CXHist : An On-line Classification-Based Histogram for XML String Selectivity Estimation

Lipyeow Lim[1] Min Wang[1] Jeffrey Scott Vitter[2]

[1] IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532, USA
{liplim,min}@us.ibm.com

[2] Purdue University
150 N. University Street
West Lafayette, IN 47907, USA
jsv@purdue.edu

Abstract

Query optimization in IBM's System RX, the first truly relational-XML hybrid data management system, requires accurate selectivity estimation of path-value pairs, i.e., the number of nodes in the XML tree reachable by a given path with the given text value. Previous techniques have been inadequate, because they have focused mainly on the tag-labeled paths (tree structure) of the XML data. For most real XML data, the number of distinct string values at the leaf nodes is orders of magnitude larger than the set of distinct rooted tag paths. Hence, the real challenge lies in accurate selectivity estimation of the string predicates on the leaf values reachable via a given path.

In this paper, we present CXHist, a novel workload-aware histogram technique that provides accurate selectivity estimation on a broad class of XML string-based queries. CXHist builds a histogram in an on-line manner by grouping queries into buckets using their true selectivity obtained from query feedback. The set of queries associated with each bucket is summarized into feature distributions. These feature distributions mimic a Bayesian classifier that is used to route a query to its associated bucket during selectivity estimation. We show how CXHist can be used for two general types of $\langle path, string \rangle$ queries: exact match queries and substring match queries. Experiments using a prototype show that CXHist provides accurate selectivity estimation for both exact match queries and substring match queries.

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

1 Introduction

In IBM's System RX [19], XML data are stored natively in a tree data structure. XML data are queried using the SQL/XML and XQuery/XPath query languages. A *path or path expression* is the most fundamental addressing unit for locating node(s) in an XML tree. Paths are processed using either indexes or tree traversals. For efficient processing, complex path expressions in XML queries are preprocessed into a set of candidate $\langle path, pred \rangle$ query pairs, where $path$ is a linear rooted path and $pred$ is a string predicate on the leaf value reachable via $path$. Consequently, an XML query (such as XQuery) can be mapped to several retrieval operations using $\langle path, pred \rangle$ query pairs [29, 25, 30, 13]. For the example data in Figure 1,

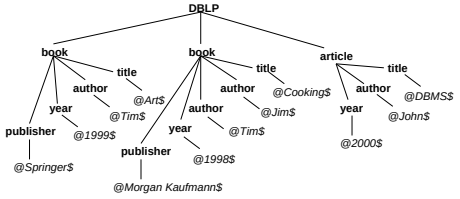

Figure 1: An example XML data tree. Tag names are in bold and data values are in italics.

the path expression `//author=@Tim$`[1] can be mapped to query pairs $\langle$`/DBLP/book/author`, `@Tim$`$\rangle$ and $\langle$`/DBLP/article/author`, `@Tim$`$\rangle$, where the string predicate is the exact match predicate. These retrieval operations using
$\langle path, pred \rangle$ query pairs form the set of basic logical query processing operators. Accurate estimation of the selectivity of such $\langle path, pred \rangle$ query pairs is therefore crucial for choosing an optimal execution plan in cost-based query optimization.

The XML string selectivity estimation problem is defined as follows: Given a $\langle path, pred \rangle$ query, where $pred$ is a string predicate, estimate the number of nodes in the

[1]The symbols '@' and '$' denote the start and end of a string respectively.

XML data that are reachable by $path$ and whose associated string values satisfy the string predicate $pred$. The focus of this paper is on the string predicate portion of a $\langle path, pred \rangle$ query, because the problem of estimating the selectivity of the path portion of a $\langle path, pred \rangle$ query is already well addressed in the literature [1, 5, 20, 21, 16, 9, 28, 27]. Examples of queries with different string predicates include exact match queries and substring queries. An *exact match query* is specified by a $\langle path, string \rangle$ pair and retrieves all nodes reachable via $path$ whose string value matches the query string exactly. A *substring query* is specified by a $\langle path, substring \rangle$ pair and retrieves all nodes reachable via $path$ whose string value contains $substring$. For the XML tree in Figure 1, the selectivity of exact match query $\langle$`/DBLP/book/author`, `@Tim$`$\rangle$ is 2 and the selectivity of substring query $\langle$`/DBLP/book/author`, `im$`$\rangle$ is 3.

The XML string selectivity estimation problem is an important and non-trivial problem. The string aspect of the problem is especially important to XML DBMSs for two reasons. First, XML data are weakly-typed and therefore XML databases must support the retrieval of XML data as string data, even though some data could be treated as numeric data as well. For numeric leaf values, traditional histograms can be used for selectivity estimation [21, 9]. Second, the set of distinct path-string pairs in an XML tree is typically orders of magnitude larger than the set of distinct rooted paths. For example, the DBLP XML data set has 2,026,814 distinct path-string pairs, but only 275 distinct rooted paths. The small number of paths means that the selectivity of all distinct rooted paths can be easily stored in an index using little space[2]. The selectivity of distinct path-string pairs, however, would require more sophisticated approximation techniques.

XML string selectivity estimation is a non-trivial problem because of three unique challenges:

1. How to capture substring statistics accurately in limited storage?
2. How to capture the correlation between paths and the substring statistics accurately?
3. How to support a broad class of query types?

Capturing substring statistics for selectivity estimation in relational databases has been studied in [14, 26, 12, 10, 11, 4]; however, the accuracy of these proposed solutions is far from satisfactory [4]. The problem is hard because the set of distinct strings is large and the corresponding set of possible substrings is even larger (by orders of magnitude). Storing the selectivity of each distinct string or substring is clearly infeasible and more sophisticated approximation techniques are required to capture the substring statistics.

The main difference between the XML string selectivity estimation problem and the relational substring selectivity estimation problem is that a correlated path (whether implicitly encoded as a path ID or explicitly as a XPath) is associated with the query substring in the XML problem. Hence, XML string selectivity estimation is a harder problem than relational substring selectivity estimation, because the correlation between path and substring statistics needs to be captured as well. Previous work on XML selectivity estimation has emphasized the selectivity of navigational paths and tree structures [1, 5, 20, 21, 16, 9] and has not fully address the XML string selectivity estimation problem. While correlated sub-path trees (CSTs) [5] and XPathLearner [16] do address the correlation between paths and their associated string values in a limited way, XPathLearner does not capture substring statistics and CSTs suffer from the underestimation problem [4]. The problem of accurately capturing substring statistics that are correlated with paths has not been adequately addressed.

As a consequence of the flexibility of XML, XML query languages, and XML applications, XML DBMSs need to support a broad class of query types (exact match, substring, prefix, suffix, etc.). Two particularly important query types are exact match queries and substring queries. Exact match queries occur frequently in transaction processing and in application generated queries. Substring queries, on the other hand, occur frequently in user generated queries, because users do not usually remember a text string exactly. While separate statistics and techniques can be used for each query type, we propose a single estimation framework that is accurate for a broad class of query types in this paper. Previous work that supports string predicates is very restrictive in this aspect. The CST method [5] supports substring queries, but is prone to underestimation for exact match queries [4]. The XPathLearner [16] method supports exact match queries, but cannot handle substring queries at all. The challenge then lies in designing a single framework for the accurate estimation of the selectivity of a broad class of string predicates.

