst\_set\rangle$ in line 8) is a 2-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 the out-edges of this vertex that have been explored in the XML subtree. The set of out-edges in the tuple is used to increment the parent-count of these edges in the case of a close tag event (line 19).

The rl_cnt in line 2 is a "counter stacks" data structure to efficiently calculate the recursion level of a path. Since the vertices in the path are pushed and popped as in a stack,

Algorithm 1 Constructing XSEED Kernel

```
CONSTRUCT-KERNEL($S$ : Synopsis, $X$ : XMLDoc)
1  $path\_stk \leftarrow$ empty stack;
2  $rl\_cnt \leftarrow$ initialize to empty;
3  while the parser generates more events from $X$
4    do $x \leftarrow$ next event from $X$;
5       if $x$ is an opening tag event
6           $v \leftarrow$ GET-VERTEX$(S, x)$;
7           if $path\_stk \neq \emptyset$
8               $\langle u, vst\_set\rangle \leftarrow path\_stk.pop()$;
9               $e \leftarrow$ GET-EDGE$(S, u, v)$;
10              $vst\_set \leftarrow vst\_set \cup \{e\}$;
11              $path\_stk.push(\langle u, vst\_set\rangle)$;
12              $l \leftarrow rl\_cnt.push(v)$;
13              $e[l][\text{C\_CNT}] \leftarrow e[l][\text{C\_CNT}] + 1$;
14              $path\_stk.push(\langle v, \emptyset\rangle)$;
15          else $path\_stk .push(\langle v, \emptyset\rangle)$;
16      elseif $x$ is a closing tag event
17          $\langle v, vst\_set\rangle \leftarrow path\_stk.pop()$;
18          for each edge $e \in vst\_set$
19            do $e[l][\text{P\_CNT}] \leftarrow e[l][\text{P\_CNT}] + 1$;
20          $rl\_cnt .pop(v)$;
```

the recursion level can be computed in expected $O(1)$ every time a new item is pushed onto (line 12) or popped out (line 20) from the data structure . The key idea 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 (that is why the complexity is expected $O(1)$, not $O(1)$). Whenever an item is pushed onto the rl_cnt, the hash table is checked, the counter is incremented, and the item is pushed

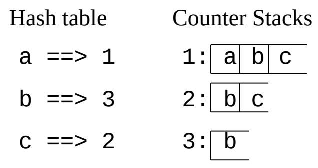


Figure 3: Counter stacks for efficient recursion level calculation

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 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. When pushing a and b into the counter stacks, they are pushed to 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 of great importance in the cardinality estimation algorithm introduced in Section 4.

The functions GET-VERTEX and GET-EDGE (lines 6 and 9) 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.

3.3 Synopsis update

When the underlying XML document is updated, i.e., some elements are added or deleted, the kernel can incrementally be updated accordingly. The basic idea is to for each of the subtree that is added or deleted, compute the kernel structure for the subtree, then it can be added or subtracted from the original kernel using the efficient graph merging or subtracting algorithm [5].

When delete a subtree, we can construct a new kernel for the subtree. Furthermore, we still need to know which vertex in the original kernel corresponds to the parent of the root of the new kernel. Suppose the the new kernel and the original kernel are k' and k, respectively, the root of k' is r' and its parent in k is p, then subtraction of k' from k takes two steps: 1) get the label of the edge (p, r'), subtract 1 from the child-count of the integer pair at the recursion level of r'. If the child-count is 0, then set the parent-count to 0 as well, and adjust the size of the vector if necessary. 2) for each edge e' in k', locate the same edge e in k, subtract the parent-count and child-count in e' from e at each recursion level. The vector size should also be adjusted accordingly, and if the size of a vector is 0, the edge should be deleted. When adding a subtree to the XML tree, the way to incrementally update the kernel is similar. The only difference is to change the minus operation to plus, and adding edges if necessary.

