Bloom Histogram: Path Selectivity Estimation for XML Data with Updates*

Wei Wang

School of Computer Science
and Engineering
University of NSW, Australia
and
NICTA, Australia
weiw@cse.unsw.edu.au

Haifeng Jiang   Hongjun Lu

Dept. of Computer Science
Hong Kong Univ. of Sci. & Tech.
Hong Kong, China
{jianghf, luhj}@cs.ust.hk

Jeffrey Xu Yu

Department of System Engineering
and Engineering Management
The Chinese Univ. of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

Abstract

Cost-based XML query optimization calls for accurate estimation of the selectivity of *path expressions*. Some other interactive and internet applications can also benefit from such estimations. While there are a number of estimation techniques proposed in the literature, almost none of them has any guarantee on the estimation accuracy within a given space limit. In addition, most of them assume that the XML data are more or less static, i.e., with few updates. In this paper, we present a framework for XML path selectivity estimation in a dynamic context. Specifically, we propose a novel data structure, *bloom histogram*, to approximate XML path frequency distribution within a small space budget and to estimate the path selectivity accurately with the bloom histogram. We obtain the upper bound of its estimation error and discuss the trade-offs between the accuracy and the space limit. To support updates of bloom histograms efficiently when underlying XML data change, a *dynamic summary* layer is used to keep exact or more detailed XML path information. We demonstrate through our extensive experiments that the new solution can achieve significantly higher accuracy with an even smaller space than the previous methods in both static and dynamic environments.

*This work is partially supported by the Research Grant Council of the Hong Kong Special Administrative Region, China (grant AoE/E-01/99).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

Both the amount of XML data and the number of XML applications have exploded in recent years as XML becomes the *de facto* standard for information representation and exchange. Consequently, there is a great demand for efficient XML data management systems for managing complex queries over large volumes of local and Internet-based XML data.

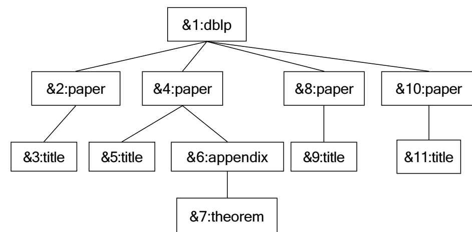


Figure 1: A sample XML data tree (Each node is labeled with a unique ID and its tag)

Efficient query processing requires accurate selectivity estimation of *path expressions*, which are commonly used in XML query languages to locate a subset of XML data. A query optimizer needs such estimation to judiciously select the most efficient query execution plan among alternative ones. For example, given the sample XML data in Figure 1, the following XPath query selects titles of all the papers that have at least one theorem appearing in their appendix: `/dblp/paper[appendix/theorem]/title`. According to the data, it might be most efficient to retrieve the `paper` that has a `theorem` in its `appendix` first, and then, for the only one qualified `paper`, retrieve its `title`. In other words, as is widely adopted in relational query processing, we

should process the most *selective* (path) predicates first—`/dblp/paper/appendix/theorem` in our example. With a path selectivity estimator, we can retrieve such selectivity information effectively without referring to the source data. In addition, accurate path selectivity estimation is desirable in interactive and internet applications as well. The system could warn the user that his/her query is so coarse that either the amount of results will be overwhelming for manual processing or the query will consume a large amount of system resources. Another important application is approximate query answering. The estimated value could be returned as an approximate answer to aggregate queries using the `COUNT` function.

Desiderata of any data structure that is capable of estimating the selectivity of path expressions can be summarized as follows:

- The estimate should be highly accurate. A bad estimate may mislead the query optimizer to choose a bad plan, whose cost could be orders of magnitude higher. Ideally, the estimation error should be upper bounded. This is particularly important for approximate query answering.

- The estimator should be space efficient. When the data structure is to be loaded into main memory during the query optimization phase, it must be small in size. In addition, a smaller size often implies less estimation time, which in turn helps reduce the query optimization time. Finally, if we consider internet-scale applications, both the storage and scalability of the system are dependent on the size of the estimator.

Summarizing the distribution of XML path selectivities is a substantially different problem from most statistics estimation problems considered in the relational context. Therefore, the traditional techniques designed for flat relational data cannot be directly applied to tree-structured XML data. Some work has recently appeared in the XML literature [6, 1, 14, 9, 17, 18]. However, all the work is *ad hoc* in the sense that no theoretical guarantee on the accuracy of the proposed methods is given and the trade-offs between accuracy and space limit are unknown. Another crucial drawback is that most of them consider only the case where data are static.

In this paper, we propose a two-layer solution that is capable of estimating the selectivity of XML path expressions for dynamic data. At the heart of our new solution is a *bloom histogram*, a compact yet high accurate estimator for XML path expressions. A bloom histogram provides better approximation for the original value-frequency distribution by sorting on the frequencies and using bloom filters to record values within each bucket. Several encouraging results are obtained about the bloom histogram: it occupies much less space than that is required by the previous methods, and its estimation error is small and can be bounded probabilistically. We also present optimal construction algorithms for the bloom histogram. To handle data updates, we employ in our solution a dynamic summary, which keeps an exact or approximate description of the necessary information for histogram updates. The dynamic summary is typically larger than the bloom histogram, yet it is still much smaller than the XML data and is easily maintainable. New histograms can thus be recomputed from the dynamic summary, without the costly process of accessing the huge XML data itself.

Our contributions can be summarized as follows:

- We propose a compact yet highly accurate estimator, the bloom histogram, which has theoretical upper bound on its estimation error. Furthermore, we analyze the trade-offs between accuracy and space requirement for the bloom histogram so that it becomes possible to set appropriate space limits based on specific estimation accuracy requirements in real applications. We note that the bloom histogram is an interesting data structure in its own right and might find applications in other domains.

- We consider the problem of maintaining the bloom histogram when underlying data change and propose to use the dynamic summary with a controllable size to track approximate or exact changes of path selectivities. The bloom histogram can then be rebuilt without the costly process of accessing the source XML data. This solution can be generalized to work with previously proposed estimators as well.

- We complement our analytical results with an extensive experimental study. Our results indicate that the new method can indeed estimate the selectivity of XML path expressions accurately within a small space in both static and dynamic environments.

The remainder of the paper is organized as follows. We define formally the path-expression selectivity estimation problem in Section 2. We also provide an overview of our proposed solution. Sections 3 and 4 discuss the two major components of our system: the estimator and the dynamic summary, respectively. Specifically, *bloom histogram*, a new data structure and method to accurately estimate path selectivity for XML data is presented in Section 3, and Section 4 discusses two solutions to dealing with data updates. Section 5 presents our experimental results. Related work is presented in Section 6. Section 7 concludes the paper.