Our contributions. In this paper, we propose CXHist as a novel on-line selectivity estimation method that supports a broad class of query types. CXHist is capable of capturing accurate path-correlated statistics for a broad class of string predicates on leaf values. These statistics are collected not from costly off-line scans of the underlying data, but from *query feedback*, i.e., past query answers (see Figure 2). Consequently CXHist is able to adapt to both changes in the query workload characteristics and in the underlying data. CXHist stores the mapping between queries and their selectivity using a histogram approximately. The selectivities are partitioned or quantized into a fixed number of buckets and the set of queries associated with each bucket is summarized into feature distributions using a query model. Feature distributions therefore approximately 'remember' which queries belong to which bucket. Given a query, the feature distributions mimic a Bayesian classifier [8] so as to compute the bucket that the query belongs to. The selectivity of the bucket (computed by the Bayesian classifier) is then returned as the estimated selectivity. We show how

[2] In the case that the number of distinct rooted paths is significantly large, techniques based on the Markov assumption and/or bi-similarity can be used [1, 5, 20, 21, 16].

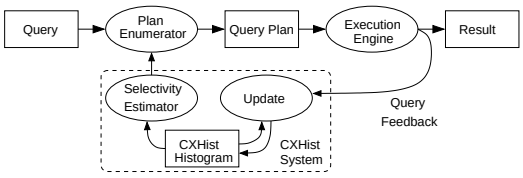

Figure 2: Workflow of CXHist.

to construct and maintain the feature distributions and the buckets using only query feedback. Our contributions are summarized below:

- CXHist is a new type of histogram that uses feature distributions and Bayesian classification techniques to capture the mapping between queries and their selectivity. To the best of our knowledge, CXHist is the first type of histogram based on classification.
- CXHist is on-line: it gathers statistics from query feedback rather than from costly data scans, and hence adapts to changes in workload characteristics and in the underlying data. We also show how to build and update a CXHist using only query feedback. With the exception of [16], all previous work is off-line.
- CXHist is general. By defining an appropriate query model CXHist is applicable to a broad class of query types. CXHist is not limited to XML data: it can be used for multidimensional string data in relational databases as well.
- CXHist can be easily implemented and deployed in practice.

The rest of the paper is organized as follows. Section 2 presents CXHist histograms in detail. Experimental validation is given in Section 3. Related work is discussed in Section 4. In Section 5 we draw conclusions and discuss future work. For ease of exposition, we present CXHist in the context of exact match and substring queries, even though CXHist is applicable to a broad class of query types. Note also that the terms cardinality and selectivity are used interchangeably in this paper.

2 CXHist Histograms

2.1 Overview

A key difference between our on-line CXHist technique and traditional off-line histograms is that CXHist models queries whereas the off-line methods model data. Modelling queries has the advantage that the relevance of the statistics collected and stored in the limited amount of space can be tuned to the query workload: resources are not wasted on storing statistics that are not relevant to the query workload. CXHist models a query q as a set of features $\vec{x} = \langle x_1, \ldots, x_k \rangle$. How a query is mapped to its corresponding feature set is determined by the *query model* (details in Section 2.2).

A CXHist histogram consists of a set of m buckets indexed by $\mathcal{B} = \{1, 2, \ldots, m\}$. Each bucket b stores

1. $sum(b)$, the sum of the selectivities of all the query feedback that is associated with bucket b,
2. $cnt(b)$, one plus the number of query feedback seen so far that is associated with bucket b,
3. $\{P(X_i|B{=}b) : i = 1, \ldots, k\}$, a set of query feature probability distributions that captures approximately the set of queries that are associated with bucket b. One distribution is stored for each feature random variable X_i.

The CXHist histogram is usually initialized with empty feature distributions and some predetermined representative values for the $sum(b)$ fields. Upon receiving query feedback, CXHist updates the feature distributions, $sum(b)$, and $cnt(b)$ fields, thereby learning and refining the statistics using query feedback.

Given a query q, CXHist estimates the selectivity $\sigma(q)$ by first computing the bucket $\hat{b}$ that is associated with q. This computation is performed by using the query model to map the query into feature space and by using the feature distributions to mimic a Bayesian classifier. The selectivity of q is then computed as

$$est(\hat{b}) = \frac{sum(\hat{b})}{cnt(\hat{b})}. \quad (1)$$

The feature distributions of a bucket b approximately "remembers" which queries belong to bucket b. Each feature distribution $P(X_i{=}x_i|B{=}b)$ is not stored as probabilities, but using counts[3], $N(X_i{=}x_i, B{=}b)$, the number of times a particular feature has been observed to be associated with bucket b. The probabilities are recovered using,

$$P(X_i{=}x_i|B{=}b) = \frac{N(X_i{=}x_i, B{=}b)}{\sum_\xi N(X_i{=}\xi, B{=}b)}. \quad (2)$$

CXHist updates itself using query feedback. Suppose the query execution engines computes $\sigma(q)$ as the true selectivity for query q. CXHist finds the bucket $b*$ whose selectivity is closest to $\sigma(q)$. If the bucket $\hat{b}$ computed using feature distributions is not the same as $b*$, the feature distributions of bucket $b*$ are updated so that CXHist will remember that query q should be associated with bucket $b*$.

The update process potentially adds new entries to the feature distributions and hence increases the memory used by CXHist. Whenever CXHist exceeds a given size threshold, the feature distributions are pruned so that the size of CXHist stays below the given threshold. Since the size threshold is set to a small constant (in practice this constant is roughly 1-2% of the data size), the space complexity of CXHist is $O(1)$. The time complexity of the selectivity estimation, update and pruning procedures are therefore also $O(1)$ because their time complexity are only dependent on the size of the CXHist histogram.

[3]Note that fractional counts are allowed for updates.

2.2 Modelling Queries

CXHist is a general histogram technique that can be used whenever a query model can be defined. A query model $\mathcal{M}(q)=\vec{x}$ is a mapping from a query to a feature set. CXHist uses a query model to compress and summarize the set of queries associated with each bucket into feature distributions. A good query model should include features that will distinguish queries associated with different buckets with respect to the classification procedure. Query models are application-specific and we first describe the query model used in this paper, before giving an example of a query model that can be used in a different application scenario.

Since we are mainly interested in substring and exact match queries in this paper, the query model we used models these two types of queries. Our query model maps each query to a pathid and all the n-grams of the query (sub)string, where n is a tunable parameter. The distinction between substring and exact match queries are implicitly encoded using a start of string symbol '@' and an end of string symbol '$'. For example, an exact match query (5, `@LIM$`) is mapped to $\langle 5$, `@L`, `LI`, `IM`, `M$`$\rangle$ for $n = 2$.

To illustrate the generality of CXHist, we briefly outline an example of a query model that can be used for single-branch queries. Suppose the query is `/DBLP/book[year=@1999$]/title=@Ar`, i.e., retrieve the titles of books published in 1999 prefixed by 'Ar'. A possible query model would include the following features: the main path `/DBLP/book/title`, the main substring predicate `@Ar`, the branching path `/DBLP/book/year`, the branching path string predicate `@1999$`. The (sub)string predicates can be further modeled using n-grams.

