On-Line Selectivity Estimation for XML Path Expressions using Markov Histograms

Lipyeow Lim [a,*,1] Min Wang [b] Sriram Padmanabhan [b,2]
Jeffrey Scott Vitter [a,3] Ronald Parr [a]

[a] *Duke University, Dept. of Computer Science, Durham, NC 27708, USA.*

[b] *IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA.*

Abstract

The extensible mark-up language (XML) is gaining widespread use as a format for data exchange and storage on the World Wide Web. Queries over XML data require accurate selectivity estimation of path expressions in order to optimize query execution plans. Selectivity estimation of XML path expression is usually done based on summary statistics about the structure of the underlying XML repository. All previous methods require an off-line scan of the XML repository to collect the statistics. In this paper, we propose XPathLearner [4], a method for estimating selectivity of the most commonly used types of path expressions without looking at the XML data. XPathLearner gathers and refines the required statistics using query feedback in an on-line manner and is especially suited to queries in Internet scale applications since the underlying XML repository is either inaccessible or too large to be scanned in its entirety. Besides the on-line property, our method also has two other novel features: (a) XPathLearner is workload aware in collecting the statistics and thus can be more accurate than the more costly off-line method under tight memory constraints, and (b) XPathLearner automatically adjusts the statistics using query feedback when the underlying XML data change. We show empirically the estimation accuracy of our method using the XMark synthetic data set and several real data sets.

Key words: XML, Query Processing, Query Optimization

[4] An earlier version of this work appeared in VLDB 2002.

1 Introduction

The extensible mark-up language (XML) [3] is becoming ubiquitous as a data exchange and storage format. Almost all commercial RDBMSs include some support for XML data; other systems such as Xyleme [22], Niagara [15] and Lore [9] are specially designed to store and query XML data on the web.

Consider an example query expressed in the XQuery language taken from the XQuery specification [5]:

```
FOR $b IN document("*")//book
WHERE $b/publisher = "Morgan Kaufmann"
  AND $b/year = "1998"
RETURN $b/title
```

This query finds the titles of all books published by Morgan Kaufmann in the year 1998. For the example data shown in Figure 1, it returns the book title "Cooking". The XQuery function `document("*")` indicates that all XML documents in the repository should be searched for the path `//book` [8].

Efficient query processing over XML data requires accurate estimation of the selectivities of the path expressions contained in the query. For example, for the query above, we need to know the selectivities of the path expressions `//book/publisher="Morgan Kaufmann"`, `//book/year="1998"` and `//book/title` in optimizing the query execution plan. In RDBMSs that support XML data, these selectivities are used to evaluate the cost of different join plans. In systems that use a tree-like data model (e.g., [9]), these selectivities are used to evaluate the cost of different search and traversal plans [14]. In both scenarios, estimating selectivities of path expressions is essential to XML query optimization and the efficiency of the query processing is highly dependent upon the accuracy of the estimation.

The most commonly used path expressions in XML queries can be classified into three types. Path expressions consisting of tags only (e.g., `//book/title`) are called *simple path expressions*. Path expressions ending in a data value

* Corresponding Author.
Email addresses: liplim@us.ibm.com (Lipyeow Lim), min@us.ibm.com (Min Wang), srp@us.ibm.com (Sriram Padmanabhan), jsv@purdue.edu (Jeffrey Scott Vitter), parr@cs.duke.edu (Ronald Parr).
[1] Present address: IBM Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA. , Tel: 1-914-784-6156, Fax: 1-914-784-7455.
[2] Present address: IBM Silicon Valley Laboratory, 555 Bailey Ave., San Jose, CA 95141, USA. .
[3] Present address: Purdue University, 150 N. University Street, West Lafayette, IN 47907 USA. .

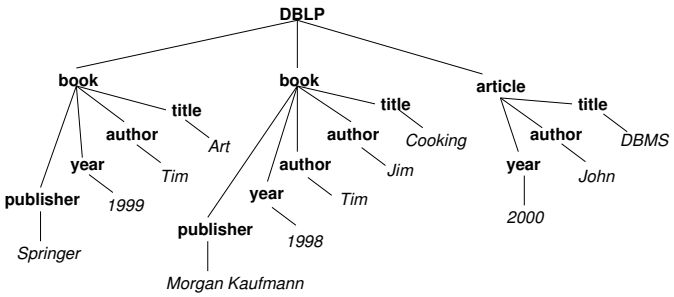


Fig. 1. An example XML data tree. Tag names are in bold and data values are in italics.

(e.g., `//book/year="1998"`) are called *single-value path expressions*. The XML specification also allows multiple tag-value bindings in a path (e.g., `//chapter="2"/section="3"`). We call such path expressions *multi-value path expressions*.

Selectivity estimation of XML path expressions is usually done at query optimization time using statistics about the structure of the XML data. The main challenges in collecting and storing these statistics are as follows:

- How to obtain the structure of the XML data? All previous work scans the entire XML repository in an off-line manner [1,14]. However, off-line scans are often not possible or feasible in Internet-scale applications since Internet-scale repositories are either inaccessible or too large to be scanned entirely.
- How to capture the statistics for the selectivities of different types of XML path expressions using a small amount of memory? State-of-the-art techniques proposed in [1,6] are unsatisfactory either because they are limited to the selectivity of simple path expressions only [1] or they are not space efficient [6].
- How to use the limited storage space in the most effective way? Ideally, more storage resource should be spent on storing the statistics that are relevant to the most frequently queried portions of the XML repository. All previous work are oblivious to workload distribution and consequently waste precious storage space in storing statistics of infrequently queried portions of the repository.
- How to incrementally update the statistics when the underlying XML data change? The XML repositories in Internet scale applications are constantly changing. To ensure accurate XML path selectivity estimation, the statistics must keep up with the change. However, the off-line periodic scan used by previous work to obtain new statistics is neither effective not efficient because of the scanning cost associated with the size of the repositories.

In this paper, we present XPathLearner, a novel on-line learning method for estimating the selectivity of XML path expressions. Our XPathLearner assumes a Markov model [14,1] of path selectivities and learns this model from

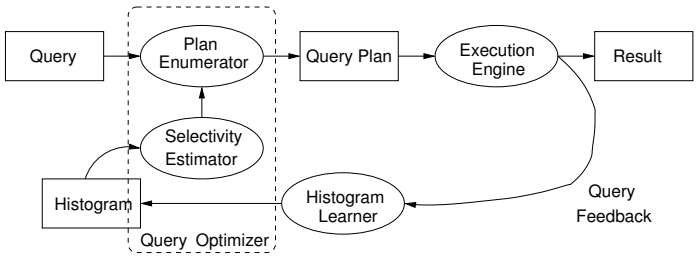


Fig. 2. Workflow of XPathLearner. The top path is for query processing and feedback loop is for workload-driven selectivity estimation processing.

query feedback using error reduction strategies. Two such strategies are presented: the heavy-tail rule and the delta rule. Our XPathLearner overcomes the limitations of previous work and has the following properties:

- Instead of scanning the XML data, XPathLearner collects the required statistics in an on-line manner from *query feedback* information (see Figure 2).
- XPathLearner learns both tag distribution and value distribution from query feedback. It is designed to estimate the selectivity of all three types of common path expressions. It can also estimate the selectivity of a path expression containing a simple wildcard.
- XPathLearner is workload-aware in collecting the required statistics. The allocated storage space is used in the most effective way since more statistics are collected for more frequently queried portions of the XML data.
- XPathLearner automatically adapts to changing XML data, because the statistics are refined on-line according to the most current query feedback. XPathLearner incurs a small overhead in updating the statistics using query feedback, but this cost is offset by an increase in estimation accuracy.