2 Problem definition and overview of our solution

In this section, we first define the problem and then present an overview of our solution. More technical details of our solution will be presented in Sections 3 and 4.

2.1 Problem definition

XML documents are usually modeled as a node-labeled, rooted tree, such as the one shown in Figure 1. In XQuery data model[1], seven types of nodes are defined: document, element, attribute, text, namespace, processing instruction, and comment nodes. In this paper, we only consider element nodes and attribute nodes. Text nodes are treated as the values of their parent element nodes. For any element or attribute node, the tags on its root-to-node path form its *label path*. For example, in Figure 1, the label path for the node `&6` is `/dblp/paper/appendix`.

XPath is a common query language for locating a subset of nodes in an XML data tree. In this paper, we only deal with *simple path expressions* in the form of $/t_1/t_2/\ldots/t_n$ and $//t_1/t_2/\ldots/t_n$, where t_i is a tag (i.e., element or attribute) name. We will focus on the former type of simple path expressions in this paper for the ease of illustration. Nonetheless, our method can be extended to handle the latter type of simple path expressions.

Given a simple path expression p, it may match with a set of label paths lp_i ($1 \leq i \leq k$). Let $c(lp_i)$ be the number of nodes whose label paths are lp_i and N be the total number of nodes in the XML data tree, then the selectivity of p is defined as: $sel(p) = \frac{1}{N}\sum_{1\leq i\leq k} c(lp_i)$. Since N can be easily maintained, we only need to estimate the $c(lp_i)$ values. Therefore, the static path-expression selectivity problem is to build a data structure within a space limit and support estimating the selectivity of path expression queries accurately. The dynamic version of the same problem studies the case when the underlying XML data are dynamic due to updates.

2.2 Overview of the solution

Figure 2 illustrates our proposed solution for XML path selectivity estimation in a dynamic environment. There are three components in the system: the data file, the dynamic summary and the estimator. The latter two components form our two-layer solution. Data files are typically large, in the magnitude of megabytes or gigabytes. The estimator is responsible for accurately and efficiently estimating the selectivity of path expression queries. Typically, the estimator is of several kilobytes in size. The dynamic summary component is designed to keep necessary information about the changing data so that the estimator can be updated *without* the need to access the data file.

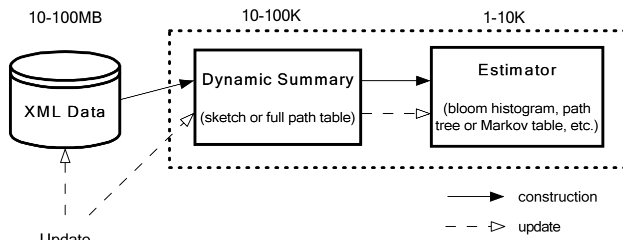


Figure 2: System overview

There are two types of activities in the system: *construction* and *updating*. In the construction phase, dynamic summaries are built from source XML data files and used to create the estimators (corresponding to data files). During *updating*, updates to XML data files are delegated to the dynamic summaries to reflect the latest data distribution. The estimators will be recomputed either periodically or on demand from the dynamic summaries.

In this paper, we focus on a new estimator, the bloom histogram. However, previous methods, such as Path Tree and Markov Table, can be used in our system as well.

3 The bloom histogram

In this section, we propose the bloom histogram, a new estimator for selectivity estimation. Compared to other alternatives, such as Path Tree and Markov Table, it is of smaller size yet offers superior accuracy. We present the basic structure of a bloom histogram, and the algorithms to estimate the selectivity of XML path expressions using bloom histograms. The algorithm for constructing a bloom histogram with minimum estimation error is also presented.

3.1 The basic bloom histogram

The bloom histogram keeps counting statistics for paths in XML data. The design objective of the bloom histogram, like all other histograms, is small size yet high estimation accuracy.

Given an XML document D, it is always possible to construct a *path-count table* $T(path, count)$ such that for each path $path_i$ in D, there is a tuple t_i in T with $t_i.path = path_i$ and $t_i.count = count_i$, where $count_i$ is the number of occurrences (also referred to as $frequency$) of $path_i$ in D. Given a path p, we can then use T to obtain the selectivity of p. A histogram is a commonly used data structure to approximate data distribution of a given attribute—or, the *target attribute*. In the context of XML path selectivity estimation, the target attribute is the *path* attribute of T. A histogram H for D is therefore a two-column table $H(paths, v)$ where *paths* represents a *set* of paths in D

[1] `http://www.w3.org/TR/xpath-datamodel`

and v is a representative value for the frequency values of all $path_i$ in $paths$. Given a path p, we can find from H a tuple H_i with $p \in H_i.paths$ and return $H_i.v$ as an estimation of the frequency of p in D. Different from the path-count table T, the table H usually contains a fixed number, b, of tuples (often referred as *buckets*). To design a good histogram, we need to choose an appropriate b and the way in which the value range of the target attribute is divided into the given b buckets so that both the size of H and the accuracy of estimation are acceptable. When we construct a histogram for XML paths, a new issue arises. That is, how to represent paths so that the bucket that contains the count for a given path can be quickly identified.

Path	Count
/a	10
/a/f	10
/a/e	499
/a/c	501
/a/b	999
/a/d	1001

Bloom Filter	Count
BF(/a, /a/f)	10
BF(/a/c, /a/e)	500
BF(/a/d, /a/b)	1000

Figure 3: An example path-count table and its bloom histogram. BF(P) is a bloom filter for a set of paths.

To provide an elegant solution to all the above design issues, we adopt a new type of histograms, *bloom histogram*, to maintain approximate counts for XML paths. The bloom histogram has two novel features:

1. Instead of dividing the value range of the target attribute into buckets, we sort the frequency values and then group paths with similar frequency values into buckets so that the estimation errors can be reduced; and
2. Bloom filters are used to represent the set of paths in each bucket so that, for a given path, the bucket containing the frequency of the path can be quickly located.

Figure 3 shows an example path table and a corresponding bloom histogram. There are 6 distinct paths in the path table. In the bloom histogram, they are grouped into 3 buckets based on their frequencies. The first column of the histogram contains bloom filters, $BF(P)$, where P is a set of paths.