2.3 Estimating Selectivity

Given a query q, selectivity estimation is done by determining which bucket the query belongs to. The mapping between a query and its bucket is modelled as a joint probability distribution with the random variable B for the bucket and a series of random variables, $\vec{X}=\langle X_1, \ldots, X_k\rangle$, for the features of the query (under a given query model). Suppose that the query model maps the query q into a set of features $\vec{x} = \langle x_1, \ldots, x_k\rangle$, then the bucket $\hat{b}$ that q belongs to is computed as the bucket that maximizes the posterior probability of the query belonging to a bucket b given its feature values,

$$\hat{b} = \arg\max_{b\in\mathcal{B}}\left\{P(B{=}b|\vec{X}{=}\vec{x})\right\} \quad (3)$$

$$= \arg\max_{b\in\mathcal{B}}\left\{P(B{=}b)P(\vec{X}{=}\vec{x}|B{=}b)\right\}, \quad (4)$$

where the second equality is obtained using Bayes rule and by simplifying the denominator into a constant that is independent of b and hence does not affect the maximal point. This technique is commonly known as a Bayesian classifier [8] (where each bucket is the class label) in machine learning literature. A *naive Bayesian classifier* further simplifies the conditional distribution $P(\vec{X}{=}\vec{x}|B{=}b)$ by assuming conditional independence among the features given the bucket (the *naive Bayes assumption*),

$$P(X_1, X_2, \ldots, X_n|B{=}b) = \prod_{i=1}^{n} P(X_i|B{=}b). \quad (5)$$

Note that assuming conditional independence of the feature distributions is not the same as

1. assuming attribute value independence in RDBMS, or
2. assuming that the path and value string are independent with respect to the $\langle path, string\rangle$ pair selectivity, or
3. assuming conditional independence of n-grams in the pruned suffix tree methods of [12].

The underlying probability space in each of the above three cases is different from the CXHist context. Despite the conditional independence assumption on the features, naive Bayesian classifiers have been shown to be very accurate even for datasets with strong dependencies among the features [7, 15]. Observe in Equation (5) that the probability terms $\{P(X_i|B{=}b) : \forall i\}$ are precisely those stored in each bucket and that the probability term $P(B{=}b)$ can be computed as,

$$P(B{=}b) = \frac{cnt(b) - 1}{\sum_{b'}(cnt(b') - 1)} \quad (6)$$

Once the bucket $\hat{b}$ for query q is computed, the selectivity of q is then computed as $est(\hat{b})$. There is a boundary case when the posterior distribution is flat and no maximum point exists. In this case, a default selectivity such as the smallest selectivity among the buckets can be returned.

Using the query model described in Section 2.2, each substring or exact match query is modelled as a random variable T for the pathID and a series of random variable $\{G_1, G_2, \ldots, G_k\}$ for the sequence of n-grams of the query (sub)string. In addition to conditional independence, we assume stationarity of the n-gram distribution,

$$P(G_i|B) = P(G|B) \quad \forall i. \quad (7)$$

The n-gram model and the stationarity assumption were first used by Shannon [24] for predicting English text and has since been widely used in text modeling. Stationarity allows us to store just one n-gram distribution per bucket instead of one distribution for each position in a string. Hence, given a query with feature values $\langle t, g_1, g_2, \ldots, g_k\rangle$, its associated bucket is computed as,

$$\hat{b} = \arg\max_{b\in\mathcal{B}}\left\{P(B{=}b) \times P(T{=}t|B{=}b) \times \prod_{i=1}^{k} P(G{=}g_i|B{=}b)\right\}, \quad (8)$$

and the selectivity of q is estimated as $est(\hat{b})$.

2.4 Initializing CXHist

Given a fixed query model, a fixed number of buckets m, the minimum selectivity l, and the maximum selectivity h, a CXHist histogram is initialized with m buckets, such that, for each bucket b, the bucket count $cnt(b)$ is set to one, the feature distributions are set to empty, and the $sum(b)$ is initialized with a *representative* value from the interval $[l, h]$. The set of representative values can be chosen in several ways. If a sample query workload is available during initialization, the MaxDiff [22] algorithm or the Lloyd-Max algorithm [17, 18] can be used. If a query workload is not available, the representative values can be picked uniformly from the given interval, or they can be picked to be exponentially increasing (eg. $\{1, 2, 4, 8, 16, \ldots\}$). In our experiments we used a combination of both ways. If the number of buckets m is larger than $\log_2(h-l)$, the smallest j representative values, where j is a tunable parameter, can be exponentially increasing and the remaining values uniformly spaced in the rest of the interval.

$$sum(b) = \begin{cases} l \times 2^{b-1} & b \leq j \\ l \times 2^{j-1} + (b-j)\frac{h-l\times 2^{j-1}}{m-j} & j < b < m \end{cases} \quad (9)$$

For example, if m=10 and j=5, the representative values for the interval $[1, 66]$ are $\{1, 2, 4, 8, 16, 26, 36, 46, 56, 66\}$.

Picking exponentially increasing representative values for the smaller selectivities gives better estimates in terms of relative error. The intuition is that if CXHist remembers which queries belong to which bucket perfectly and if CXHist does not update the representative values in the buckets, then the relative error of each estimate is no more than 50%.

2.5 Updating the CXHist Histogram

Given a query feedback, i.e. a $\langle q, \sigma(q)\rangle$ pair, we want to update the histogram so that it will give a better estimate when it encounters query q again. For our discussion assume that query q maps to the feature vector $\vec{x} = \langle x_1, \ldots, x_k\rangle$. The update algorithm is outlined in Algorithm 2 and it requires a subroutine ADDINSTANCE($b, \langle x_1, \ldots, x_k\rangle$) outlined in Algorithm 1 that increments all the feature distribution entries associated with each given feature value x_i in the given bucket b.

Algorithm 1 ADDINSTANCE ($b, \langle x_1, \ldots, x_k\rangle$)

INPUT: *bucket index b and query feature vector* $\langle x_1, \ldots, x_k\rangle$

```
1: for all i ∈ {1,...,k} do
2:   if ∄ entry N(X_i=x_i|B=b) then
3:     add new entry N(X_i=x_i|B=b) = 0
4:   increment N(X_i=x_i|B=b)
```

First, find the bucket $b*$ whose estimate $est(b*)$ is closest to $\sigma(q)$ (see line 1). Second, update the bucket $b*$ by incrementing $cnt(b*)$ and adding $\sigma(q)$ to $sum(b*)$ (lines 2-3). Third, update the feature distributions of bucket $b*$. If either no maximum point is found when estimating the selectivity of q or the closest bucket $b*$ is the same as the bucket that maximizes the posterior probability $\hat{b}$, i.e., no classification error, then increment the count

Algorithm 2 UPDATE ($\langle q, \sigma(q)\rangle, \langle x_1, \ldots, x_k\rangle, \gamma$)

INPUT: *query feedback* $\langle q, \sigma(q)\rangle$, *feature vector* $\langle x_1, \ldots, x_k\rangle$ *for query* q, *learning rate* γ

```
1: b* ← arg min_b{|σ(q) − est(b)|}
2: sum(b*) ← sum(b*) + σ(q)
3: increment cnt(b*)
4: compute b̂ using Equation (8).
5: if ∄b̂ or b*=b̂ then
6:   ADDINSTANCE(b*, ⟨x_1,...,x_k⟩)
7: else
8:   p̂ ← P(B=b̂)P(X⃗=x⃗|B=b̂)/P(B=b*)
9:   p* ← P(X⃗=x⃗|B=b*)
10:  if p*=0 then
11:    ADDINSTANCE(b*, {x_1,...,x_k})
12:    p* ← P(X⃗=x⃗|B=b*)
13:  while p* < p̂ do
14:    for all i ∈ {1,...,k} do
15:      update N(X_i=x_i|B=b*) using Equation (13)
16:    recompute p*
17:  if p* = p̂ then
18:    ADDINSTANCE(b*, ⟨x_1,...,x_k⟩)
```

$N(X_i{=}x_i|B{=}b*)$ for each $i = 1, \ldots, k$ (lines 5-6). Otherwise, there is a classification error, i.e., $b* \neq \hat{b}$. We want to update the feature distributions such that q will be classified into the bucket $b*$ instead of $\hat{b}$. Specifically, the desired condition is

$$P(B{=}b*)P(\vec{X}{=}\vec{x}|B{=}b*) > P(B{=}\hat{b})P(\vec{X}{=}\vec{x}|B{=}\hat{b}) \quad (10)$$

$$\equiv P(\vec{X}{=}\vec{x}|B{=}b*) > \frac{P(B{=}\hat{b})P(\vec{X}{=}\vec{x}|B{=}\hat{b})}{P(B{=}b*)}. \quad (11)$$

A naive method for updating the feature distributions is to keep incrementing the count $N(X_i{=}x_i|B{=}b*)$ for each $i = 1, \ldots, k$ until the above condition is met.