Since query feedback provides only partial information about the path selectivity distribution, we would expect that an on-line method using query feedback to be less accurate than an off-line method. In our experiments we show that not only does XPathLearner come close in accuracy to the off-line method in general, but sometimes surpasses the off-line method because of its workload-driven nature.

The rest of the paper is organized as follows. In the next section, we review related work. Section 3 introduces the terms used in our description of the on-line XML path selectivity estimation problem. Section 4 presents the XPathLearner: the Markov chain model for path expressions, the Markov histogram for storing the model parameters, two approaches for dealing with single-value path counts, and two learning strategies. We present our experimental evaluation in Section 5 and draw conclusions in Section 6.

2 Related Work

To estimate the selectivities of XML path expressions, the Lore system stores statistics of all distinct paths up to length m, where m is a tunable parameter [14]. Selectivity of paths longer than m are estimated assuming the Markov property (see Equation (5) of Section 4.1). The paths stored include both tags and data values and no further summarization is performed. The space requirement of the statistics used in the Lore system is therefore prohibitively large, because the number of possible data values in a big XML repository can be extremely large and the number of distinct paths with data values can therefore be even larger. Our XPathLearner presents two different approaches to address this problem: storing the counts of paths with data values in a compressed histogram [20], or, alternatively, evicting stored entries based on certain criteria (similar to a cache).

Aboulnaga et al. extended the idea used by the Lore system in their *Markov table* method [1]. The Markov table method consists of a set of pruning and aggregation techniques on the statistics used in the Lore system. One limitation of the Markov table method is that paths with data values are not considered (i.e., it can only estimate the selectivity of simple path expressions). This limitation is serious, because the selectivities of paths with data values are crucial in optimizing XML queries that have large top-down search space and highly selective data values. For such queries, a bottom-up search plan is more cost-effective than a top-down search [14]. For the XML data in Figure 1, the query "find the titles of all books authored by Jim" is an example. The path expression `//author="Jim"` is more selective than `//book`. Our method aims to overcome this limitation by storing statistics for paths with data values while keeping the space requirement low.

Aboulnaga et al. also proposed a tree-based method known as the *path tree* method [1] for estimating the selectivity of XML paths without data values. A path tree is a summarized form of the XML data tree where sibling nodes of the same tag are aggregated. Tree pruning and aggregation techniques are proposed to reduce the space requirement of path trees. Their experiments, however, have shown that the path tree method is inferior to the Markov table method for real data sets.

Chen et al. proposed another off-line tree-based method for estimating XML subtree selectivity [6]. A suffix tree based data structure is used to store the statistics of the XML data obtained from scanning the repository. Pruning and aggregation techniques are proposed to compress this data structure. However, the space requirement of their summarized data structure is especially large for XML data with long data values. Subtree selectivity estimation involves estimating the selectivity of a query that matches a subtree in the XML data

tree as opposed to matching a single path. The problem of subtree selectivity estimation is significantly more general than the path selectivity problem that our method addresses, but even if the technique in [6] is modified for path expressions containing tags only, Aboulnaga et al. have shown that their Markov table method is superior in accuracy [1].

Polyzotis et al. have recently proposed XSketch a more general model based framework [16,17], however, the construction of an XSketch model requires a greedy heuristic search that evaluates a candidate model against the XML data (or a summarized form of it). XPathLearner is much more lightweight, and builds and adapts itself using only query feedback information.

The XML path selectivity estimation methods proposed in [16,17,1,6,14] all require information from scanning the repository. These off-line methods share several limitations. The requirement of an off-line scan limits the use of these methods on large (especially Internet-scale) repositories. They are not tuned to the workload distribution: the workload may only query a small portion of the XML data and hence a small portion of the statistics stored in the allocated space. The repository needs to be rescanned whenever the data in the repository change sufficiently. Our XPathLearner overcomes these limitations by learning the statistics from query feedback in an on-line manner. Keeping statistics gathered from query feedback ensures that the allocated space is used to store statistics that are up-to-date and relevant to the query workload.

Selectivity estimation using statistics gathered from query has been proposed in [2,4] for relational data. Tree-structured data such as XML present new challenges. Whereas the self-tuning histograms of [2,4] capture continuous distribution over numeric attributes, a corresponding self-tuning XML path selectivity estimator needs to capture a discrete distribution over a set of non-numeric path labels. In the continuous case, self-tuning histograms (such as [2,4]) can start with a uniform distribution over a large interval and *refine* this distribution by creating finer partitions of this interval. In contrast, a self-tuning XML path selectivity estimator does not have an interval to start with and needs to *learn* each and every path label in the data. Even if a Markov model is imposed on the tree data to simplify the distribution entailed by the tree data, it has been shown that learning a Markov model can be hard [11].

In the off-line XML path selectivity estimation domain, Aboulnaga et al. [1] and McHugh et al. [14] use estimation techniques based on the Markov model. An order $m-1$ Markov model assumes that the selectivities of all the paths whose lengths are less than or equal to m capture all the required statistics. The experiments in [1] show that, in practice, first and second order Markov models are sufficient to capture the path selectivity statistics with little loss in information. Our method assumes the Markov model, but differs from previous work in that our method (1) gathers statistics in an on-line manner without

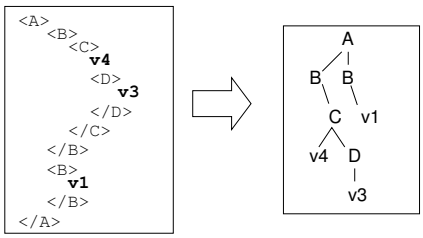


Fig. 3. An XML document and its corresponding tree representation.