A bloom filter is a succinct data structure that represents a set and supports approximate *set membership* queries [2]. Specifically, given a set $P = \{p_1, p_2, \ldots, p_n\}$, a bloom filter is a bit array of length m with k independent hash functions $h_1, h_2, \ldots, h_k$; and the hash functions are assumed to be able to hash x_i into a random number uniformly over range $[1, m]$. The bit array is initially set to 0. To insert a data element p to a bloom filter, k hash functions are applied to the value of p and the bit at position $h_i(p)$ is set to 1. To test whether a query element q is in the set represented by a bloom filter, the same k hash functions are applied to the value of q. The result is true only if every bit in the bloom filter at position $h_i(q)$ is set to 1, for all $1 \leq i \leq k$.

A good feature of a bloom filter is that it only has *false positive* error, i.e., errors due to incorrectly identifying that an element y belongs to the set S while it does not. In addition, the probability of its false positive error (denoted as ϵ) can be controlled by the parameters k and m:

$$\epsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Since the approximation is very accurate, we will use it to represent the actual error in our study. Further analysis also shows that ϵ is minimized by choosing $k = \ln 2 \cdot \frac{m}{n}$. Let $l = \frac{m}{n}$ be the *load factor*. The optimal ϵ is 0.6185^l.

In the next, we first discuss how to use a bloom histogram to estimate the selectivity of a path expression and the possible estimation errors. We then discuss the construction of an optimal bloom histogram.

3.2 Selectivity estimation using bloom histograms

In this section, we first give the algorithm that searches a bloom histogram to return the frequency of a given path from which the path selectivity can be estimated. We then analyze the estimation errors.

3.2.1 Selectivity estimation

Algorithm 1 QueryBloomHistogram(BH, p)

```
1: count = 0; k = 0;
2: for i = 1 to b do
3:    if IsMember(p, BH.BF[i]) then
4:       count+ = BH.count[i];
5:       k++;
6:    end if
7: end for
8: if  k > 0 then
9:    return count/k;
10: else
11:    return 0;
12: end if
```

With a bloom histogram BH, we can obtain the approximate frequency for a given path p. The algorithm is outlined in Algorithm 1. In line 3 of the algorithm, function IsMember(p, BF) takes a path and a bloom filter for a set of paths as input, and returns TRUE if p is in the set of paths. As mentioned previously, this can be done by applying k hash functions to check whether all bits corresponding to $h_i(p), 1 \leq i \leq k$ are set. Although a path should be a member of the path set of at most one bucket, multiple bloom filters may report that the path belongs to them. In such cases, the frequency returned is the average frequency of those sets

(lines 7-8). Note that, this is one type of error introduced by the bloom filters, and the probability that a path belongs to more than one set is very low and bounded by $b\epsilon$. The error can be further minimized by our optimization techniques discussed later.

Algorithm 1 needs to calculate $b \cdot k$ hash functions. As $k = O(\log \frac{1}{\epsilon})$, the time complexity of the algorithm is $O(b \log \frac{1}{\epsilon})$.

3.2.2 Errors in selectivity estimation

We now analyze the errors in selectivity estimation using bloom histograms. We use *absolute error* as the error metric. The absolute error e_X of an estimate X is the absolute difference between the estimated value and the real value $\hat{X}$, i.e., $e_X = |X - \hat{X}|$. The absolute-error metric has been used in previous XML path selectivity work as well. The main reasons are (a) the relative error is meaningless for queries which have an empty result set; and (b) the relative error with sanity bound is sensitive to the value of the sanity bound parameter.

During our analysis, we assume the full path-count table has n entries; and the bloom histogram has b buckets and the bloom filter error rate is ϵ. The count values in the bloom histogram are within $(0, M]$, for some constant M; and the count value in bucket i is V_i, $1 < V_i \leq M$. Let V be the actual frequency of a path. Query workload on selectivity can be divided into two types according to V of the query path. Those queries with $V = 0$ are referred as *negative query workload*, and those with $V > 0$ are referred as *positive query workload*. We assume queries in the workload are distinct and have the same frequency. It is easy to generalize our method for skewed workload.

Estimation error for negative query workload

For negative query workload, the estimation error comes from one or more buckets incorrectly reporting that the given query belongs to them. Let e_k denote the absolute error when k buckets report that a query path with frequency V belongs to them. We have $e_k = \left| \frac{1}{k} \sum_{i=1}^{k} V_i - V \right|$ where V_i is the count value in bucket i (and $V = 0$ for negative query workload). It is obvious that $0 \leq e_k \leq M$. Then, the expected error for negative queries can be calculated by summing over all possible errors when k buckets incorrectly report that the given query belongs to them, for $0 \leq k \leq b$.

$$
\begin{aligned}
E[e] &= 0 + \sum_{i=1}^{b} \binom{b}{i} \epsilon^i (1-\epsilon)^{b-i} \cdot E[e_i] \\
&< b\epsilon \cdot M
\end{aligned}
\qquad (1)
$$

By *Markov Inequality*, for any given $\gamma \geq 0$:

$$
Pr[e \geq \gamma] \quad \leq \quad \frac{b\epsilon \cdot M}{\gamma}
$$

Therefore, we have the following theorem that bounds the error of a bloom histogram for negative queries.

Theorem 3.1. *For any given γ and δ, there exists a bloom histogram such that with probability at least $1 - \delta$, the estimation error of the bloom histogram for the negative queries is within γ, i.e., $Pr[|X - \hat{X}| \leq \gamma] \geq 1 - \delta$ holds.*

Estimation Error for Positive Query Workload

For positive queries, the expected estimation error is

$$
\begin{aligned}
E[e] &= (1-\epsilon)^{b-1} \cdot E[|V - V_*|] \\
&+ \sum_{i=1}^{b-1} \binom{b-1}{i} \epsilon^i (1-\epsilon)^{b-1-i} e_{i+1}
\end{aligned}
$$

where V and V_* are the actual and returned frequency values of the query, respectively. In the above equation, the first term is the error due to histogram approximation when the correct bucket is located; and the second term is the error due to the conflict resolution method when a query is reported to belong to multiple buckets. Again, since e_i is bounded by M, we have

$$
E[e] < (1-\epsilon)^{b-1} \cdot E[|V - V_*|] + (b-1)\epsilon \cdot M \qquad (2)
$$

Similarly, by Markov Inequality, we have the following theorem that bounds the error of a bloom histogram for positive queries.

Theorem 3.2. *For any given γ and δ, there exists a bloom histogram such that with probability at least $1 - \delta$, the estimation error of the bloom histogram for the positive queries is within γ, i.e., $Pr[|X - \hat{X}| \leq \gamma] \geq 1 - \delta$ holds.*

With those relationships established, we are ready to discuss our optimal histogram construction algorithm, which directly minimizes the error for positive queries.