A more principled method for getting the same effect is to perform *gradient descent* [23] until the condition is met (line 13). The number of iterations can be bounded by a specified constant to prevent the `while`-loop (line 13) from taking too long. For gradient descent, we define the error function as,

$$E(\vec{x}) = \left[P(\vec{X}{=}\vec{x}|B{=}b*) - \hat{p}\right]^2, \quad (12)$$

where $\hat{p}{=}P(B{=}\hat{b})P(\vec{X}{=}\vec{x}|B{=}\hat{b})/P(B{=}b*)$ is a constant target value. Let w_i be a shorthand for the count $N(X_i{=}x_i, B{=}b*)$. For each feature value x_i of the query, the gradient descent method updates the count w_i using the negative gradient,

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - \gamma\frac{\partial E(\vec{x})}{\partial w_i}, \quad (13)$$

where $w_i^{(t)}$ and $w_i^{(t+1)}$ are the counts before and after the update, and γ is the learning rate parameter.

Two boundary cases need to be addressed. The first boundary case occurs before the gradient descent iteration when $w_i{=}0$ for some i and hence $P(\vec{X}{=}\vec{x}|B{=}b*){=}0$ and the gradient may be undefined. This situation is avoided by

incrementing $N(X_i{=}x_i, B{=}b*)$ for $i = 1, \ldots, k$ (lines 10-12). The other boundary case occurs after the gradient descent loop when $P(\vec{X}{=}\vec{x}|B{=}b*) = \hat{p}$. The desired condition is $P(\vec{X}{=}\vec{x}|B{=}b*) > \hat{p}$, so the feature distributions in $b*$ are incremented once more to break the tie (line 17).

One problem with using gradient descent in this form is that the updates $\Delta w_i = \frac{\partial E(\vec{x})}{\partial w_i}$ can be very small (less than 10^{-10}) when the number of feature values are large. We address this problem by computing the updates Δw_i for all i and normalizing them so that the smallest Δw_i is one. The learning rate is then applied onto the normalized updates.

We have discussed gradient descent update for CXHist without assuming a specific query model. We now show the update equations for the query model we adopted for substring and exact match queries. To simplify the notation, we will use the following shorthand,

$$\begin{aligned} p(t, g_1, \ldots, g_k) =& P(T{=}t|B{=}b*) \prod_{i=1}^{k} P(G{=}g_i|B{=}b*), \\ w_x =& N(X{=}x, B{=}b*), \\ W_X =& \sum_x N(X{=}x, B{=}b*), \end{aligned}$$

where X is a feature random variable (either T or G). The squared error function for gradient descent is,

$$E(t, g_1, \ldots, g_k) \quad = \quad \Big[p(t, g_1, \ldots, g_k) \; - \; \hat{p}\Big]^2. \quad (14)$$

For the pathID distribution, the update equation is,

$$\begin{aligned} \Delta w_t = \frac{\partial E(t, g_1, \ldots, g_k)}{\partial w_t} = 2\Big[p(t, g_1, \ldots, g_k) - \hat{p}\Big] \\ \times p(t, g_1, \ldots, g_k)\left(\frac{1}{w_t} - \frac{1}{W_T}\right). \quad (15) \end{aligned}$$

For the n-gram distribution, recall that we have assumed stationarity and that $\{g_1, \ldots, g_k\}$ are not necessarily all distinct; hence, for each distinct n-gram g that occurs α times in $\{g_1, \ldots, g_k\}$, the update in w_g is computed as,

$$\begin{aligned} \Delta w_g = \frac{\partial E(t, g_1, \ldots, g_k)}{\partial w_g} = 2\Big[p(t, g_1, \ldots, g_k) - \hat{p}\Big] \\ \times p(t, g_1, \ldots, g_k)\left(\frac{\alpha}{w_g} - \frac{k}{W_G}\right). \quad (16) \end{aligned}$$

2.6 Pruning CXHist

The size of a CXHist histogram is the sum over each bucket b of the size of the $sum(b)$ field, the $cnt(b)$ field and the feature distributions $\{N(X_i|B{=}b)\}$. The initialization and selectivity estimation procedures do not increase the size of the histogram. The update procedure potentially adds new entries to the feature distributions. After some number of updates, the CXHist histogram might need to be pruned so that it does not exceed the memory allocated to it.

Pruning is governed by two parameters both in units of bytes: the $triggersize$ and the $targetsize$. Whenever the size of the histogram exceeds $triggersize$ bytes, the pruning procedure is activated and the CXHist histogram is pruned down to $targetsize$ bytes.

Pruning is achieved by discarding entries with small counts in the feature distributions. All feature distribution entries $\{N(X_i{=}x_i|B{=}b) : \forall i, x_i, b\}$ are sorted by magnitude and we keep discarding the smallest entry until the desired target size is achieved. Mathematically, discarding an entry is the same as setting the magnitude of that entry to zero. The entries with the smaller counts are good candidates for pruning because (1) they are likely to be less frequently used, otherwise the update procedure would have incremented their counts, and (2) they are less likely to affect the maximal point in the selectivity estimation computation, since their probabilities are already small.

2.7 Example

Consider the following query workload:

No.	Path	String	Selectivity
1	/x/y	@LIM$	2
2	/x/z	@MIN	20
3	/x/y	@LIM	10
4	/x/y	@LIM$	2
5	/x/y	IM	18

Given that the minimum and maximum selectivity are 1 and 20 respectively, and that 5 buckets are to be used, we initialize the representative values as $1, 2, 4, 8, 16$. The $cnt(b)$ field for each b is set to 1. The query model maps each query into a path random variable T, and a series of 2-gram random variables G_i. Assuming stationarity we store only a single 2-gram distribution per bucket. The feature distributions are initially empty. We adopt the following shorthand for computing the posterior probability for a given query,

$$\begin{aligned} L(b) =& \quad P(B{=}b)P(T{=}t|B{=}b) \prod_{i=1}^{k} P(G{=}g_i|B{=}b), \\ \hat{p} =& \quad \frac{P(B{=}\hat{b})}{P(B{=}b*)} P(T{=}t|B{=}\hat{b}) \prod_{i=1}^{k} P(G{=}g_i|B{=}\hat{b}) \\ p* =& \quad P(T{=}t|B{=}b*) \prod_{i=1}^{k} P(G{=}g_i|B{=}b*) \end{aligned}$$

Query 1. CXHist gives a default estimate of 1 (50 % relative error) for the first query because all the feature distributions are empty and no maximal point exists. CXHist receives the true selectivity 2 as feedback and updates itself. The bucket with the closest estimate is bucket $b*{=}2$ and we update bucket 2 by adding 2 to $sum(2)$ and incrementing $cnt(2)$. The feature distributions are initially empty, so ADDINSTANCE is called and the resulting CXHist is:

B	$sum(b)$	$cnt(b)$
1	1	1
2	4	2
3	4	1
4	8	1
5	16	1

B	G	$N(G,B)$
2	@L	1
2	LI	1
2	IM	1
2	M$	1

B	T	$N(T,B)$
2	/x/y	1

Query 2. The second query is processed similarly and the resulting feature distributions are shown as follows:

B	$sum(b)$	$cnt(b)$
1	1	1
2	4	2
3	4	1
4	8	1
5	36	2

B	G	$N(G,B)$
2	@L	1
2	LI	1
2	IM	1
2	M$	1
5	@M	1
5	MI	1
5	IN	1

B	T	$N(T,B)$
2	/x/y	1
5	/x/z	1

Query 3. For the third query, only bucket 2 gives a non-zero posterior probability,

$$\begin{aligned} L(2) = \; & P(B{=}2) \times P(T{=}/x/y|B{=}2) \times P(G{=}\text{@L}|B{=}2) \\ & \times P(G{=}\text{LI}|B{=}2) \times P(G{=}\text{IM}|B{=}2) \\ = \; & \frac{1}{2} \times 1 \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \quad = \quad 0.007813, \end{aligned}$$

therefore the selectivity is estimated as $est(2) = 2$ with a relative error of 400 %.

Using the query feedback, the bucket with the closest estimate is $b* = 4$ and $cnt(4)$ and $sum(4)$ are updated accordingly. Since the feature distributions in bucket 4 are empty, we apply the subroutine ADDINSTANCE once and check if gradient descent is required,

$$\begin{aligned} \hat{p} = \; & \frac{1/3}{1/3} \times 1 \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} = 0.015625, \\ p* = \; & 1 \times \frac{1}{3} \times \frac{1}{3} \times \frac{1}{3} = 0.037037. \end{aligned}$$

Since $p* > \hat{p}$, no further updates are required . The resulting state of CXHist is:

B	$sum(b)$	$cnt(b)$
1	1	1
2	4	2
3	4	1
4	18	2
5	36	2

B	G	$N(G,B)$
2	@L	1
2	LI	1
2	IM	1
2	M$	1
4	@L	1
4	LI	1
4	IM	1
5	@M	1
5	MI	1
5	IN	1

B	T	$N(T,B)$
2	/x/y	1
4	/x/y	1
5	/x/z	1

Query 4. Only bucket 2 has a non-zero posterior probability,

$$L(2) = \quad \frac{1}{3} \times 1 \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} \times \frac{1}{4} = 0.001302, \quad (17)$$

so CXHist computes the estimate as $est(2) = 2$ with zero error. The update process applies the subroutine ADDINSTANCE once on the feature distributions of bucket 2. The resulting histogram is as follows.

B	$sum(b)$	$cnt(b)$
1	1	1
2	6	3
3	4	1
4	18	2
5	36	2

B	G	$N(G,B)$
2	@L	2
2	LI	2
2	IM	2
2	M$	2
4	@L	1
4	LI	1
4	IM	1
5	@M	1
5	MI	1
5	IN	1

B	T	$N(T,B)$
2	/x/y	2
4	/x/y	1
5	/x/z	1

Query 5. Computing the posterior probabilities,

$$\begin{aligned} L(1) = \; & L(3) = L(5) = 0 \\ L(2) = \; & \frac{2}{4} \times 1 \times \frac{2}{8} = 0.125 \\ L(4) = \; & \frac{1}{4} \times 1 \times \frac{1}{3} = 0.083333, \end{aligned}$$

CXHist concludes that bucket 2 maximizes the posterior probability and returns $est(2) = 2$ as the estimate (with 89 % relative error).

Using the query feedback, the bucket with the closest estimate is computed to be $b* = 5$, and $cnt(5)$ and $sum(5)$ are updated accordingly. Since $L(5) = 0$, the subroutine ADDINSTANCE is applied once. The resultant state of the CXHist histogram is shown below:

B	$sum(b)$	$cnt(b)$
1	1	1
2	6	3
3	4	1
4	18	2
5	54	3

B	G	$N(G,B)$
2	@L	2
2	LI	2
2	IM	2
2	M$	2
4	@L	1
4	LI	1
4	IM	1
5	@M	1
5	MI	1
5	IN	1
5	IM	1

B	T	$N(T,B)$
2	/x/y	2
4	/x/y	1
5	/x/z	1
5	/x/y	1

In order to determine whether gradient descent is required, the update procedure computes $\hat{p}$ and $p*$,

$$\hat{p} = \frac{2/5}{2/5} \times 1 \times \frac{2}{8} = 0.25, \quad p* = \frac{1}{2} \times \frac{1}{4} = 0.125.$$

Since $\hat{p} > p*$, gradient descent is required and we compute the deltas for $w_t{=}N(T{=}/x/y, B{=}5)$ and $w_q{=}N(G{=}\text{IM}, B{=}5)$ using Equation (15) and (16),

$$\begin{aligned} \Delta w_t &= 2 \times (0.125 - 0.4) \times 0.125 \times (1 - \frac{1}{2}) = -0.034 \\ \Delta w_q &= 2 \times (0.125 - 0.4) \times 0.125 \times (1 - \frac{1}{4}) = -0.052. \end{aligned}$$

Normalizing $(\Delta w_t, \Delta w_q)$ by the minimum absolute delta value, we have $(\Delta w_t{=}-1, \Delta w_q{=}-1.5)$. Using a learning rate of 1, CXHist updates the following,

$$\begin{aligned} N(T{=}\text{/x/y}, B{=}5) &\leftarrow N(T{=}\text{/x/y}, B{=}5) + 1, \\ N(G{=}\text{IM}, B{=}5) &\leftarrow N(G{=}\text{IM}, B{=}5) + 1.5. \end{aligned}$$

Recomputing $p*$, we have

$$p* = \frac{2}{3} \times \frac{2.5}{5.5} = 0.303030 > \hat{p},$$

and the gradient descent loop terminates.

3 Experiments

In this section, we evaluate the performance of CXHist under varying memory constraints and workload skew, and we also investigate the stability of CXHist under changes in the workload characteristics.

Data Set. We have used the DBLP XML data (as of October 2003). Because our main focus is on string predicates and also motivated by the fact that many XML DBMSs map rooted paths to path identifiers (pathIDs) for more efficient data storage and query processing [29, 25, 30, 13][4] our implementation also map all rooted paths into pathIDs. The XML data file is preprocessed into a collection of $\langle path, string, count \rangle$ triples by counting the occurrences of each distinct $\langle path, string \rangle$ pair. The total number of leaves is 5,045,240 of which there are 2,026,814 distinct $\langle path, string \rangle$ pairs.

Query workloads. For each experiment we have used three types of workloads: substring match workload, exact match workload, and a mixed workload. Each query is generated from the randomly permuted set of $\langle path, string, count \rangle$ triples by randomly picking a triple according to a discretized Gaussian distribution $G(n, \mu, s)$, where n is the number of triples, μ is the mean, and s is the standard deviation. For an exact match query, the pathid and text string from the chosen triple is returned. For a substring query, the pathid from the chosen triple and a substring token uniformly randomly picked from the text string of the chosen triple is returned. After all the queries of a workload have been generated, the workload is randomly permuted. Mixed workloads are generated by mixing and randomly permuting a substring workload and an exact match workload.