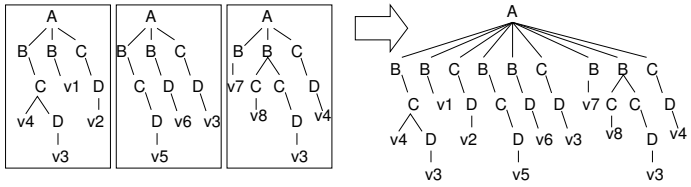


Fig. 4. The XML data tree (right) is constructed from a repository of three XML documents (left). Alphabets in upper case denote tag names, 'v' followed by a number denotes a data value. The selectivity of the simple path expression //B/C/D is 3, the selectivity of the single-value path expression //B/C/D=v3 is 2, the selectivity of the multi-value path expression //B/C=v4/D=v3 is 1, and the selectivity of //A/*/D is the sum of the selectivities of //A/B/D and //A/C/D, which yields 4.

scanning the repository, (2) handles the three types of common query path expressions, and (3) is workload-aware.

3 Preliminaries

In this section, we introduce basic terms and notation used in describing the XML path selectivity problem. In particular we introduce the different types of path expressions and define their corresponding selectivities.

An XML document is structurally a tree (we ignore IDREFs) where each node is associated with a tag or a value. In practice, values are almost always associated with leaf nodes. An XML *data tree* is a huge tree constructed either by merging the roots of all the XML documents if the tag associated with the root of each document is the same or by introducing a *super root* as the parent of the root node of each XML document. An XML data tree represents a repository of XML documents (see Figure 4).

A *simple path expression* p of length n is a sequence of tags $\langle t_1, t_2, \ldots, t_n \rangle$, $t_i \in \Sigma$, where Σ is the set of all possible tag names. The tag sequence in a path expression encodes a navigational path through the XML data tree where each pair in the sequence (t_i, t_{i+1}) correspond to a (directed) edge with the tags (t_i, t_{i+1}) in the XML data tree. Using XPath notation, such a navigational sequence can also be written as $//t_1/t_2/\ldots/t_n$ (e.g., `//book/title`). We will

use as shorthand, where there is no confusion, the string $t_1t_2 \ldots t_n$ to represent $//t_1/t_2/ \ldots /t_n$.

A *multi-value path expression* is a simple path expression where values are associated with one or more tags in the path expression [5]. A special case of a multi-value path expression is a *single-value path expression* where only the last tag in the path is associated with a value. The length of a single-value path expression $//t_1/t_2/ \ldots /t_n{=}v_n$ is $n+1$.

Consider the XML data tree shown in Figure 4. The path //B/C/D is a simple path expression; the path //B/C=v4/D=v3 is a multi-value path expression, and the path //B/C/D=v3 is a single-value path expression.

We denote the selectivity of a (simple, multi-value or single-value) path expression p as $\sigma(p)$. The selectivity of a simple path expression p is defined to be the number of paths in the XML data tree that match the tag sequence in p. The selectivity of a single-value path expression is similar to that of simple path expressions except that the navigational path ends in a value instead of a tag in the XML data tree. The selectivity of a multi-value path expression p is defined to be the number of subtrees that matches the tag and value sequence in p.

The path expressions that we consider in this paper are allowed to contain one wildcard. In this paper, we restrict each wildcard ('*') to match a single tag. Moreover we do not consider path expressions beginning or ending with a wildcard [6]. The selectivity of a path expression p with a single wildcard is the sum of the selectivities of all the (non-wildcard) path expressions that are possible matches to p. Example selectivities are given in Figure 4.

A *query feedback* is a tuple $(p, \sigma(p))$ consisting of a path expression and its corresponding true selectivity. Our definition of query feedback assumes minimal information about the query execution engine. It is possible to obtain more feedback information from the query execution engine. The amount of information we can obtain depends upon the underlying data storage model and the query plan used by the execution engine. For example, using the Lore model and a top-down plan, the query execution engine can provide as feedback the selectivities of all prefixes of the given path. Since XML storage and retrieval technology is still evolving, we assume minimal feedback information in this paper.

[5] A multi-value path expression is a special case of a twig [6].

[6] We are investigating the extensions for more complicated wildcards in ongoing work.

4 XPathLearner

In this paper, we propose XPathLearner, an on-line method for estimating the selectivity of a given path expression (simple, single-value, multi-value), using statistics that are obtained from query feedback only and reside in a given amount of space (memory).

XPathLearner models the selectivity of path expressions as an order $(m-1)$ Markov chain or network. The Markov model has been used by [14,1] to model the selectivities of all possible paths in the XML data tree. XPathLearner differs from [14,1] in that only the selectivities of paths that occur in the query workload are modeled. Moreover, in contrast to previous work, XPathLearner learns the parameters associated with the Markov model in an on-line manner using query feedback information only. XPathLearner stores these parameters using a *Markov histogram* and updates the Markov histogram using query feedback.

In this section we describe in detail the Markov model for path expressions, the associated Markov histogram, two approaches to dealing with single-value path counts, the on-line update algorithm, and two update strategies.

4.1 Path Expression Model

A simple path expression query of length n can be modeled as a sequence of random variables $\langle X_1 X_2 \dots X_n \rangle$ (in XPath notation $//X_1/X_2/\dots/X_n$), where each X_i is instantiated with some tag name from Σ the set of all possible tags. The selectivity of an instance of a simple path expression $t_1 t_2 \dots t_n$ can be computed from

$$\sigma(t_1 t_2 \dots t_n) = P(X_1 X_2 \dots X_n) \times N, \tag{1}$$

where N is the total number of nodes in the XML data tree and $P(X_1 X_2 \dots X_n)$ is a shorthand for $P(\langle X_1 X_2 \dots X_n \rangle = \langle t_1 t_2 \dots t_n \rangle)$, the probability that each $X_i = t_i$ for $i = 1, 2, \dots, n$. An order $(m-1)$ Markov model for the sequence of random variables $X_1 X_2 \dots X_n$ assumes that each random variable X_i is only dependent on its $m-1$ predecessors in the sequence,

$$P(X_1 X_2 \dots X_n) \approx \frac{\prod_{j=1}^{n-m} P(X_j X_{j+1} \dots X_{j+m-1})}{\prod_{i=2}^{n-m} P(X_i X_{i+1} \dots X_{i+m-2})}. \tag{2}$$

Moreover, the Markov model makes the stationary assumption,

$$\begin{aligned} P(\langle X_j X_{j+1} \dots X_{j+m-1} \rangle = \langle t_j t_{j+1} \dots t_{j+m-1} \rangle) \\ = P(\langle X_i X_{i+1} \dots X_{i+m-1} \rangle = \langle t_j t_{j+1} \dots t_{j+m-1} \rangle), \end{aligned} \tag{3}$$

for any integers $i, j \geq 1$. Using the $m-1$ order Markov model, it is sufficient to store $P(X_1X_2 \ldots X_l)$, for $l = 1, 2, \ldots m$, in order to approximate the selectivity of any simple path expression. In practice, each $P(\langle X_1X_2 \ldots X_l \rangle = \langle t_1t_2 \ldots t_l \rangle)$ is stored in a table of frequency counts $f(t_1t_2 \ldots t_l)$, where

$$P(X_1X_2 \ldots X_l) = f(t_1t_2 \ldots t_l)/N. \tag{4}$$

Each $f(t_1t_2 \ldots t_l)$ counts the number of occurrences of the path $t_1t_2 \ldots t_l$ in the XML data tree. Combining Equations (1), (2), (3) and (4), the selectivity of a simple path expression can be approximated by

$$\hat{\sigma}(t_1t_2 \ldots t_n) = \frac{\prod_{j=1}^{n-m} f(t_jt_{j+1} \ldots t_{j+m-1})}{\prod_{i=2}^{n-m} f(t_it_{i+1} \ldots t_{i+m-2})}. \tag{5}$$