The hyper-edge table can also be incrementally updated when a subtree is added/deleted to/from the XML tree. We only need to (re-)compute the errors related to the paths that are updated by the new kernel. The old entries in the table are deleted and the new entries with the new errors are added.

4 Cardinality Estimation based on XSEED Kernel

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 edge (v_i, v_{i+1}), the sub-path $(\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_i})$ as p_i, and the recursion level of p_i as r_i, then the *forward selectivity* and *backward selectivity* of the path p_{n+1} is defined as:

$$\begin{aligned} fsel(p_{n+1}) &= \frac{|/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}/\mathtt{v_{n+1}}|}{S_{n+1}} \\ bsel(p_{n+1}) &= \frac{|/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}[\mathtt{v_{n+1}}]|}{|/\mathtt{v_1}/\mathtt{v_2}/\cdots/\mathtt{v_n}|} \end{aligned}$$

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

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

□

Intuitively, the forward selectivity is the proportion of $\mathtt{v}_{n+1}$ that are contributed by the path $(\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_n})$. The backward selectivity captures the proportion of $\mathtt{v}_n$ under the path $(\mathtt{v_1}, \mathtt{v_2}, \ldots, \mathtt{v_{n-1}})$ that have a child $\mathtt{v}_{n+1}$. Both notions are defined using the cardinalities of path expressions, which we shall introduce how to estimate them next.

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

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

where S_n is defined similarly as S_{n+1} in Definition 5. 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:

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

More generally, given a 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 ended 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:

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

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 approximation, the $fsel$ can be estimated as:

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

Since the $fsel$ is defined recursively, we should calculate $fsel(p_{n+1})$ from bottom-up. That is, we calculate the $fsel(p_1)$ first, and then use it to calculate $fsel(p_2)$, and so on. At the same time, the estimated cardinalities of all subexpressions are also calculated. We illustrate this process in the following example.

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.

The first row in the table means the path consisting of the single root node $\boxed{\mathtt{a}}$; the second row means 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`.

vertex	cardinality	*fsel*	*bsel*
$\boxed{\mathtt{a}}$	1	1	1
$\boxed{\mathtt{c}}$	2	1	1
$\boxed{\mathtt{s}}$	5	1	1
$\boxed{\mathtt{s}}$	2	1	0.4
$\boxed{\mathtt{t}}$	1	1	0.5

When traversing the first vertex $\boxed{\mathtt{a}}$, we set the cardinality, *fsel*, and $bsel$ as their initial values 1. When traversing the second vertex $\boxed{\mathtt{c}}$, the cardinality is approximated as $|/\mathtt{a}/\mathtt{c}| = e_{(\mathtt{a},\mathtt{c})}[0][\mathrm{C\_CNT}] \times fsel(\mathtt{a}) = 2 \times 1 = 2$, since the recursion level of path $(\mathtt{a}, \mathtt{c})$ is 0. The $fsel(\mathtt{a}, \mathtt{c})$ is estimated as $\frac{|/\mathtt{a}/\mathtt{c}|}{S_{(\mathtt{a},\mathtt{c})}} = \frac{2}{2} = 1$, where $S_{\mathtt{a},\mathtt{c}}$ is the sum of child-counts of all in-edges of $\mathtt{c}$ at recursion level specified by path $(\mathtt{a}, \mathtt{c})$. The $bsel(\mathtt{a}, \mathtt{c})$ is estimated as $\frac{e_{(\mathtt{a},\mathtt{c})}[0][\mathrm{P\_CNT}]}{S_{(\mathtt{a})}} = \frac{1}{1} = 1$. 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 above description present a way to estimate the cardinality of a simple path query. 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. 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. At last, an end-of-stream (EOS) event is generated when the whole graph is traversed. The matcher accepts these stream of events and maintain 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 visit is the root of the synopsis, an open event with initial values are generated, otherwise the NEXT-EVENT function calls the TRAVERSE-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