3.3 Optimal histogram construction

Given a bloom histogram with b and ϵ parameters, in order to minimize the expected error for the positive workload, we need to minimize $E[|V - V_*|]$ in Equation 2. This is the goal of our optimal histogram construction algorithm.

We assume that all the paths and their counts are stored in a table sorted in the non-decreasing order on their counts, as shown in Figure 3; otherwise an additional sorting is required. For simplicity, we use

$count[k]$, $1 \leq k \leq n$, to denote the count of the kth path with $count[1] \leq count[2] \leq \ldots count[n]$. We note that:

$$E[|V - V_*|] = \frac{1}{n} \sum_{i=1}^{b} \left(\sum_{j=s[i]}^{e[i]} (|count[j] - V_i|) \right)$$

where $s[i]$ and $e[i]$ are the start and end indexes of the path table for the set of paths in the ith bucket, respectively.

Because this error metric is different from that in traditional histograms, such as the sum of square error in V-optimal histogram [13], we cannot directly apply previous algorithms. However, we can follow the same spirit in [13] when devising our optimal histogram construction algorithm. There are two main tasks: (1) to determine the optimal bucket boundaries efficiently; and (2) to select appropriate count values for each bucket.

We first address the second issue, i.e., to select appropriate V_i values for the buckets after the bucket boundaries have been fixed. To minimize $E[|V - V_*|]$, we only need to choose V_i to be the smaller median value in each bucket, based on the following observation: the median value of the data points in a bucket minimizes the function $\sum_{j=s[i]}^{e[i]}(|count[j] - V_i|)$.

Next, we investigate how to determine the optimal bucket boundaries efficiently. We leverage the dynamic programming paradigm, due to the following observation:

$$OPT[n, b] = \min_{i=b-1}^{n-1} \{OPT[i, b-1] + f(i+1, n)\} \quad (3)$$

where $OPT[x, b]$ denotes the sum of errors when b buckets are used to approximate the first x data points in an optimal way. $f(x, y)$ is the error of a single bucket comprising of data points within the index range $[x, y]$. The naïve computation of $f(x, y)$ takes $O(y - x + 1)$ time and will increase the running time of the dynamic programming algorithm. We observe that

$$\begin{aligned} f(x, y) &= PSUM[y] + PSUM[x-1] \\ &\quad -2PSUM[t] + (2t - x - y + 1)count[t] \end{aligned}$$

where $PSUM[]$ is the prefix sum array and t is the index of the median value. Hence we can compute $f(x, y)$ in constant time by building the prefix sum array for all the data points.

Algorithm 2 shows the optimal histogram-construction algorithm based on dynamic programming. Lines 1–4 initialize the $PSUM[]$ array, which helps to compute $f(x, y)$ function in constant time. Lines 5–7 initialize the $OPT[]$ for the special case of 1 bucket. Lines 8–15 are the dynamic programming part: We compute $OPT[j, b]$ according to the recurrence equation (Equation 3). Optimal bucket boundary can be reported by additional bookkeeping, which is omitted in the algorithm. The total errors of the optimal histogram, $OPT[n, b]$, are returned.

Algorithm 2 BuildHistogram($x[]$, n, b)

```
1: PSUM[1] = x[1]
2: for i = 2 to n do
3:   PSUM[i] = PSUM[i-1] + x[i]
4: end for
5: for i = 1 to n do
6:   OPT[i,1] = f(1,i)
7: end for
8: for k = 2 to b do
9:   for j = 1 to n do
10:    OPT[j,k] = +∞
11:    for i = k-1 to j-1 do
12:      OPT[j,k] = min(OPT[j,k], OPT[i,k-1]+f(i+1,j))
13:    end for
14:  end for
15: end for
16: return OPT[n,b]
```

The time complexity of the algorithm is $O(bn^2)$. The space complexity is $O(n)$.

We note that $E[|V - V_*|]$ is exactly $\frac{1}{n} \cdot OPT[n, b]$, thus it can be computed after the building of the histogram. Therefore, given a dataset, the expected error for both positive and negative queries can be determined once the parameters b, ϵ and M are fixed.

3.4 Space bounded bloom histograms

In this subsection, we analyze the space requirement of our bloom histogram and discuss how to construct a bloom histogram within a space limit. The relationship between the space limit and the expected error of the resulting bloom histogram is also discussed.

The size of a bloom histogram is the sum of the sizes of b bloom filters for n paths and b approximate count values. The total size of b bloom filters is $l * (\sum_{1 \leq i \leq b} n_i) \approx 0.2602 \cdot \ln \frac{1}{\epsilon} \cdot n$, where l is the load factor defined in Section 3.1 and each bucket contains n_i values. Therefore, the total size of a bloom histogram is $0.2602 \cdot \ln \frac{1}{\epsilon} \cdot n + 4b$.

The minimum size of a bloom filter with a fixed ϵ parameter is therefore $S_{\min} = 0.2602 \cdot \ln \frac{1}{\epsilon} \cdot n + 4$, if we use only one bucket. Since the maximum number of buckets needed is upper bounded by the number of distinct values of path counts, the maximum size of a bloom filter, $S_{\max}$, is in turn bounded by $0.2602 \cdot \ln \frac{1}{\epsilon} \cdot n + 4n$.

Given a space limit S,

- If $S > S_{\max}$, we can either use less space by setting b to the number of distinct path counts, or choosing a smaller ϵ to utilize the additional space and reduce the estimation error.

- If $S \in [S_{\min}, S_{\max}]$, we fix the number b of buckets to $\frac{S - S_{\min}}{4}$.

- If $S < S_{\min}$, we need to increase ϵ. Another possible solution is to prune paths with the smallest count values, thus reducing n. This is similar to the No-* pruning strategy used in [1].

Therefore, by replacing the parameter b with the function of space limit S in Equations 1 and 2, we can establish the association between the space a bloom histogram uses and the expected estimation errors. This is a desirable feature and enables users to have a better understanding of the trade-offs.

We note that in practice, we do not always choose the largest possible b under a given budget. The rationale is that, although a larger b can reduce the error due to value distributions within each bucket, it will increase the errors by misclassifying a path into multiple buckets (see, Equation 2). This is reflected in our experiments, especially when ϵ is not small enough. Since the term $E[|V - V_*|]$ cannot be explicitly expressed as a function on b, currently we can only find the optimal b value empirically by considering all possible b values.