In particular, when $m = 2$, the approximation formula becomes,

$$\hat{\sigma}(t_1t_2 \ldots t_n) = \frac{f(t_1t_2)f(t_2t_3) \ldots f(t_{n-1}t_n)}{f(t_2)f(t_3) \ldots f(t_{n-1})}. \tag{6}$$

Previous work [1,6,10] has shown that the Markov model works well for many real XML data sets for $m = 2$ and $m = 3$.

The selectivity of single-value path expressions of the form $p = //t_1/t_2/ \ldots /t_n{=}v_n$ can be approximated analogously by associating an additional random variable $X_{n'}$ with the value v_n and instantiating the $X_{n'}$ with the value v_n,

$$\hat{\sigma}(//t_1/t_2/ \ldots /t_n{=}v_n) = \hat{\sigma}(t_1t_2 \ldots t_n)\frac{f(t_{n-m+2}t_{n-m+3} \ldots t_nv_n)}{f(t_{n-m+2}t_{n-m+3} \ldots t_n)}. \tag{7}$$

Similarly, for multi-value path expressions,

$$\hat{\sigma}(//t_1{=}v_1/t_2{=}v_2/ \ldots /t_n{=}v_n) = \hat{\sigma}(t_1t_2 \ldots t_n) \prod_{j=1}^{n} \frac{f(t_{j-m+2}t_{j-m+3} \ldots t_jv_j)}{f(t_{j-m+2}t_{j-m+3} \ldots t_j)}. \tag{8}$$

4.2 Markov Histograms

An order $(m-1)$ Markov model requires storing the counts of all length-l simple or single-value paths, i.e., all $f(t_1t_2 \ldots t_l)$ and $f(t_1t_2 \ldots t_{l-1}v_{l-1})$, for $l = 1, 2, \ldots m$, in order to approximate the selectivity of any path expression. In the offline case, storing just the counts of the length-m paths are sufficient, because the count of a length-l path, where $l < m$, can be computed from the counts of all length-$(l+1)$ paths,

$$f(t_1t_2 \ldots t_l) = \sum_{\tau \in \Sigma} f(\tau t_1t_2 \ldots t_l) \tag{9}$$

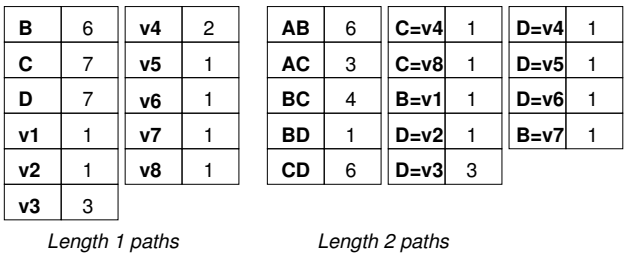


Fig. 5. The first order Markov histogram corresponding the XML data tree in Figure 4.

by assuming that the XML data is a tree and that the counts of all the length-$(l+1)$ paths are known. In the on-line case, not all the length-$(l+1)$ path counts have been learnt at a given time and for those that have been learnt, the corresponding counts may not have converged to the true counts. Hence, the counts of paths with length $l < m$, have to be stored for on-line selectivity estimation using a Markov model.

An order $(m-1)$ *Markov histogram* is a table storing a set of distinct paths, with length $l < m$, along with their associated occurrence counts or selectivities. For simplicity of presentation we consider $m = 2$ for the rest of this paper. An example of a first order Markov histogram is shown in Figure 5. Markov histograms approximate the selectivity of simple, single-value, and multi-value path expressions using the formulas in Equations (5), (7), and (8) respectively. When the count of a length-(l) path $(l < m)$ is needed and is not stored in the Markov histogram, a default count of 1 is used.

Storing the counts of single-value path (of length $l < m$) efficiently presents a unique challenge, because the number of distinct data values is typically very large compared to the number of distinct tags in an XML data tree. For example, the DBLP data set contains 91,878 unique values and only 29 unique tag names. For $m = 2$, this translates to 841 possible tag-tag pairs and 2,664,462 possible tag-value pairs. Given a small amount of memory, it is not possible to store the counts of all length $(l < m)$ paths exactly. Two approaches are possible. First, store the all simple path counts exactly and use a form of compressed histogram [20] to store the counts of single value paths. The second approach is not to differentiate between the Markov histogram entries for simple and single-value path expressions, but to delete entries based on some criteria when the Markov histogram runs short of memory (much like in a cache).

4.3 Compressed Histogram Approach

The large number of XML data values prohibits storing the counts of single-value path expressions with length $(l < m)$ exactly. A similar problem is

addressed in [12]; however, we adopt a simpler approach. Motivated by the fact that most of the probability mass (of counts) is concentrated in a very small number of single-value paths in many real XML data sets, we use a compressed histogram approach similar to [20].

(1) Store the k single-value paths with the largest counts exactly. The parameter k can be tuned. Each entry in the 'top k' data structure stores the tuple $\langle$*single-value path*, *count*$\rangle$.
(2) Single-value paths with a count smaller than the minimum count among the top k single-value paths are aggregated into buckets. A bucket is a tuple $\langle$*simplepath*, *feature*, *sum*, *num*$\rangle$, where *simple path* is the tag-only prefix of the single-value path, the field *num* is the number of single-value paths it represents, the field *sum* is the sum of the counts of those single-value paths assigned to that bucket, and the field *feature* is the feature of the data value corresponding to that bucket. A single-value path is assigned to a bucket based on some feature of the data value in the single-value path.

For example, if a first-order Markov histogram is used and the first letter of the data value in a single-value path is used as the bucket assignment feature, the compressed histogram for the single-value path counts will consist of at most k tuples of the form $\langle$*single-value path*, *count*$\rangle$ and at most $36 \cdot |\Sigma|$ tuples (where Σ is the set of tags) of the form $\langle$*simple path*, *feature*, *sum*, *num*$\rangle$, assuming that values are not case sensitive and are alphanumeric strings. Ideally, the feature should be chosen so that the counts in each bucket are as uniform as possible, that is, the variance of the counts represented in a particular bucket should be minimized. Choosing features dynamically and maintaining the data structure for dynamic features are part of our future work.