We present results from a subset (mean $\mu{=}992,942$) of the workloads we experimented with. Table 1 summarizes the important workload characteristics. The cardinality and size (in bytes) of the set of distinct queries in each workload (see Table 1) are important in allocating memory to a selectivity estimation method: If the allocated memory is large enough to store the set of distinct queries (and the corresponding selectivities), it forms a cache that can estimate selectivity with zero error. Figure 3 shows the typical distribution of query selectivities (sorted in descending order) of the three types of workloads. Note that these distributions are consistent with Zipf's Law for word frequencies [31]. Each workload will be referenced using its ID and a letter (s,e,m) denoting whether it is a substring, exact match, or mixed workload respectively. For example, workload IIm refers to the mixed workload generated with standard deviation 10,000.

Comparisons. CXHist will be compared with two other methods: the off-line pruned suffix tree method (PST) and the on-line compressed histogram method (CH). PST constructs one pruned suffix tree per pathID for the leaf values associated with the pathID. Note that this is the same as constructing one large suffix tree where the first level nodes contain the pathIDs as alphabets. The suffix trees are pruned by increasing threshold counts to obtained the desired size. CH is adapted from XPathLearner [16] and caches the top-k query feedback with the largest selectivity and aggregates the selectivity of the other query feedback into buckets according to the pathID and the first q letters of the leaf value. Whenever CH uses more than $triggersize$ bytes of memory, CH discards buckets in increasing order of selectivity until memory usage is less than $targetsize$ bytes.

Counting Memory. For CXHist, 4-byte integers are used for the $sum(b)$ and $cnt(b)$ fields. Each pathID distribution stores a list of 8-byte $\langle pathid, count \rangle$ pairs and each n-gram distribution stores a list of $(n{+}4)$-byte $\langle n\text{-gram}, count \rangle$ pairs. For the CH method with parameters k and q, the memory usage consists of the memory used to store the top k queries and their selectivity, and the memory used by the list of buckets. Each bucket stores a q-byte prefix string, and three 4-byte fields for the pathID, sum, and count. For the PST method, consult [14] for memory accounting.

Parameters. For both CXHist and CH the parameter $targetsize$ is set to be 10% lower than $triggersize$. The parameter $triggersize$ is set to be a fraction of $usize$, the size of the set of distinct queries in the workload. For CH, the parameter k needs to be a small fraction of $uniq$, the number of distinct queries in the workload, otherwise all the distinct queries will be cached. Where there is no confusion which workload is being used, the values for $triggersize$ and k will be specified as a percentage. For our experiments, we have set $k = 0.125 \times uniq$ or equivalently $k = 12.5\%$. The notation CH($k, q, triggersize$) denotes the CH method with parameters k, q and $triggersize$. For CXHist, the buckets are initialized with $nexp$ exponential values and the gradient descent update method with a learning rate of 1 is used. The maximum selectivity is set to half the maximum number of $\langle path, string \rangle$ pairs, i.e., 2,522,620. The notation CXHist($n, m, nexp, triggersize$) denotes the CXHist method with the following parameters:

[4] If the set of distinct paths is too large to be mapped to a set of pathIDs, a Markov model can be used to model the set of paths explicitly [1, 16] in the query model.

		substring			exact match			mixed		
ID	std. dev.	uniq	usize	size	uniq	usize	size	uniq	usize	size
I	100,000	3,279	56,449	87,825	4,964	221,426	223,020	8,243	277,875	310,841
II	10,000	3,337	57,283	87,302	4,687	207,945	222,033	8,024	265,228	309,331
III	1,000	2,712	46,877	87,584	2,808	125,285	223,184	5,520	172,162	310,764
IV	100	1,175	20,674	88,000	522	24,583	239,024	1,697	45,257	327,020
V	10	253	4,484	87,793	67	3,436	265,349	320	7,920	353,138

Table 1: Workload characteristics. The standard deviation of the Gaussian distribution used to generate the workload is given by the "std. dev." column. The number and size of the distinct queries in the workload is given by the "uniq" and "usize" fields. The "size" field gives the size of the workload in bytes. Each exact match and substring workload has 5000 queries per workload and each mix workload has 10,000 queries.

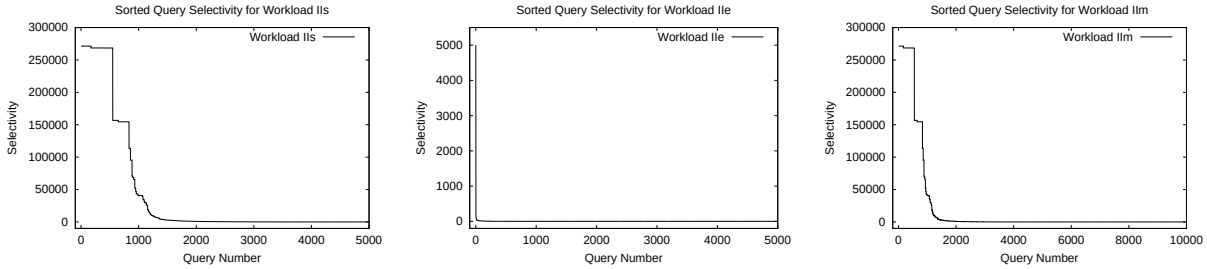

Figure 3: Sorted query selectivities of Workload IIs, IIe, and IIm (from left to right respectively).

n-gram features, m buckets, $nexp$ exponential representative bucket values and $triggersize$ as the pruning threshold. We have also studied the effect of the parameters n, m and $nexp$ on accuracy and a detailed discussion will be included in the full paper. For PST, we have used the independence strategy I_1 of [14] because we found it to be the most accurate in terms of average relative error among the different estimation strategies in [14, 12].

Metrics. We use the average relative error ($a.r.e.$) to measure the accuracy of the selectivity estimation methods. The $a.r.e.$ with respect to a set W of n queries is defined as

$$a.r.e. = \frac{1}{n} \sum_{q \in W} \frac{|\sigma(q) - \hat{\sigma}(q)|}{\sigma(q)}, \tag{18}$$

where $\sigma(q)$ is the selectivity of query q in the workload W and $\hat{\sigma}(q)$ is the corresponding estimated selectivity. The state of the selectivity estimation method is assumed to remain unchanged for all the queries in workload W. For on-line methods, the *on-line* $a.r.e.$ better reflects the performance of the on-line selectivity estimation method during deployment. The *on-line* $a.r.e.$ is defined as in (18), except that the selectivity estimation method is allowed to update itself in between queries. For an off-line method, the *on-line* $a.r.e.$ and the $a.r.e.$ are the same.

3.1 Accuracy vs Memory

In this experiment, we vary the amount of memory allocated to each method via the $triggersize$ parameter and measure the relative error performance for each workload. Note that $triggersize$ is set to a fraction ($\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1$) of the $usize$ of the workload (see Table 1), which in turn is less than 1% the size of the XML data (ca. 125 MB). For brevity, we present a set of representative results (Figure 4) using Workload II. The off-line PST method performs so poorly compared to CH and CXHist that we omit its results in the plots in order to use a more suitable scale on the y-axis. The results in Figure 4 compare the performance of CH and CXHist. Two prefix lengths (3 and 4) are used for the CH method and two n-gram lengths (3 and 4) are used for CXHist, because accuracy is dependent on both memory and the prefix/n-gram length. For substring and mixed workloads, CXHist is more accurate than CH. For exact match queries, CH is slightly (less than 1%) more accurate than CXHist, because the selectivity distribution of the exact match queries is more skewed than that of the substring queries (see Figure 3) and hence the top-k cache in CH captures almost all of the queries with large selectivities and the buckets need only capture the remaining selectivities which are uniformly small and close to one.