Compared to the previous Path Tree estimator [1], the space requirement of a bloom histogram is extremely small. Assume there are n distinct paths. A full path tree will occupy at least $16n + 16$ bytes. This translates to the following statement: a full path tree is always larger than the worst-case bloom histogram unless $\epsilon < 9 \times 10^{-21}$.

4 Dynamic summary

In this section, we discuss the dynamic summary component, which enables us to maintain the histogram under updates. Two alternative solutions are proposed and compared.

4.1 Overview

The dynamic summary component is an intermediate, small-sized data structure from which the bloom histogram can be recomputed periodically or upon request. The component can be described as follows:

- When updates arrive, we update not only the XML data, but also the dynamic summary. Specifically, all the paths are extracted from the updates and then grouped. In general, the dynamic summary takes as input a sequence of primitive update operations denoted as $U_i(p_i, \Delta_i)$, where Δ_i is the amount of *relative* change of the frequency of path p_i. Note that such primitive update operations can express both insertion and deletion.

- The dynamic summary processes the sequence of primitive update operations. Depending on the actual data structure of the dynamic summary, either approximate or exact value distribution can be maintained.

- The dynamic summary supports a *rebuild* operation to recompute a new bloom histogram from an approximate or exact path-count table maintained by the dynamic summary.

Next, we discuss some of the technical details that are related to the implementation of the above operations.

4.2 Two candidate data structures

Here, we consider two alternative data structures that implement the dynamic summary interface: the *Count-Min Sketch* based method and the *Full Label-Path Table* based method.

Count-min sketch based method

Our count-min sketch based method comprises of two parts:

1. A list of *hashID*s, in the domain of $[0, N]$.
2. A count-min sketch built for a length-N array of values.

The count-min sketch is proposed recently as an efficient and versatile synopsis structure for an array of n values [8]. Given parameters ε and δ, it employs $d = \lceil \log \frac{1}{\delta} \rceil$ pair-wise independent hash functions. Each hash function can map an incoming value into a random position within an array of $\lceil \frac{2}{\varepsilon} \rceil$ integers. With probability at least $1 - \delta$, point query, range query and inner product of two arrays can all be well approximated by the sketch. Its advantages over previous proposals (e.g., backing sample [10] or other sketches [11]) are (a) it supports deletion; and (b) the size requirement of count-min sketch is smaller than other synopses both in theory and in practice. The size of a count-min sketch is exactly $4\lceil \frac{e}{\varepsilon} \rceil \lceil \ln \frac{1}{\delta} \rceil + 8 \lceil \ln \frac{1}{\delta} \rceil$ bytes, while other sketches use space linear to $\frac{1}{\varepsilon^2}$ with some constant hidden in the $O()$ notation.

The update and rebuilding operations can be supported as follows:

- To process the update operation $U_i(p_i, \Delta_i)$, we first obtain an integer ID_i by hashing the path p_i; and then invoke the standard update procedure of the count-min sketch that accepts an (ID_i, Δ_i) pair. We add ID_i into the hashID list if it does not exist in the current hashID list.
- An approximate path-count table can be reconstructed to rebuild the bloom histogram: we iterate through the hashID list and issue a point query to retrieve the approximate count value for each hashID.

We note that the rationale to use the hashID list is to save space. A naïve solution without using hashing would need to store the entire list of paths, which is usually much larger. One subtle thing is that those hashIDs should be treated as "paths" when building

Table 1: Statistics of datasets

Dataset	Size	#Path	M	| Path Table |	|Path Tree|	| MT ($m = 2$) |	Comment
DBLP	184M	155	666920	3069	3100	1384	Regular
XMark	111M	548	59486	14578	10860	1584	Irregular
MathML	110K	734	23	49393	14680	1248	Extremely Irregular

the bloom filters for the bloom histogram and the same hash function needs to be applied when querying the resulting bloom histogram with a real path.

Update and rebuilding overhead: The update overhead for the count-min sketch based method can be shown to be $O(\ln \frac{1}{\delta})$. Let $R(n, b)$ be the cost of the bloom histogram construction algorithm (exclusive of the sorting cost). The cost of rebuilding is $O(n \ln \frac{1}{\delta} + n \log n + R(n, b))$. The rebuilding cost can be further reduced by computing on samples only.

Full label-path table based method

A full label-path table records the count values for all distinctive label paths in the XML data tree. Therefore, both the update and rebuilding operations can be supported in an exact manner:

- To process the update operation $U_i(p_i, \Delta_i)$, we simply use a hash lookup to locate and then update the corresponding entry in the full label-path table.
- The full label-path table is exactly the path-count table that can be fed into the bloom histogram construction algorithm.

Update and rebuilding overhead: The update overhead for the full label-path based solution is $O(1)$. Let $R(n, b)$ be the rebuilding cost of the bloom histogram construction algorithm (exclusive of the sorting cost). The total rebuilding cost is $O(n \log n + R(n, b))$.

4.2.1 Discussions

The two methods mainly differ in the update overhead, the size, and the quality of the resulting bloom histogram. In terms of size, the count-min sketch based solution takes $4\lceil \frac{e}{\varepsilon} \rceil \lceil \ln \frac{1}{\delta} \rceil + 8\lceil \ln \frac{1}{\delta} \rceil + 4n$ space, while the full label-path table based solution takes $(4 + k)n$ space, where k is the average length of the paths. Therefore, the count-min sketch based method will take less space when n or k is large. An empirical comparison of the two methods is included in the experiment section.

5 Experiment

We describe in this section the result of extensive experiments we have conducted to compare our method to previous ones. We first describe the experiment setup and then present results that address different aspects of the proposed solution.

5.1 Experiment setup

We implemented our dynamic XML path selectivity system in C++. The hardware is a PC with AMD Atholon 900MHz CPU, 512M memory and 30G hard disk. The operating system is Windows XP Professional. We implemented the proposed two-layer solution. For the estimator component, we implemented the bloom histogram, Path Tree (with sibling-* pruning strategy) and Markov Table algorithms (with suffix-* pruning strategy). They are abbreviated as **BH**, **PT** and **MT**, respectively. For the dynamic summary component, we implemented the CM-Sketch based approach and the Full Label-Path Table based approach. They are abbreviated as **CM** and **LP**, respectively.

The performance metric used in the experiment is the *average* absolute error, defined as $\frac{1}{n} \sum_i^n |X_i - \hat{X}_i|$, where X_i is the real selectivity of the i-th query in the workload and $\hat{X}_i$ is its estimated selectivity. For the BH method, we take the average of its average absolute errors over 10 runs.