Retrieval. Accessing the count of a given single-value path requires searching through the top k entries first. If the required single-value path is not found, the feature of the given single-value path is used to locate the corresponding bucket and the count is computed as sum/num.

Update. Given a single-value path and an updated count, the top k entries are searched first and if a matching single-value path is found, its count is updated. Otherwise, we check if the updated count of the given single-value path is larger than the minimum count in the top k entries. If it is larger, the minimum entry in the top k is displaced into the bucket corresponding to the displaced entry. If it is smaller, the count of the given single-value path is added to the *sum* field of the corresponding bucket, and the *num* field of the same bucket is incremented. Each bucket therefore encodes the average

selectivity of all the current instances of the single-value paths belonging to that bucket.

Compress. When memory is scarce, the tag-feature histogram can be further compressed by aggregating buckets with similar selectivities.

4.4 Cache-based Approach

A different and even simpler approach to keeping the Markov histogram small is to remove or evict entries based on some criteria. Each entry in a Markov histogram stores a simple or single-value path p and its associated count $f(p)$.

The first criterion is to evict entries with counts smaller than a threshold parameter. Recall that our Markov histogram assigns a default value of 1 to entries that are not stored, so removing entries whose counts are close to 1 represents little loss of information.

The second criterion is to evict entries that are seldom used (similar to the least frequently used policy in caching). To keep track of frequency of use, a counter needs to be associated with each entry in the Markov histogram. To minimize memory overhead, a single byte counter can be used, but all counters in the Markov histogram will need to be reset every 256 units of use. Each Markov histogram entry will be of the form $\langle$*single-value path*, *count*, *counter*$\rangle$

In our experiments, whenever memory is short, the small count criterion is first applied, followed by the least frequently used criterion, until sufficient memory has been freed up.

4.5 An Example

We illustrate how selectivity estimation can be done using our Markov histogram with the compressed histogram approach. Consider our earlier example in Figure 5. The corresponding representation using our Markov histogram with $k = 1$ is shown in Figure 6.

The selectivity of the simple path expression //B/C/D can be estimated by

$$\hat{\sigma}(BCD) = \frac{f(BC)}{f(C)} \cdot f(CD) = \frac{4}{7} \cdot 6 = 3.43,$$

tag	count
A	1
B	6
C	7
D	7

(a) tag counts

tag-tag	count
AB	6
AC	3
BC	4
BD	1
CD	6

(b) tag-tag counts

tag=value	count
D=v3	3

(c) top k tag-value counts

tag	feat.	sum	#pairs
B	a	1	1
B	b	1	1
D	a	2	2
D	b	2	2
C	a	1	1
C	b	1	1

(d) tag-feature histogram

Fig. 6. A first order Markov histogram using $k = 1$ and the first letter of the data value as the bucketing feature. Further suppose that the data values {v1, v2, v3, v4} all begin with the letter 'a' and {v5, v6, v7, v8} with the letter 'b'.

which has an absolute error of 0.43. The selectivity of the single-value path expression //B/C/D=v3 can be estimated by

$$\hat{\sigma}(BCD = v3) = \frac{f(BC)}{f(C)} \cdot \frac{f(CD)}{f(D)} \cdot f(D = v3) = 1.47,$$

which has an absolute error of 0.53, since the real selectivity is 2. The selectivity of the multi-value path expression //B/C=v4/D=v3 can be estimated by

$$\begin{aligned}&\hat{\sigma}(//B/C = v4/D = v3)\\ &= \frac{f(BC)}{f(C)} \cdot \frac{f(CD)}{f(D)} \cdot f(D = v3) \cdot \frac{f(C = v4)}{\sum_v f(C = v)} = 0.735,\end{aligned}$$

which has an absolute error of 0.265. Note that the value v4 has feature "a". The selectivity of the simple path expression //A/*/D (with wildcard) can be estimated by

$$\hat{\sigma}(A * D) = \sum_{\alpha} \hat{\sigma}(A\alpha D) = 3.57,$$

which has an absolute error of 0.43, since $\sigma(ABD) + \sigma(ACD) = 1 + 3 = 4$.

4.6 On-line Update Algorithms

We describe the two update algorithms that our XPathLearner uses to learn a Markov histogram from query feedback: the heavy-tail rule and the delta rule.

Our two update algorithms follow the high-level steps outlined in Algorithm 1 and differ in the update equations used in line 10. The Markov Histogram is assumed to be initially empty. The function `compress-add entry` adds any unknown length-2 path to the Markov histogram: it learns the set of labels of a discrete distribution. A physical entry is not necessarily added to the histogram whenever `compress-add entry` is called. When memory is scarce, `compress-add entry` can trigger pruning or aggregation techniques (such as those in [1]) to compress the histogram. In contrast to learning the labels, the update equation in the algorithm learns the frequency counts of the labels in the discrete distribution.

Algorithm 1 UPDATE (Mhistogram f, Feedback $(p, \sigma(p))$, Estimate $\hat{\sigma}(p)$)

```
 1: if |p| ≤ 2 then
 2:    if not exists f(p) then
 3:       compress-add entry f(p) = σ(p)
 4:    else
 5:       f(p) ← σ(p)
 6: else
 7:    for all (t_i, t_{i+1}) ∈ p do
 8:       if not exists f(t_i t_{i+1}) then
 9:          compress-add entry f(t_i t_{i+1}) = 1
10:       f(t_i t_{i+1}) ← update {depends on update strategy}
11: for all t_i ∈ p, i ≠ 1 do
12:    if not exists f(t_i) then
13:       compress-add entry f(t_i)
14:    f(t_i) ← max{f(t_i), Σ_α f(α t_i)}
```

4.6.1 The Heavy-tail Rule

Given estimated selectivity $\hat{\sigma}(p)$ and query feedback $(p, \sigma(p))$, where $p = t_1 t_2 \ldots t_n$ and $\sigma(p)$ is the real selectivity, we first compute the observed error

$$\epsilon = \sigma(p) - \hat{\sigma}(p). \tag{10}$$

Recall that the selectivity of path p is computed as

$$\hat{\sigma}(t_1 t_2 \ldots t_n) = \left(\prod_{i=1}^{n-2} \frac{f(t_i t_{i+1})}{f(t_{i+1})} \right) \cdot f(t_{n-1} t_n). \tag{11}$$

We need to refine all the $f(t_i t_{i+1})$ terms in this product based on the observed error ϵ. The updates to the $f(t_{i+1})$ terms are dependent on the $f(t_i t_{i+1})$ terms through Equation (9) and will be described later. We may also want to attribute more of the estimation error to the terms associated with the end of the path p. There are two reasons for this: First, the terms closer to the end of the path p are naturally more relevant to the selectivity of path p. Second, attributing more of the estimation error to them also minimizes the effect on other paths sharing the same prefix as p. We therefore assign weights to the $f(t_i t_{i+1})$ terms that increase with i.