We further analyze the results for $triggersize$=25% by partitioning the selectivity estimates according to the true selectivity. Because small selectivities occur much more frequently in the data and consequently in the workload, we show the average relative error for the small selectivity partitions in Figure 5 and augment the figures with the fraction (normalized to a percentage) of the queries in the workload with the specified true selectivity. CH is not accurate for queries with small selectivities when the selectivity distribution in the workload is less skewed (for example, in the substring workload, see the first plot in Figure 5). The reason is that the selectivities of the queries not captured by the top-k cache is less uniform and cannot be captured by the buckets accurately. CXHist, on the other hand, seems to be consistently accurate for queries with small selectivities. For queries with very large selectivity, however, CH is more accurate than CXHist, because it caches the selectivity of the k queries with the largest selectivity. This is not a limitation in CXHist, because (1) queries with large selec-

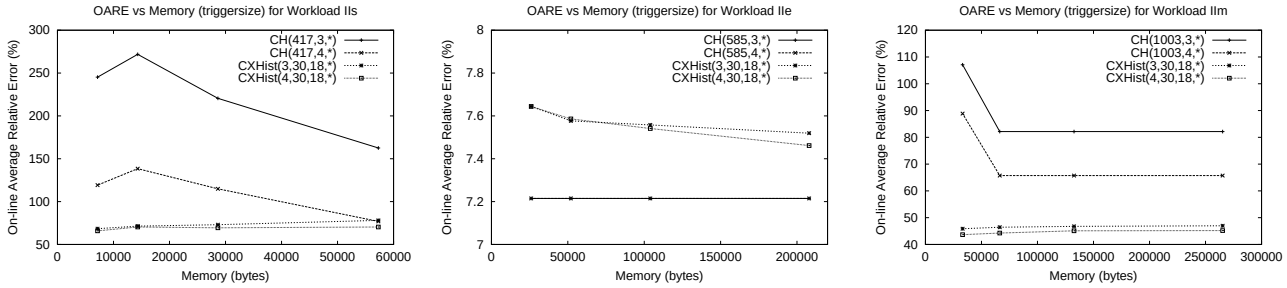

Figure 4: Performance of CH and CXHist over varying memory (triggersize) allocations.

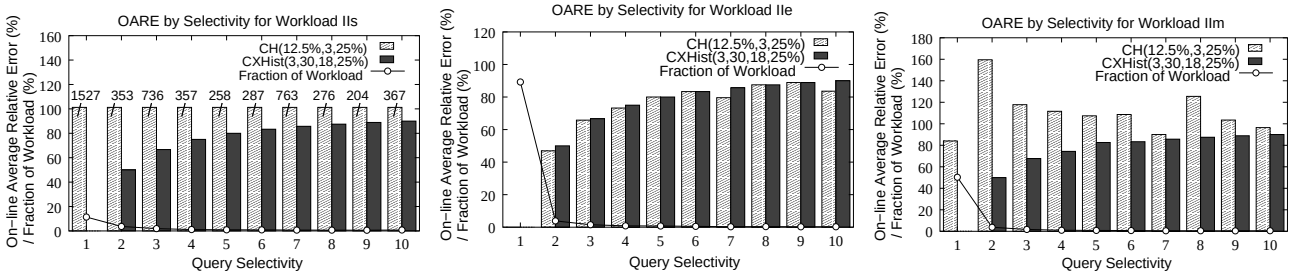

Figure 5: Performance of CH and CXHist partitioned by selectivity for the smallest 10 selectivities.

tivity are infrequent, and (2) the CXHist buckets with large representative values can be modified to store their associated queries exactly (without query modelling) to mimic the behavior of a cache for queries with large selectivity.

3.2 Accuracy vs Workload Skew

In this experiment, the parameter $triggersize$ is fixed at $25\% \times usize$ and the performance on workloads generated with different skew (standard deviation) is measured. Increasing standard deviation denote increasing uniformity, i.e., queries in the workload cover the XML data more uniformly. Highly skewed workloads are generally easy for on-line selectivity estimators, because the set of distinct queries are small and repetitions are frequent. Highly uniform workloads are generally hard, because repetitions seldom occur and almost every query is unique. Figure 6 shows the *on-line* $a.r.e.$ vs skew plots. For substring workloads, CH is significantly less accurate than CXHist especially when the workload is highly skewed. For exact match workloads, both CH and CXHist are very accurate with CH being slightly more accurate for medium skew. For mixed workload, the superior accuracy of CXHist on substring queries dominates.

3.3 Changing Workload Characteristics

In this experiment, we investigate how changes in the workload characteristics affect accuracy of the estimates. The change in workload characteristics is simulated by concatenating two workloads generated with different mean and standard deviation. For brevity, the results for a concatenated workload of IIm and IIIm are presented. Other combinations of workloads exhibit similar behavior.

In Figure 7, we measure the $a.r.e.$ with respect to IIm after each of the first 10,000 queries and with respect to IIIm after each of the last 10,000 queries. CXHist remains accurate throughout the 20,000 queries, whereas the accuracy of CH degrades after the first 5000 queries.

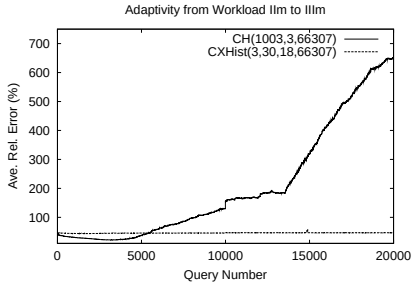

Figure 7: Average relative errors of CH and CXHist on Workload IIm after each update for queries 1 to 10,000 and on Workload IIIm after each update for queries 10,001 to 20,000.

In Figure 8, we show the accuracy of CH and CXHist for different memory allocations on the same concatenated workload IIm-IIIm. Compared to the performance on Workload IIm alone (see Figure 4), the performance of CH on the concatenated workload has degraded significantly (the parameter settings are the same for both figures). The performance of CXHist, on the other hand, remains stable.

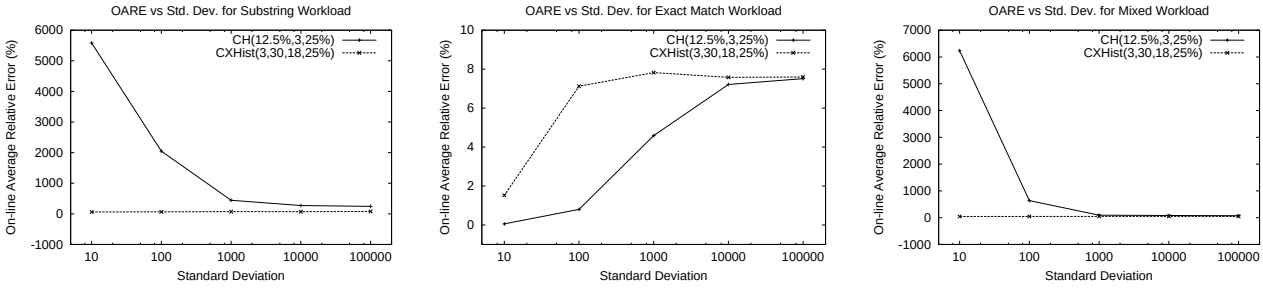

Figure 6: Performance of CH and CXHist over workloads with different skew.

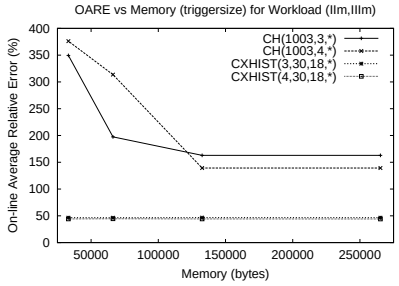

Figure 8: On-line performance of CH and CXHist with different $triggersize$ settings over Workload IIm-IIIm.