Algorithm 2 Synopsis Traveler

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

TRAVERSE-NEXT-CHILD()
1  $\langle u, card, fsel, bsel, chdcnt, hsh\rangle \leftarrow pathTrace.top()$;
2  $kids \leftarrow$ children of $curV$;
3  while $kids.size() > chdcnt$
4    do $v \leftarrow kids[chdcnt]$;
5       if ¬END-TRAVELING($v$, $chdcnt$)
6           $curV \leftarrow v$;
7           $\langle v, card, fsel, bsel, hsh\rangle \leftarrow pathTrace.top()$;
8           $evt \leftarrow$ OPEN-EVENT($v$, $card$, $fsel$, $bsel$);
9           return $evt$;
10      increment $chdcnt$ in $pathTrace.top()$ by 1;
11 $evt \leftarrow$ CLOSE-EVENT($u$);
12 return $evt$;

END-TRAVELING($v$ : SynopsisVertex, $chdCnt$ : int)
1  $old\_rl \leftarrow$ the recursion level of current path without $v$;
2  $rl \leftarrow$ the recursion level of current path and $v$;
3  $\langle stop, card, fsel, bsel, n\_h\rangle \leftarrow$ EST($v$, $rl$, $old\_rl$);
4  if $stop$
5      return true;
6  $fp \leftarrow \langle v, card, fsel, bsel, 0, n\_h\rangle$;
7  $pathTrace.push(fp)$;
8  return false;