Let w_i be the (unnormalized) weight associated with t_i in path p. We update the $f(t_i t_{i+1})$ terms as follows:

$$f_{k+1}(t_i t_{i+1}) \leftarrow f_k(t_i t_{i+1}) + sign(\epsilon)\,(\gamma|\epsilon|)^{w_i / \sum_{j<n} w_j}, \tag{12}$$

where $\sum_{j<n} w_j$ is the normalization factor, γ is the learning rate or discount factor, t_i is the ith tag in the query path p, and $f_k(\cdot)$ and $f_{k+1}(\cdot)$ are the counts before and after the update, respectively. The weights we used are

$$w_i = 2^i, \qquad i = 1, 2, \ldots \tag{13}$$

If the last element t_n in the query path is a data value, the weight for $f(t_{n-1}t_n)$ is defined to be the same as that for $f(t_{n-2}t_{n-1})$. The intuition for this is that an instance of a tag cannot take more than one data value in an XML data tree. Note that for query path $p = t_1 \ldots t_n$, the updates to the relevant Markov histogram entries have the following property:

$$\prod_{i=1}^{n-1} (\gamma|\epsilon|)^{w_i / \sum_j w_j} = \gamma|\epsilon|. \tag{14}$$

In general, the discount factor γ is set to be less than one and therefore it makes the error correction smaller. This prevents XPathLearner from over reacting to an estimation error and smoothens the error reduction process. The $f(t_i)$ terms are updated as

$$f_{k+1}(t_i) \leftarrow \max\{\sum_j f_{k+1}(t_j t_i),\ f_k(t_i)\}, \tag{15}$$

since the sum of the counts for all length-2 paths ending in t_i must be a lower bound on the true $f(t_i)$ (by Equation (9)). The term $f_k(t_i)$ could be greater than $\sum_j f_{k+1}(t_j t_i)$, if XPathLearner has encountered a query feedback with a length-1 query path t_i previously.

As an example, suppose that the Markov histogram maintained by XPathLearner is in the state as shown in Figure 6. Further suppose that the feedback for path ACD is $(ACD, 6)$ and its estimated selectivity is $\hat{\sigma}(ACD) =$

$3 \cdot 6 \div 7 \approx 3$. The observed error is $\epsilon = 6 - 3 = 3$ and using $\gamma = 1$, the following updates are made:

$$\begin{aligned} f_{k+1}(AC) &\leftarrow round(3 + 3^{1/3}) = 4, \\ f_{k+1}(CD) &\leftarrow round(6 + 3^{2/3}) = 8, \\ f_{k+1}(C) &\leftarrow \max\{4 + 4,\ 7\} = 8, \\ f_{k+1}(D) &\leftarrow \max\{1 + 8,\ 7\} = 9, \end{aligned}$$

The estimated selectivity of the path ACD after the update is $4 \cdot 8 \div 8 = 4$. The estimation error has been reduced.

4.6.2 The Delta Rule

A more principled way of updating the Markov histogram using query feedback is to attribute the estimation error to the relevant edge counts using the delta rule. The delta rule is an error reduction learning technique first proposed by Rumelhart et al. [18].

The learning scenario is the same as that in the heavy-tail method. The histogram learner is given estimated selectivity $\hat{\sigma}(p)$ and query feedback $(p, \sigma(p))$, where $p = t_1 \dots t_n$ and $\sigma(p)$ is the real selectivity. We compute the observed error as before,

$$\epsilon(p) = \sigma(p) - \hat{\sigma}(()p). \tag{16}$$

The delta rule minimizes an error function. We choose our error function to be the squared error,

$$E = [\epsilon(p)]^2 = [\sigma(p) - \hat{\sigma}(()p)]^2 . \tag{17}$$

We also adopt the following shorthand to make the equations more readable:

$$w_{\alpha\beta}^{(k)} = f_k(\alpha, \beta), \tag{18}$$

$$W_{\beta}^{(k)} = f_k(\beta) = \sum_{\alpha \in \Sigma} w_{\alpha\beta}^{(k)} = w_{\alpha\beta} + \sum_{\substack{x \in \Sigma \\ x \neq \alpha}} w_{\alpha\beta}^{(k)} \tag{19}$$

The superscript $^{(k)}$ will be dropped if there is no confusion over the time of the variable.

The delta rule states that for an error function $E(w_{\alpha\beta})$ the update to term $w_{\alpha\beta}$ should be proportional to the negative gradient of $E(w_{\alpha\beta})$ with respect to $w_{\alpha\beta}$ evaluated at time k,

$$w_{\alpha\beta}^{(k+1)} \leftarrow w_{\alpha\beta}^{(k)} - \gamma \frac{\partial E(w_{\alpha\beta}^{(k)})}{\partial w_{\alpha\beta}}, \tag{20}$$

where γ is the proportionality constant or learning rate. Simplifying the derivative using Equation (17),

$$\frac{\partial E(w_{\alpha\beta})}{\partial w_{\alpha\beta}} = 2\epsilon(p)\frac{\partial \epsilon(p)}{\partial w_{\alpha\beta}} = -2\epsilon(p)\frac{\partial \hat{\sigma}(()p)}{\partial w_{\alpha\beta}}$$

Using the Equation (5) for computing $\hat{\sigma}(()p)$, the term $w_{\alpha\beta}$ can occur multiple times in the numerator and in the denominator. For example, in $\hat{\sigma}(()ABCBAB)$, the term w_{AB} would appear twice in the numerator, and twice in the denominator, because W_B appears twice in the denominator. Consider the general case that $w_{\alpha\beta}$ appears u times in the numerator and v times in the denominator of the expression for $\hat{\sigma}(()p)$,

$$\frac{\partial \hat{\sigma}(()p)}{\partial w_{\alpha\beta}} = \frac{\partial}{\partial w_{\alpha\beta}}\left(\frac{w_{\alpha\beta}^u}{W_\beta^v} \times r\right), \tag{21}$$

where $r = \hat{\sigma}(()p) \times W_\beta^v / w_{\alpha\beta}^u$ contains the rest of the terms that do not contain $w_{\alpha\beta}$. Differentiating using the quotient rule and simplifying,

$$\frac{\partial \hat{\sigma}(()p)}{\partial w_{\alpha\beta}} = \hat{\sigma}(()p)\left(\frac{uW_\beta - vw_{\alpha\beta}}{w_{\alpha\beta}W_\beta}\right). \tag{22}$$

Hence, our update equation for the Markov histogram term $w_{\alpha\beta} = f(\alpha, \beta)$ that occurs u in the numerator and v times in the denominator of the formula for $\hat{\sigma}(()p)$ is

$$w_{\alpha\beta}^{(k+1)} \leftarrow w_{\alpha\beta}^{(k)} + 2\gamma\epsilon(p)\hat{\sigma}(()p)\left(\frac{uW_\beta^{(k)} - vw_{\alpha\beta}^{(k)}}{w_{\alpha\beta}^{(k)}W_\beta^{(k)}}\right). \tag{23}$$

The learning rate parameter γ is usually chosen by experimentation. A learning rate that is too small may result in slow convergence to the minimum error and a learning rate that is too big may result in oscillations between non-optimal error values.