4 Related Work

Sub-string selectivity estimation for relational databases has been studied in [14, 26, 12, 10, 11]. All these techniques rely on a pruned suffix tree (PST) as the summary data structure. Note that to adapt these techniques for XML selectivity estimation, one PST will be required for each distinct path. PST-based methods typically parse a query string into possibly (overlapping) component substrings that have an entry in the PST. The counts of the component substrings are combined using either complete independence or Markov assumptions. These assumptions often result in large underestimation errors [4]. CXHist is a new type of histogram that is different from a PST and therefore do not suffer from the same limitations.

Recently, Chaudhuri et al. [4] proposed using a regression tree on top of a basic string estimation technique such as a PST to correct the underestimation problem often associated with independence and Markov assumptions. While modelling of estimation errors can always be used to reduce or correct the errors in a basic string estimation technique, a simpler, more elegant solution is preferable. CXHist is based on a new paradigm using Bayesian classifiers. CXHist does not combine constituent substring selectivities using the independence or Markov assumptions; therefore CXHist does not share the same limitations as PST-based methods.

XML selectivity estimation has also been studied in [1, 5, 20, 21, 16, 9, 28, 27]. These methods can be categorized by whether they are on-line or off-line, whether they handle subtree queries or path queries only, whether they handle leaf values, and whether the leaf values are assumed to be strings or numbers. One major difference between CXHist and previous work is that previous work has focused on queries on the navigational paths or subtrees, whereas CXHist focuses on string-based queries on the leaf values reachable via a given path.

The correlated sub-path tree (CST) method [5, 6] is an augmented PST constructed from the XML data by treating element tags as atomic alphabets and the leaf values as strings. The CST method is off-line, handles subtree queries, and supports substring queries on the leaf values. Being a PST-based technique, the CST method suffers from the underestimation problem. CXHist is a new histogram technique that overcomes this limitation.

The Markov Table (MT) method of [1] uses a fixed order Markov model to capture sub-path statistics in XML data. Queries on leaf values are not supported. The XPathLearner method [16] extends the MT method [1] to an on-line method. In addition, XPathLearner supports exact match queries on leaf values using a form of compressed histogram to store the tag-value pair distribution. However, XPathLearner cannot handle substring queries at all. CXHist is a new type of histogram (not based on the Markov model) that can handle a broad class of query types including substring queries.

The XSKETCH summary structure [20] is a variable order Markov model constructed by performing a greedy search for the best fitting model. XSKETCH as presented in [20] handles subtree queries on element tags only. The extension presented in [21] handles numeric leaf values using standard histograms – string predicates on leaf values are not supported. CXHist assumes that ambiguity in navigational path has been resolved to a set of rooted paths and addresses string-based queries on the leaf values instead.

A different approach is taken by StatiX [9]. StatiX exploits structural information in XML schemas to build histograms that support subtree queries on numerical leaf values. CXHist addresses string-based queries on leaf values.

Position histograms [28] encode the ancestor-descendant relationship of each XML node using points on a 2D plane and builds a histogram for those points. Position histograms specifically address ancestor-descendant

queries, whereas CXHist addresses string-based queries on leaf values.

The bloom histogram [27] method bears some resemblance to CXHist in that the "data" is sorted and partitioned into buckets using selectivity. However, CXHist is fundamentally different in that it is on-line and it models queries instead of data. Because bloom histograms are offline, they are not workload-aware and do not adapt to query workload characteristics.

The concept of using query feedback to build histograms has been used in [3, 2] for relational, numerical data and in [16] for XML path expressions. CXHist uses query feedback in a similar way, but builds a different kind of histogram for string data.

5 Conclusion

In this paper, we have proposed CXHist as a new type of histogram that is on-line, adaptive to changes in workload characteristics and in the underlying data, and applicable to a broad class of query types including string predicates on leaf values. CXHist is the first histogram technique that is based on classification. The key idea is to partition the query selectivities into buckets and use feature distributions to approximately capture the queries that are associated with each bucket. Bayesian classification methods are used with these feature distributions to compute the selectivity of a given query. Our experiments have shown that CXHist provides very accurate estimates for both exact match and substring queries — no previous technique have this property. Moreover, its performance is stable over changes in workload characteristics. While we have only applied CXHist to string predicate queries on XML data in this paper, CXHist is a general histogram technique that is applicable to other types of complex queries in relational-XML hybrid databases.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB 2001*, pages 591–600, 2001.

[2] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD 1999*, pages 181–192, 1999.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a multidimensional workload-aware histogram. In W. G. Aref, editor, *SIGMOD 2001*, pages 211–222. ACM Press, 2001.

[4] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE 2004*, 2004.

[5] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting twig matches in a tree. In *ICDE 2001*, pages 595–604, 2001.

[6] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *PODS 2000*, pages 216–225, 2000.

[7] P. Domingos and M. J. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *International Conference on Machine Learning*, pages 105–112, 1996.

[8] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1972.

[9] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simeon. StatiX: Making XML count. In *SIGMOD 2002*, pages 181–191, 2002.

[10] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multidimensional substring selectivity estimation. In *VLDB 1999*, pages 387–398, 1999.

[11] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. One-dimensional and multi-dimensional substring selectivity estimation. *VLDB Journal 2000*, 9(3):214–230, 2000.

[12] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *PODS 1999*, pages 249–260, 1999.

[13] H. Jiang, H. Lu, W. Wang, and J. X. Yu. XParent: An efficient RDBMS-based XML database system. In *ICDE 2002*, pages 335–336, 2002.

[14] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD 1996*, pages 282–293, 1996.

[15] P. Langley, W. Iba, and K. Thompson. An analysis of bayesian classifiers. In *National Conference on Artificial Intelligence*, pages 223–228, 1992.

[16] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. XPath-Learner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *VLDB 2002*, 2002.

[17] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. on Info. Theory*, 28:129–137, March 1982.

[18] J. Max. Quantizing for minimum distortion. *IRE Trans. on Info. Theory*, 1960.

[19] F. Ozcan, R. Cochrane, H. Pirahesh, J. Kleewein, K. Beyer, V. Josifovski, and C. Zhang. System RX: One part relational, one part XML. In *SIGMOD 2005*, 2005.

[20] N. Polyzotis and M. N. Garofalakis. Statistical synopses for graph-structured XML databases. In *SIGMOD 2002*, pages 358–369, 2002.

[21] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for XML data graphs. In *VLDB 2002*, pages 466–477, 2002.

[22] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB 1997*, pages 486–495, 1997.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing—Explorations in the Microstructure of Cognition*, chapter 8, pages 318–362. MIT Press, 1986.

[24] C. Shannon. Prediction and entropy of printed english. *Bell Systems Technical Journal*, 30:50–64, 1951.

[25] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD 2002*, pages 204–215, 2002.

[26] M. Wang, J. S. Vitter, and B. R. Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *ICDE 1997*, pages 169–180, 1997.

[27] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In *VLDB 2004*, pages 240–251, 2004.

[28] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *EDBT 2002*, pages 590–608, 2002.

[29] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Inter. Tech.*, 1(1):110–141, 2001.

[30] A. Zhou, H. Lu, S. Zheng, Y. Liang, L. Zhang, W. Ji, and Z. Tian. VXMLR: A visual XML-relational database system. In *VLDB 2001*, pages 719–720, 2001.

[31] G. K. Zipf. Human behaviour and the principle of least effort: an introduction to human ecology, 1949.