EST($v$ : SynopsisVertex, $rl$ : int, $old\_rl$ : int)
1  $\langle u, card, fsel, bsel, chdcnt, hsh\rangle \leftarrow pathTrace.top()$;
2  $e \leftarrow$ GET-EDGE($u$, $v$);
3  if $rl < e.label.size()$
4      $n\_card \leftarrow e[rl][\text{C\_CNT}] * fsel$;
5      $sum\_cCount \leftarrow$ TOTAL-CHILDREN($u$, $old\_rl$);
6      $n\_bsel \leftarrow e[rl][\text{P\_CNT}] / sum\_cCount$;
7  else $n\_card \leftarrow 0$;
8  $sum\_cCount \leftarrow$ TOTAL-CHILDREN($v$, $rl$);
9  $n\_fsel \leftarrow n\_card \,/\, sum\_cCount$;
10 if $n\_card < CARD\_THRESHOLD$
11     $stop \leftarrow$ true;
12 else $stop \leftarrow$ false;
13     return $\langle stop, n\_card, n\_fsel, n\_bsel, n\_hsh\rangle$;
```

traverse reaches an end (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 some threshold ($CARD\_THRESHOLD$), the END-TRAVELING function returns `true`, otherwise `false`. The OPEN-EVENT function accepts the vertex, the estimated cardinality, the forward selectivity, and the backward selectivity as input, and generate an event including the input parameters and the DeweyID as output. The DeweyID in 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 attributed with cardinality and selectivities, 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"/>
 <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 *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 high recursion level will validate 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 heuristics greatly reduces the size of the EPT without causing much error.

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 set*, 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 match with the event, the new frontier set should also include all QTN's 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.

Algorithm 3 Synopsis Matcher

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

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

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 up 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 predicate QTNs. After the the summation, the output queue of the resulting QTN should be cleaned up and the output queues of all its descendant QTNs are also cleaned up.

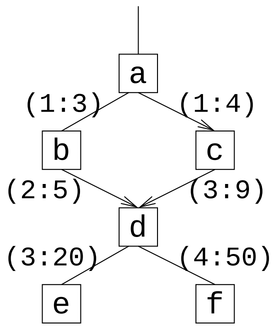


Figure 4: Example synopsis structure

5 Optimization for Accuracy—XSEED Shell

In this section, we introduce the data structures that keep auxiliary information to improve the accuracy of cardinality estimation. The construction of this data structure and the modification of the estimation algorithm to exploit this extra information are also introduced.

5.1 Overview

The accuracy of cardinality estimation depends upon how well the independence assumption (built into the XSEED kernel) holds on a particular XML document. Here, the independence assumption (explained in detail later) refers to that whether u has a child v is independent of whether u has 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 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 the vertex $\boxed{\texttt{d}}$ in the graph has two in-edges incident to $\boxed{\texttt{b}}$ and $\boxed{\texttt{c}}$, we have to assume that the total number of $\boxed{\texttt{e}}$'s (20) from $\boxed{\texttt{d}}$'s are independent from whether $\boxed{\texttt{d}}$'s parents are $\boxed{\texttt{b}}$'s or $\boxed{\texttt{c}}$'s. Under this assumption, the cardinality of `b/d/e` is the cardinality of `d/e` times the proportion of $\boxed{\texttt{d}}$ elements that are contributed by $\boxed{\texttt{b}}$ elements, namely the forward selectivity of e in the path b/d/e:

$$\begin{aligned}|\texttt{b/d/e}| &= |\texttt{d/e}| \times \mathit{fsel}(\texttt{b/d/e}) \\ &= |\texttt{d/e}| \times \frac{|\texttt{b/d}|}{|\texttt{b/d}| + |\texttt{c/d}|}.\end{aligned}$$

In the above formula, the cardinality of a long simple path is broken down into the cardinalities of short paths—binary edges. In fact, a simple path of arbitrary length can be rewritten to a formula that consists of only binary edges based on the independence assumption. Since the cardinalities of the binary edges are the child-counts of the edge labels, the cardinality of path /b/d/e is:

$$|\texttt{b/d/e}| = 20 \times \frac{5}{5+9} \approx 7.14.$$

The fact that the estimate of |b/d/e| being a real number rather than an integer indicates the estimate is not 100% accurate. The only reason is the independence assumption mentioned above. □

The second case that relies on the independence assumption is 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's that have a child v is independent of whether or not u has a child w. This assumption ignores 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 $\boxed{\texttt{d}}$ elements that have a $\boxed{\texttt{f}}$ child, namely the backward selectivity of f in the path b/d/f:

$$\begin{aligned}|\texttt{b/d[f]/e}| &= |\texttt{b/d/e}| \times \mathit{bsel}(\texttt{b/d/f}) \\ &= |\texttt{b/d/e}| \times \frac{|\texttt{d[f]}|}{|\texttt{b/d}| + |\texttt{c/d}|} \\ &= 20 \times \frac{5}{14} \times \frac{4}{5+9} \approx 2.04.\end{aligned}$$

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) or 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. For the case of branching paths, the reasons that we keep the correlated backward selectivity instead of cardinality is that, firstly, the cardinality can be accurately calculated by the correlated backward selectivity, and, secondly, the correlated backward selectivity can be used in branching paths with multiple predicates.

5.2 Construction

The HET can be pre-computed or added by the optimizer through query feedback. While constructing the HET through query feedback is relatively straightforward, there are two issues related to the pre-computation: (1) although we can estimate the cardinality using the XSEED kernel (the estimation algorithm is introduced in Section 4), we need an efficient way to evaluate the actual cardinalities to calculate the errors; and (2) the number of simple paths is usually reasonably small, but the number of branching paths is exponential in the number of simple paths. Therefore, we need a heuristics to select a subset of 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 determined by 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 [16], which performs tree pattern matching while scanning the data storage (see Figure 1) once, and return the actual cardinality of a branching path.

To solve the second issue, we limit the branching paths to have only one predicate. This will reduce the number of branching paths from exponential to the size of the path tree to $\sum_{i=1}^{n} \binom{f_i}{2}$, 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. In the worst case, this is still quadratic in the size of the path tree. To further reduce the number of branching paths to be examined, we can use various heuristics. For example, we can setup a threshold for the backward selectivity of the path tree node to be examined. That is, when traversing the path tree, if the backward selectivity of the node is less than the threshold, we evaluate the actual backward selectivity of the branching paths that have this node as a predicate; otherwise they are omitted.

Based on the description above, the construction of the hyper-edge table is straightforward: for every node in the path tree, the estimated cardinality and actual cardinality are calculated. The path is put into a priority queue keyed by the estimation error. Also if the backward selectivity of the path is less than the threshold, the branching paths with this node as predicate are calculated, and the paths are put into the priority queue. To limit the memory consumption of the hyper-edge table, we use a hashed integer instead of the string of path expression. When the hash function is reasonably good, the number of of collisions is negligible. The hashed integer serves as a key to the "value", actual cardinality and the correlated backward selectivity, of the table. Table 1 is an example HET for the XSEED kernel in Figure 4. In this table, for the purpose of presentation, the actual simple or branching paths (i.e., hyper-edges) are given instead of their hashed values (column 1).

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 use a simple way to manage the hyper-edge table: 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. These top k entries should be large enough to fill the memory budget. In practice, the hyper-edge table is not likely to take a lot of disk space, since there are less than 500,000 hyper-edges in the most complex data set (Treebank) that we tested, and less than 1,000 entries for all the other tested data sets. Consequently, the hyper-edge table can be maintained dynamically to reflect changing memory budget: when the memory budget decreases, the only thing we need to do is to discard some number of entries with the smallest errors from main memory; when the memory budget increases, we just need to bring more entries with the largest errors to the main memory.

5.3 Cardinality estimation

If the hyper-edge table (HET) is available, we need to modify the traveler and matcher algorithms to exploit the extra information. In the traveler algorithm, we need to modify the lines 2 to 7 in function EST as the following:

```
1  **if** HET is available
2        $n\_hsh \leftarrow incHash(hsh, v)$;
3        **if** $n\_hsh$ is in HET
4              $\langle n\_card, n\_bsel\rangle \leftarrow HET.lookup(n\_hsh)$;
5  **else**
6        $e \leftarrow$ GET-EDGE$(u, v)$;
7        **if** $rl < e.label.size()$
8              $n\_card \leftarrow e[rl][\mathrm{C\_CNT}] * fsel$;
9              $sum\_cCount \leftarrow$ TOTAL-CHILDREN$(u, old\_rl)$;
10             $n\_bsel \leftarrow e[rl][\mathrm{P\_CNT}] / sum\_cCount$;
11       **else** $n\_card \leftarrow 0$;
```

If the HET is available, This snippet of code guarantees that the actual cardinalities of simple paths are retrieved from the HET. The $incHash$ function incrementally compute the hash value of a path: given an old hash value for the path up until the new vertex and the new vertex to be added in the path, the function return 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 snippet of code should be inserted after line 11 in function CARD-EST:

```
1  **if** HET is available and $q$ is a predicate QTN
2        $p \leftarrow q$'s parent QTN;
3        $r \leftarrow p$'s non-predicate child QTN;
4        $hsh \leftarrow incHash(\text{“}p[q]/r\text{”})$;
5        **if** $hsh$ is in HET
6              $\langle card, bsel\rangle \leftarrow HET.lookup(hsh)$;
7              $evt\,.\,bsel \leftarrow bsel$;
```

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

In this section, we first evaluate the performance of the synopsis structure in terms of the following:

- compression ratio of the synopsis on different type of data sets, and
- accuracy of cardinality estimation for different types of queries: simple paths, branching paths, and complex paths.

To evaluate the combined effects of the above two properties, we compare accuracy in different space budgets against a state-of-the-art synopsis structure TreeSketch [10]. TreeSketch is considered the best synopsis in terms of accuracy for branching path queries, and it subsumes XSketch 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 prediction 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 code of the TreeSketch system is obtained from the original authors. The experiments for the efficiency of estimation algorithms are run five times and the averages are reported.

6.1 Data sets and workload

We tested synthetic and real data sets with different characteristics: simple without recursion (DBLP[4], Swis-

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

data sets	total size	# of nodes	avg/max depth	avg/max fan-out	avg/max rec. level	# distinct paths	kernel size
DBLP	169 MB	4022548	3 / 6	10.1 / 396243	1 / 2	127	2.8 KB
XMark10	11 MB	167865	5.56 / 12	3.66 / 2550	1.04 / 2	502	2.7 KB
XMark100	116 MB	1666315	5.56 / 12	3.67 / 25500	1.04 / 2	514	2.7 KB
Treebank.05	3.4 MB	121332	8.44 / 30	2.33 / 2791	2.3 / 9	34133	24.2 KB
Treebank	86 MB	2437666	8.42 / 36	2.33 / 56384	2.3 / 11	338748	72.7 KB
SwissProt	114 MB	2977031	3.57 / 5	6.75 / 50000	1 / 1	117	0.7 KB
TPC-H	34 MB	1106689	3.87 / 4	14.8 / 15000	1 / 1	27	0.73 KB
NASA	25 MB	476646	5.98 / 8	2.78 / 2435	1 / 2	95	2.22 KB
XBench TC/MD	121 MB	1115661	6.3 / 8	3.73 / 2600	1.81 / 3	33	0.8 KB

Table 2: Characteristics of experimental data sets

sProt[5], and TPC-H[5]), complex with small degree of recursion (XMark [11], NASA[5], and XBench Tand (c, s) are TC/MD [14]), and complex with high degree of recursion (Treebank[5]). In this paper, we choose DBLP, XMark10 and XMark100 (XMark with 10MB and 100MB of sizes, respectively), and Treebank.05 (randomly chosen 5% of Treebank) and full Treebank to be representative data sets for the three categories. The basic statistics about the data sets are listed in Table 2.

We divided the workload into three categories: simple path (SP) queries that are linear paths containing /-axes only, branching path (BP) queries that include predicates but also only have /-axes, and complex path (CP) queries that contain predicates and //-axes. For each data set, we generate all possible SP queries, and $1,000$ random BP and CP queries. The randomly generated queries are non-trivial. A sample CP query looks like `//regions/australia/item[shipping]/location`. The full set of test workload for the data sets DBLP, XMark10, XMark100, Treebank.05 and Treebank can be found at `http://db.uwaterloo.ca/~ddbms/publications/xml/XSeed_workload.tgz`.

6.2 Construction time