Both synthetic and real-world datasets were used in our experiments. In this paper, we present results on DBLP, XMark and MathML datasets. DBLP is a real-world dataset describing computer science bibliography information. The structure of the DBLP dataset is relatively shallow and non-recursive. XMark is a commonly used XML benchmark dataset, simulating an auction database. The structure of the XMark dataset is relatively deep and recursive. Another real-world dataset, the MathML dataset[2], is deliberately chosen as an extreme dataset, because its structure is very deep and highly recursive. Some statistics about the datasets are listed in Table 1, including the size of the dataset, number of paths, the maximum path-count value, size of the Path Table, size of the Path Tree, size of the Markov Table.

We generated both the positive and negative workloads as follows. For the positive workload, we randomly chose 1000 paths from the label-path table. For the negative workload, we chose random tags from the set of distinctive tags and concatenated them into a path. We controlled the distribution of the length of the generated paths to follow a zipf distribution of $\alpha = 0.4$, with the lengths varying from 2 to 4. A generated path was discarded if its selectivity was not 0. We will describe the generation of the update workload in Section 5.4.

[2] Available from `http://support.sciencedirect.com/tectext_sgml.shtml`.

We chose the load factor l of the bloom histogram to be a multiple of 8, so that the resulting bloom histograms are aligned at the byte boundary. Table 2 gives the corresponding error probability ϵ for different choices of load factors. The space budgets are always chosen to be between S_{min} and S_{max} (see, Section 3.4).

Table 2: Load factor vs. error probability

l	$\epsilon = 0.6185^l$
16	4.59×10^{-4}
24	9.82×10^{-6}
32	2.10×10^{-7}

5.2 Accuracy

In this set of experiments, we study and compare the accuracy of different estimators. Figures 4(a)—(c) show the accuracy of three estimators for the positive queries on the DBLP, XMark and MathML datasets. Figures 4(d)—(f) show the accuracy of three estimators for the negative queries on the three datasets. We fix the load factor parameter l to 24, thus the bloom filter error rate ϵ is fixed at 9.82×10^{-6}. We vary the space limit and plot the estimation errors. In all the figures, X-axis is the space limit and Y-axis is the error metric (in *logarithm* scale).

The range of the size limit is chosen to be starting from slightly greater than the minimum size of the bloom histogram to a size much larger than the maximum size of the bloom histogram. For example, in DBLP dataset, the minimum size of the bloom histogram is 469 and the maximum size of the bloom histogram is 801. Our current BH implementation does not use the extra space beyond its current maximum size.

For all the datasets, the accuracy of BH is almost always better than the other two estimators for all the positive queries. The main reasons are as follows.

- BH method is space efficient. Thus, for a given space limit, other methods need to perform significant amount of pruning which sacrifices their accuracy.
- On the other hand, our BH method is guaranteed to find the best grouping of all the paths and this directly minimizes the estimation error. The pruning methods of PT and MT, however, are *ad hoc* in nature.

For the negative queries, BH outperforms PT and MT on DBLP and MathML datasets. It also achieves reasonably good accuracy on the XMark dataset, as its errors are almost always below 10.

There are two interesting findings here:

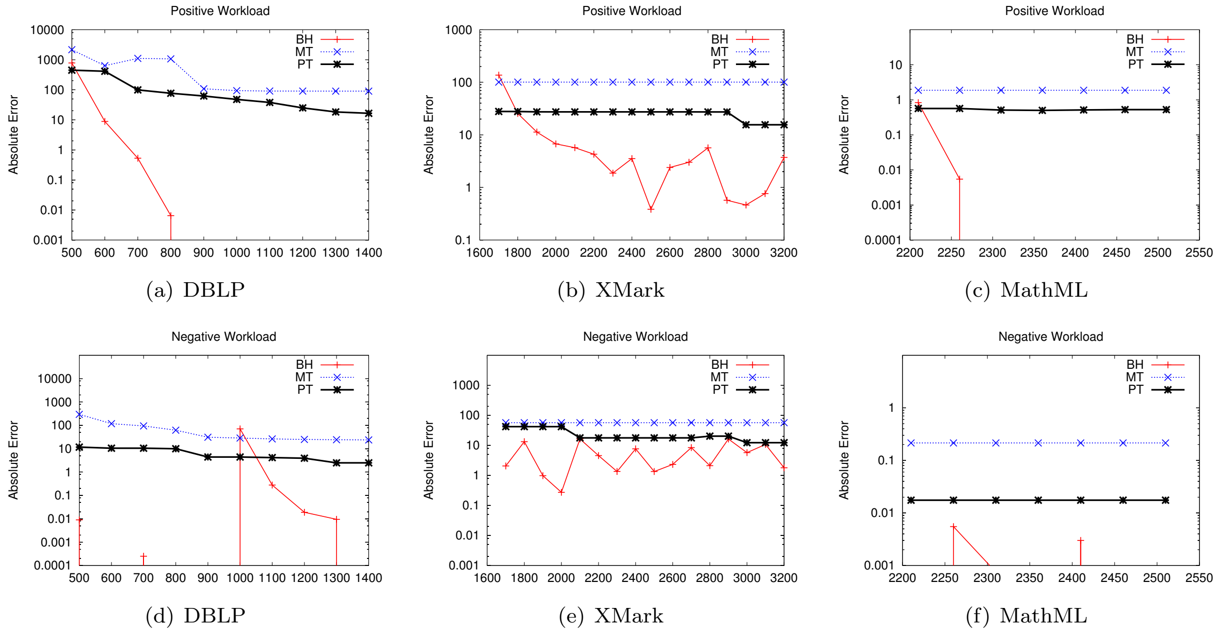
- Comparatively speaking, the BH method has greater accuracy advantages than PT and MT on DBLP and MathML datasets. This is because the number of paths in DBLP is smaller, thus we only need to use a smaller number of buckets (i.e., b is small); and the maximum value of the MathML dataset is very small (i.e., M is small). According to the formulae for the errors of BH, these all lead to a smaller estimation error.
- There are some spikes and fluctuations on the curves. This is because when space budget increases, the number of buckets b also increases. As discussed in Section 3.4, the misclassification error due to a large b value will increase. This part of error is probabilistic in nature and leads to the fluctuations. On the other hand, such error is scaled by M, the maximum path-count value. As a result, fluctuations are more significant on DBLP and XMark datasets than those in the MathML dataset.