As an example, suppose that the Markov histogram maintained by XPathLearner is in the state as shown in Figure 6. Assume again that the feedback for path ACD is $(ACD, 6)$ and its estimated selectivity is $\hat{\sigma}(()ACD) = 3{\cdot}6 \div 7 \approx 3$. The observed error is $\epsilon(ACD) = 6 - 3 = 3$ and using $\gamma = 0.5$, the following updates are made:

$$\begin{aligned}
f_{k+1}(AC) &\leftarrow \mathit{round}(3 + 2 \cdot 0.5 \cdot 3 \cdot 3 \cdot \frac{7-3}{3 \cdot 7}) = 5,\\
f_{k+1}(CD) &\leftarrow \mathit{round}(6 + 2 \cdot 0.5 \cdot 3 \cdot \frac{3}{6}) = 8,\\
f_{k+1}(C) &\leftarrow \max\{5+4,\ 7\} = 9,\\
f_{k+1}(D) &\leftarrow \max\{1+8,\ 7\} = 9,
\end{aligned}$$

The estimated selectivity for path ACD after the updates is $5 \cdot 8 \div 9 \approx 4$. The estimation error has been reduced.

4.7 Update Overhead

Let the time needed to access an entry in the Markov histogram be $O(l)$, where l is the size of the data structure used to implement the Markov histogram. Let the query path in question be $p = t_1 t_2 \ldots t_n$. The update equations for the heavy-tail rule method (Equation (12)) and the delta rule method (Equation (23) both take $O(l)$ time. There are $O(n)$ iterations of the two loops starting on line 7 and line 11 of the update algorithm (Algorithm 1). Each iteration of the loop starting on line 11 requires $O(l)$ time, since the summation in line 14 is over at most all the length-m paths in the order $(m-1)$ Markov histogram that has size l. Therefore each update takes

$$O(nl) \tag{24}$$

time, where n is the query path length and l is the size of the Markov histogram. Since n is bounded by the height of the XML data tree and l by the small amount memory allocated to store the Markov histogram, l and n are practically constants. Therefore the update overhead is a constant.

4.8 Batch Update Strategies

While the emphasis of this paper is on-line methods, we briefly outline two batch processing strategies that can be used with XPathLearner. Batch updates assume that a buffer is available to hold b query feedback tuples until the batch update procedure is activated. Batch updates are useful for two reasons.

(1) When it is not feasible or possible to update the Markov histograms after each query, a batch of query feedback can be collected and used to update the Markov histogram periodically.
(2) When a batch of query feedback is available during initialization, the Markov histogram can be initialize using a batch update strategy instead of being initialized to empty histograms. This may reduce the estimation errors associated with a 'cold start'.

4.8.1 Subpath Elimination

The subpath elimination strategy for batch updates is based on three assumptions:

(1) the set of query paths in the batch contains some paths that are subpaths of other paths in the batch,
(2) the underlying data remained the same for the entire batch of query paths, and
(3) the Markov assumption holds reasonably well for the path selectivities

The implication of these assumptions is that the feedback for the subpath can give additional statistical information about the longer path. For example, if the query paths ABCD and ABC both occur in the batch, then using their true selectivities from query feedback, we know that $f(CD)/f(C)$ should be equal to $\sigma(ABCD)/\sigma(ABC)$. We can now use the on-line delta rule method to update $f(CD)$ and $f(C)$ using $\sigma(ABCD)/\sigma(ABC)$ as the true value for $f(CD)/f(C)$. The subpath elimination strategy assumes the delta rule method for updating the Markov histogram.

We sketch the algorithm for the subpath elimination batch update method in Algorithm 2. The batch of query feedback is first sorted (line 1) according to

Algorithm 2 SUBPATHELIMINATION(f, *Qfb*)

INPUT: *Markov histogram f, Batch of Query Feedback Qfb*

```
 1: Sort Qfb in increasing path length and lexicographic order.
 2: while stopping criterion ≠ true do
 3:   for all query path p ∈ Qfb do
 4:     Search for longest query path z ∈ Qfb that is a subpath of p
 5:     if z does not exist then
 6:       UPDATE(f, (p, σ(p)), σ̂(p)) using delta rule.
 7:     else
 8:       σ ← σ(p)/σ(z)
 9:       σ̂ ← σ̂(p)/σ̂(z)
10:       for all w_αβ in σ̂(p), but not in σ̂(z) do
11:         Update w_αβ using σ, σ̂, and delta rule Equation (23).
12:       Update all affected W_β using Equation (15).
```

increasing query path length and according to lexicographic order for paths with the same length. Each iteration of the `while`-loop (line 2) then applies the subpath elimination batch update method until some stopping criterion is true. Exactly one scan through the batch of query feedback is made in each iteration. The sorted batch of query feedback facilitates the search for the longest query path that is a subpath of the current query path p that is being processed (line 4). If no subpath is found, the on-line delta rule update procedure is used to perform the update (line 6). If a subpath is found, the true selectivity of the subpath is eliminated from the true selectivity of the current query path p. The delta rule is then applied to the histogram entries that are used in the computation of $\hat{\sigma}(p)$ and not used in the subpath (line 11).

The `while`-loop applies the subpath elimination batch update method until some stopping criterion is true. The `while`-loop can terminate when

- the error has reached a given error threshold, or
- the decrease in error between consecutive iterations has fallen below a given threshold, or
- the number of iterations has exceeded a given constant.

4.8.2 Batch Delta Rule

Let B be the set of b query paths whose query feedback has been buffered. We can use delta rule to update the Markov histogram by defining a sum of squared error function for the set of query paths B,

$$E(B) = \sum_{p \in B} [\epsilon(p)]^2 = \sum_{p \in B} [\sigma(p) - \hat{\sigma}(p)]^2 .$$

To update a particular histogram entry $w_{\alpha\beta}$, we compute the derivative,

$$\frac{\partial E(w_{\alpha\beta})}{\partial w_{\alpha\beta}} = \sum_{p \in B} \frac{\partial [\epsilon(p)]^2}{\partial w_{\alpha\beta}} = \sum_{p \in B} 2\epsilon(p) \frac{\partial \epsilon(p)}{\partial w_{\alpha\beta}}$$