For each data set, we measure the time for constructing the kernel and shell separately. The big picture of where the construction and estimation fit are shown in Figure 1. In this picture, the path tree and the NoK storage structure are used to calculate the real cardinalities of the simple path and branching path queries, and the kernel is used to calculate the estimated cardinality. As described in Section 5, branching paths are estimated only for those path tree nodes whose backward selectivity is less than some threshold (denoted as $BSEL\_THRESHOLD$). We use 0.1 as the $BSEL\_THRESHOLD$ for all the data sets except Treebank, for which the threshold is set as 0.001.

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

The construction time for XSEED and TreeSketch are given in Table 3. In this table, "DNF" indicates that the construction did not finish in the time limit of 24 hours. The construction time for XSEED consists of the kernel construction time and the shell construction time (first and second part, respectively). The total construction time is the sum of these two numbers. As shown in the table, the kernel construction time is negligible for all data sets, and the shell construction time is reasonable. Comparing to TreeSketch, XSEED construction times are much smaller.

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 three error metrics to evaluate the goodness of the estimations: Root-Mean-Squared Error (RMSE), Normalized RMSE (NRMSE), and Coefficient of Determination (R-sq). The RMSE is defined to be $\sqrt{(\sum_{i=1}^{n}(e_i - a_i)^2)/n}$, where e_i and a_i stands for the estimated and actual result sizes, respectively for the i-th query in the workload. The RMSE measures the average errors over the 1000 queries. The NRMSE is adopted from [15] and is defined to be RMSE/$\bar{a}$, where $\bar{a} = (\sum_{i=1}^{n} a_i)/n$. NRMSE is a measure of the average errors per one unit of accurate result size. The $R\text{-}sq$ measures the proportion of variability in the cardinality estimation, is given by $R\text{-}sq = \frac{\left(\sum_{i=1}^{n}(e_i-\bar{e})(a_i-\bar{a})\right)^2}{\left(\sum_{i=1}^{n}(e_i-\bar{e})^2\right)\left(\sum_{i=1}^{n}(a_i-\bar{a})^2\right)}$, where $\bar{a}$ and $\bar{e}$ are the averages of a_i and e_i, respectively.

Since TreeSketch could not finish after 24 hours on XMark100 and Treebank, we only listed, in Table4, the error metrics on the DBLP, XMark10, and Treebank.05 to represent the three data categories: simple, complex with small degree of recursion, and complex with high degree of recursion. The workload is the combined SP, BP, and CP queries. We tested both systems using 25KB and 50KB memory budgets, as well as testing XSEED kernel without shell, thus reducing the memory requirement. For the

cons. time	DBLP	XMark10	XMark100	TreeBank.05	TreeBank	SwissProt	TPC-H	NASA	XBench TC/MD
TrSktch	37	176	DNF	1140	DNF	DNF	0.47	196	DNF
XSEED	0.24/27	0.01/0.27	0.1/2.7	0.008/52	0.168/261	0.17/127	0.286/0	0.28/0.071	0.35/0.37

Table 3: Synopses construction time for different data sets (all times are in minutes)

DBLP and XMark10 data sets, XSEED only uses 20KB and 25KB memory respectively for the total of kernel and shell, thus their error metrics on 25KB and 50KB are the same. Even without the help from shell, the XSEED kernel outperforms TreeSketch on XMark10 and Treebank.05 data sets with 50KB memory budget. The reason is that the TreeSketch synopsis does not recognize recursions in the document, thus 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, the TreeSketch has a better performance than the bare XSEED kernel. However, spending a small amount of memory on the XSEED shell greatly improves its performance. The RMSE for XSEED with 25KB (i.e., generating and using a modest amount of hyper-edge table) 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 a better heuristics to address this problem, although we have not investigate it in this paper.

6.4 Efficiency of cardinality estimation algoirthm

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 [16] that is extended to support //-axes.

The efficiency of the cardinality estimation algorithm depends on how many tree nodes are there the 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 for the next vertex to visit. In the above experiments, the EPT are generated using the threshold of 20 (which means that if the estimated cardinality of the next vertex in depth-first order is less than 20, then it will not be visited), and the ratio of size of EPT to the size of the original XML tree is 6.9% and 5.5%.

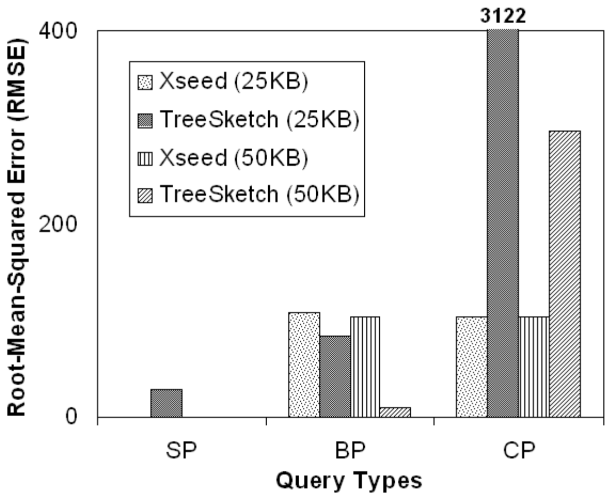


Figure 5: Performance comparison for separate query types on DBLP

The average ratios of the estimation time to the actual 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 are more than an order of magnitude is because the XSEED kernel for them are very similar (because they are generated using the same schema and distribution, but different scale factor), but the size of the XML documents differs by an order of magnitude.