5.3 Effect of the bloom filter error (ϵ)

In this subsection, we investigate the effect of the bloom filter error ϵ on the accuracy of the BH method. Figure 5 shows the accuracy of both the positive and negative queries for the DBLP datasets, with different ϵ values ranging from 0.6185^{32} to 0.6185^{16}. As our estimator error formulae predict, the smaller the ϵ value is, the smaller the estimation error will be. Our experiment results agree with this prediction well. For the largest $\epsilon = 0.6185^{16}$, BH does not have significant advantages over the other estimators for both the positive and negative queries. Furthermore, the accuracy of BH fluctuates a lot. If we decrease the ϵ to 0.6185^{24}, BH clearly outperforms other estimators for the positive queries. The accuracy for the negative queries is greatly enhanced as well, with infrequent spikes and fluctuations. If we take $\epsilon = 0.6185^{32}$, BH is then the consistent winner for both query workloads. In terms of storage space, we note that the additional increase in space consumption due to the decrease of ϵ is moderate: the ranges of the bloom histograms for the three cases are $[314, 646]$, $[469, 801]$ and $[624, 956]$, respectively.

5.4 Handling dynamic data

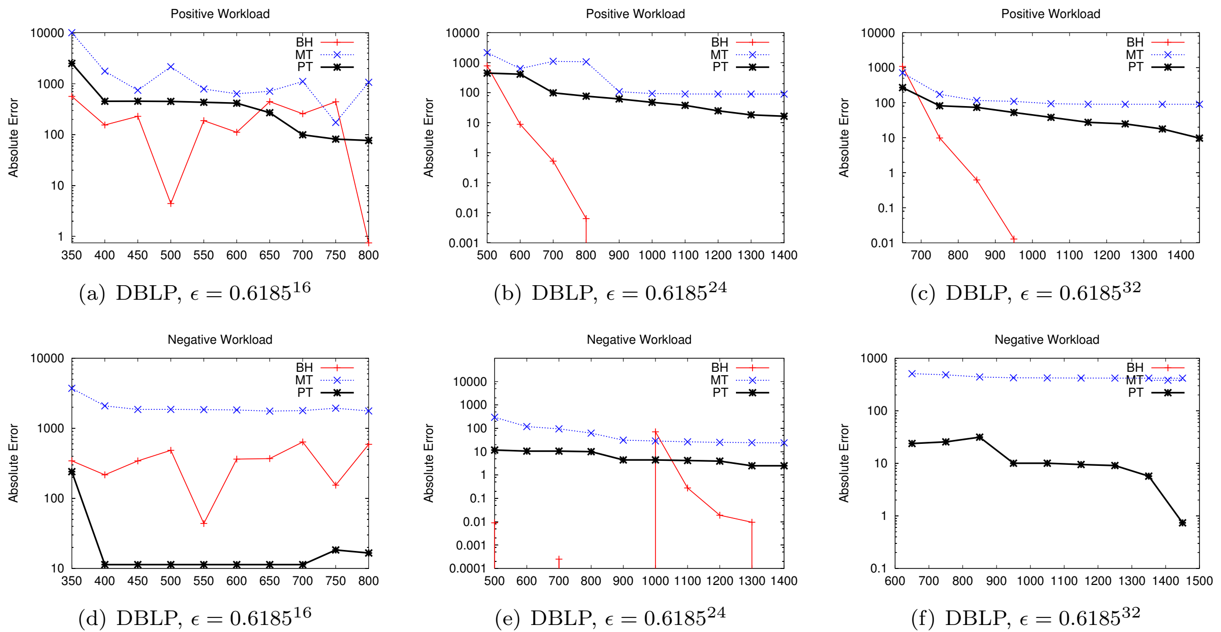
In order to test our proposed methods for dynamic data, we designed an update workload to simulate the process of populating an initially empty XML database in k batch updates. The estimator is rebuilt k times from the dynamic summary. Specifically, we generate the update and querying workloads as follows. We start with a final XML data file: for each distinctive path p_i, its count value is denoted as $count_i$. We randomly pick $k-1$ integers as cut points within the range $[0, count_i]$. After sorting the $k-1$ cut points (denoted as v_{i_j} and specify $v_{i_0} = 0$), we can generate the k updates for this path p_i as $U(p_i, v_{i_j} - v_{i_{j-1}})$. We repeat this process for all paths and thus generate k batch



(a) DBLP (b) XMark (c) MathML

(d) DBLP (e) XMark (f) MathML

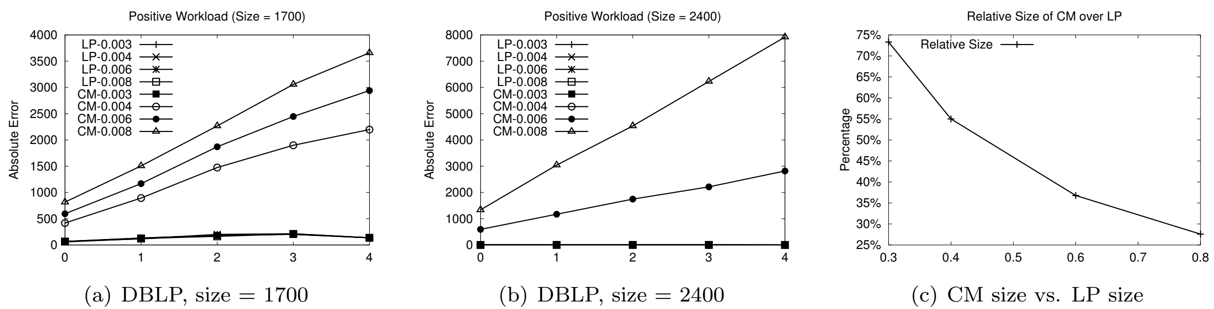
Figure 4: Comparison of different estimators ($\epsilon = 0.6185^{24}$ for BH)



(a) DBLP, $\epsilon = 0.6185^{16}$ (b) DBLP, $\epsilon = 0.6185^{24}$ (c) DBLP, $\epsilon = 0.6185^{32}$

(d) DBLP, $\epsilon = 0.6185^{16}$ (e) DBLP, $\epsilon = 0.6185^{24}$ (f) DBLP, $\epsilon = 0.6185^{32}$

Figure 5: Effectiveness of ϵ



(a) DBLP, size = 1700 (b) DBLP, size = 2400 (c) CM size vs. LP size

Figure 6: LP vs. CM for dynamic data

updates. The generation of both the positive and the negative query workloads is the same as previous experiments.