7 Related work

There are many approaches dealing with cardinality estimation for path queries; see, e.g., [7, 4, 1, 8, 6, 13, 2, 10, 12]. Some of them [7, 1, 13, 12] focus on a subset of path expressions, e.g., simple paths (linear chain of steps that are connected by /-axis) or linear recursive paths. Moreover, none of them directly addresses recursive data sets, and it is

memory budgets		DBLP			XMark10			Treebank.05		
		RMSE	NRMSE	R-sq	RMSE	NRMSE	R-sq	RMSE	NRMSE	R-sq
XSEED kernel		1960.5	0.154	0.99	39.6	0.151	0.99	22.7	1.69	0.97
25KB mem	XSEED	103	0.0081	1	3.737	0.0143	1	22.7	1.69	0.97
	TreeSketch	221.5	0.0167	1	62.738	0.2373	0.994	229.5823	8.7714	0.6018
50KB mem	XSEED	103	0.0081	1	3.737	0.0143	1	12.82	0.9561	0.991
	TreeSketch	203.09	0.0159	1	58.3946	0.2209	0.9948	227.1157	8.6771	0.6082

Table 4: Error metrics for XSEED and TreeSketch

performances	DBLP	XMark10	XMark100	Treebank.05	Treebank
EPT to XML tree ratio	0.0035%	0.036%	0.05%	6.9%	5.5%
estimation time to actual running time ratio	0.018%	0.57%	0.0916%	2%	1.5%

Table 5: Estimation time vs. actual execution time

not clear how to extend them to support such type of data. In these work, only [7] and [12] support incremental maintenance of the summarization structures to make it adaptable to data updates.

TreeSketch [10], an extension to XSketch [8], synopsis can estimate the cardinality of branching path queries quite 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 hardly 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 (kernel and shell) of granularity; while TreeSketch tries to preserve structural information as a complex and unified structure.

The idea of hyper-edge table has been inspired by previously proposals [2, 12]. Aboulnaga et al. [2] try to record the actual statistics of previous workload into a table and reuse it later. Wang et al. in [12] uses the bloom filter technique to compactly store the cardinality information about simple paths. In this paper, we only use a hash value for that purpose since practice demonstrate that a single good hash function only produce a few conflicts for thousands of simple paths.

8 Conclusion and future work

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 structure 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.

To further improve the synopsis, we are working on methods to further compress the XSEED kernel when it is large (e.g., in Treebank), and investigate different heuristics for efficiently constructing the synopsis and for capturing the most erroneous queries.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 591–600, 2001.

[2] A. Aboulnaga and J. F. Naughton. Building XML Statistics for the Hidden Web. In *Proc. 12th Int. Conf. on Information and Knowledge Management*, 2003.

[3] D. Chamberlin. XQuery: An XML Query Language. *IBM Systems Journal*, 41(40):597–615, 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 *Proc. 17th Int. Conf. on Data Engineering*, pages 595–604, 2001.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. StatiX: Making XML Count. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 181–191, 2002.

[7] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. 23th Int. Conf. on Very Large Data Bases*, pages 436–445, 1997.

[8] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph Structured XML Databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 358–369, 2002.

[9] N. Polyzotis and M. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proc. 28th Int. Conf. on Very Large Data Bases*, pages 466–477, 2002.

[10] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML Query Answers. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 263–274, 2004.

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

[12] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 240–251, 2004.

[13] Y. Wu, J. M. Patel, and H. Jagadish. Estimating Answer Sizes for XML Queries. In *Advances in Database Technology — EDBT'02*, pages 590–680, 2002.

[14] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proc. 20th Int. Conf. on Data Engineering*, 2004.

[15] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical Learning Techniques for Costing XML Queries. In *Proc. 31st Int. Conf. on Very Large Data Bases*, 2005.

[16] N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. 20th Int. Conf. on Data Engineering*, pages 54 – 65, 2004.

[17] N. Zhang, M. T. Özsu, A. Aboulnaga, and I. F. Ilyas. XSeed: Accurate and Fast Cardinality Estimation for XPath Queries. Technical report, University of Waterloo, 2005. Available at `http://db.uwaterloo.ca/~ddbms/publications/xml/TR_XSEED.pdf`.