We compared the following two experimental configurations: LP + BH and CM + BH. Notice that CM is only an approximation of the real path-count distribution, while LP records the exact distribution. In the interest of space, Figure 6 only shows the result of the positive queries. We used five batch updates and fixed the δ parameter for the CM to be 0.013. Figures 6(a) and (b) are obtained by fixing the space limit of the BH estimator to 1700 and 2400 bytes, respectively. For each case, we vary the other parameter of CM, ε, from 0.003 to 0.008. Therefore, the X-axis in Figures 6(a) and (b) is the ε value, and the Y-axis is the estimation error. As we can see from both figures, CM's error is very sensitive to the ε parameter, which determines how well the CM-Sketch approximates the original data distribution. In Figure 6(a), for $\varepsilon = 0.003$, CM can achieve almost the same accuracy as LP, which is optimal as the exact path-count distribution is preserved. However, for $\varepsilon > 0.003$, the estimation errors of CM are significantly larger. Similar observation can be made for the case when the size limit is set to 2400. This time, the cut point of the ε parameter is 0.004 instead. We note that for all the cases, CM occupies less space than LP. This is shown in Figure 6(c). Specifically, the relative sizes of CM over LP with ε equal to 0.003 and 0.004 are about 73% and 55%, respectively.

In summary, the experiment results indicate that

- The bloom histogram is the most accurate estimation method under small space budgets.
- Reasonable accuracy for dynamic XML data can be achieved by choosing an appropriate dynamic summary and building bloom histograms periodically.

6 Related work

6.1 Statistics estimation for XML data

There has been some recent work to estimate the selectivity of path expressions for XML data. Most existing approaches solve the problem in a *top-down* manner, by capturing the full structure of the XML data tree (or graph) with a small-sized synopsis structure and pruning it until a given space constraint is satisfied. [1] proposed two techniques, namely path trees and Markov tables. The path tree approach starts with a trie of all sub-paths and their associated counts; The Markov table approach starts with the full path-selectivity table of paths length up to m. A set of greedy pruning rules are proposed, and they differ mainly in the amount of information preserved in the pruning process. XPathLearner [14] used the Markov Histogram, an variant of Markov table which additionally captures value distribution. Their focus is on the learning methods of the Markov Histogram though. XSketch [17] exploited localized graph stability in a graph-synopsis model to approximate path and branching distributions in an XML data graph. Its successor, XSketches [18], integrates support for value constraints as well, by using multidimensional synopsis to capture value correlations. For tree structured data, XSketch synopsis is just a path tree in [1]. [9] proposed StatiX, which takes advantage of XML schema types and builds both structural and value histograms as statistical summaries. However, the effectiveness of StatiX is highly dependent on the quality of system-generated OIDs. [6] proposed correlated subpath tree (CST), which is a pruned suffix tree with set hashing signatures that help to determine the correlation between branching paths when estimating the selectivity of twig queries. CST is usually large in size and has been outperformed by [1] for simple path expressions.

6.2 Traditional statistics estimation methods

Histogram is one of the most important statistics estimation data structure in relational DBMSs. [12] offers a latest, comprehensive survey on this subject. In particular, V-optimal histogram was proposed as the optimal histogram under the *sum of square error* metric [13]. Its optimal construction algorithms were also presented.

Maintaining histogram for dynamic data is a hard task. For one-dimensional histograms, [5] proposed to recompute the histogram periodically or under request by using samples from the base relation. [10] and [11] proposed to maintain a secondary data structure, *backing sample*, and a sketch, respectively, from which histograms can be recomputed. Similar approaches are adopted for maintaining multidimensional histograms, as in [20]. Another fundamentally different approach to construct and maintain histograms for dynamic data is to build histograms solely from query feedback, without looking at the data [4, 19, 15]. They are usually termed dynamic histograms or self-tuning histograms.

6.3 Bloom filter

Bloom filter was first proposed in [2] as a space efficient data structure for answering approximate membership queries over a given set. It was used in Bloom Join in [16]. [3] is a recent survey of various applications of bloom filters in the network domain. Most recently, a spectral bloom filter was proposed that generalize the original bloom filter to answer queries regarding the multiplicity of an element for a multiset [7]. Our bloom histogram can be viewed as a further compressed spectral bloom filter, which occupies even much smaller space.

7 Conclusion

In this paper, we studied the problem of dynamically estimating the selectivity for XML path expressions. We proposed a two-layer solution consisting of an estimator component and a dynamic summary component. A novel *bloom histogram* was proposed as a compact yet highly accurate estimator for XML path expressions. One unique feature is that the estimation error can be theoretically bounded by its size. We investigated both approximate and exact forms of the dynamic summary and discussed how they assist in updating the bloom histogram. Our extensive experiment results demonstrated the effectiveness of our proposed solution compared to the previous methods under both static and dynamic environments.

This paper presents our first endeavor towards solving the selectivity estimation problem for general XML queries. Some future work can be pursued hereafter. We are investigating the appropriate way to generalize our method to support other types of path expressions (e.g., with value predicates and branching subexpressions). On the other hand, we are also seeking opportunities to apply the bloom histogram method in other database applications.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, pages 591–600, 2001.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Proceeding of Allerton Conference*, 2002.

[4] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*.

[5] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random sampling for histogram construction: How much is enough? In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD 1998)*, pages 436–447.

[6] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001)*, pages 595–604, 2001.

[7] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, pages 241–252.

[8] G. Cormode and S. Muthukrishnan. Improved data stream summaries: The count-min sketch and its applications (extended abstract). In *Latin American Theoretical Informatics 2004 (LATIN 2004)*, 2004.

[9] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: making XML count. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 181–191, 2002.

[10] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997)*.

[11] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing (STOC 2002)*.

[12] Y. E. Ioannidis. The history of histograms (abridged). In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003)*, pages 19–30.

[13] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proceedings of 24rd International Conference on Very Large Data Bases (VLDB 1998)*, pages 275–286.

[14] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. XPathLearner: An on-line self-tuning Markov histogram for XML path selectivity estimation. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 442–453, 2002.

[15] L. Lim, M. Wang, and J. S. Vitter. SASH: A self-adaptive histogram set for dynamically changing workloads. In *Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003 (VLDB 2003)*.

[16] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD 1986)*, pages 84–95, 1986.

[17] N. Polyzotis and M. N. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 358–369, 2002.

[18] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for XML data graphs. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 466–477, 2002.

[19] L. Qiao, D. Agrawal, and A. E. Abbadi. RHist: adaptive summarization over continuous data streams. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2002)*.

[20] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*.