Let B' be the subset of B containing paths that require the histogram entry $w_{\alpha\beta}$ in the computation of their selectivity estimates. The selectivity estimates of all the paths in $B - B'$ do not depend on $w_{\alpha\beta}$ and their derivative is zero,

$$\frac{\partial E(w_{\alpha\beta})}{\partial w_{\alpha\beta}} = \sum_{p \in B'} 2\epsilon(p) \frac{\partial \epsilon(p)}{\partial w_{\alpha\beta}} = - \sum_{p \in B'} 2\epsilon(p) \frac{\partial \hat{\sigma}(p)}{\partial w_{\alpha\beta}}. \tag{25}$$

Using Equation (21) and the delta rule, we can update a Markov histogram entry $w_{\alpha\beta} = f(\alpha, \beta)$ using the query feedback for a set of query paths B' using,

$$w_{\alpha\beta}^{(k+1)} \leftarrow w_{\alpha\beta}^{(k)} + \frac{2\gamma}{w_{\alpha\beta}^{(k)} W_{\beta}^{(k)}} \sum_{p \in B'} \left\{ \epsilon(p) \hat{\sigma}(p) \left[u(p) W_{\beta}^{(k)} - v(p) w_{\alpha\beta}^{(k)} \right] \right\}, \tag{26}$$

where γ is the learning rate parameter, and $u(p)$ and $v(p)$ are the number of times that $w_{\alpha\beta}$ occurs in the numerator and denominator of the formula for $\hat{\sigma}(p)$ respectively.

We sketch the algorithm for the delta rule batch update method in Algorithm 3. The first `for`-loop (line 1) initializes all non-existent entries in the histogram that are required for the batch of queries. The `while`-loop (line 4) implements the delta rule batch update method as derived in Equation (26). Our implementation is optimized using the assumption that the size of the batch of query feedback is much larger than the total number of histogram entries updated. Hence each iteration of the `while`-loop scans through the

Algorithm 3 BATCHDELTARULEUPDATE(f, *Qfb*)

INPUT: *Markov histogram f, Batch of Query Feedback Qfb*

1: **for all** $w_{\alpha\beta} = f(\alpha\beta)$ used in *Qfb* **do**
2: Initialize $w_{\alpha\beta} = 1$, if $w_{\alpha\beta}$ is not in histogram.
3: Update W_β using Equation (15).
4: **while** stopping criterion $\neq$ true **do**
5: Initialize array U to zeros.
6: **for all** query path $p \in$ *Qfb* **do**
7: **for all** $w_{\alpha\beta}$ involved in $\hat{\sigma}(p)$ **do**
8: $U[w_{\alpha\beta}] \leftarrow U[w_{\alpha\beta}] + \epsilon(p)\hat{\sigma}(p)\left\{u(p)W_\beta^{(k)} - v(p)w_{\alpha\beta}^{(k)}\right\}$
9: **for all** $w_{\alpha\beta} = f(\alpha\beta)$ used in *Qfb* **do**
10: $w_{\alpha\beta} \leftarrow w_{\alpha\beta}^{(k)} + \frac{2\gamma}{w_{\alpha\beta}^{(k)} W_\beta^{(k)}} U[w_{\alpha\beta}]$
11: Update all affected W_β using Equation (26).

batch of query feedback only once. If this assumption is not true, the order of the two `for`-loops (line 6 and 7) needs to be reversed.

The `while`-loop applies the delta rule until some stopping criterion is true. Reasonable stopping criteria includes stopping when the error has reached a given error threshold, or when the decrease in error between consecutive iterations has fallen below a given threshold, or when the number of iterations has exceeded a given constant.

5 Experiments

We implemented our XPathLearner in C/C++ using the XML Parser Toolkit [7]. We investigate the following issues in our experiments:

(1) the accuracy of XPathLearner under varying memory constraints when it is trained on one query workload and evaluated using a different workload,
(2) the convergence properties associated with XPathLearner,
(3) the adaptivity of XPathLearner when the workload changes from one distribution to another,
(4) the on-line accuracy of XPathLearner, i.e., the estimation accuracy on one workload, as XPathLearner updates itself after each query,
(5) the distribution of on-line errors according to the true selectivity of the queries, and
(6) the occurrence frequency of on-line errors.

We describe briefly the data sets used, the query workloads, the performance measures, and the methods used in the comparisons, before describing each experiment in greater detail.

Performance Measures. We have used the average relative error and the average absolute error to measure the accuracy of XPathLearner. The average relative error (*a.r.e.*) and the average absolute error (*a.a.e.*) with respect to a set of queries Q of size n are defined as

$$a.r.e. = \frac{1}{n} \sum_{q \in Q} \frac{|\sigma(q) - \hat{\sigma}(q)|}{\sigma(q)}, \quad a.a.e. = \frac{1}{n} \sum_{q \in Q} |\sigma(q) - \hat{\sigma}(q)|, \tag{27}$$

where $\sigma(()q)$ is the selectivity of query path q in the workload Q 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 Q. In an on-line setting, we would also like to measure the on-line or dynamic performance of an estimation method. The *on-line a.a.e.* and the *on-line a.r.e.* are defined as in (27), except that the selectivity estimation method is allowed to update itself in between queries.

Data Set. We performed our experiments on the XMark synthetic data set [21,19] and on several real data sets: DBLP [13], Swiss protein [7], and Shakespeare [8]. For brevity, we present only the results from the DBLP data set and the XMark data set in this paper. The characteristics of each data set are summarized in Table 1.

Characteristic	DBLP	XMark
Size (MBytes)	10	116
No. of nodes	261,256	1,479,327
No. of distinct tags	29	74
No. of distinct values	91,878	415,262
Path tree depth	5	13
No. of path tree tag nodes	57	514
No. of path tree value nodes	109,741	675,844

Table 1
Characteristics of the DBLP data set and the XMark data set.

Query Workload. In the experiments we present in this section we used workloads consisting of simple and single-value query path expressions with positive selectivity. We did generate negative workloads (consisting of query paths that do not appear in the data, i.e., query paths with zero selectivity) by

[7] http://www.expasy.ch/sprot
[8] http://metalab.unc.edu/bosak/xml/eg/shaks200.zip

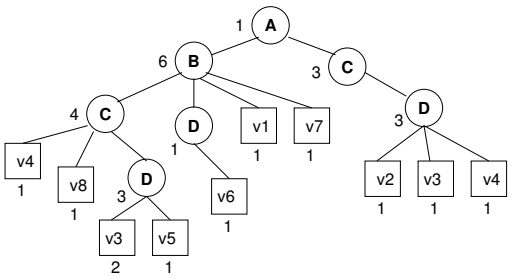


Fig. 7. The path tree corresponding to the XML data tree in Figure 4. Circle nodes denote tag names and square nodes denote values.

generating random sequences of legal tags ending with a random legal value; however, for all the negative workloads that we generate, our XPathLearner consistently returns a selectivity of 1 for each negative path[9]. (The default return value for paths that are not captured in our Markov histogram is 1.) Hence, the average absolute error is 1. This result contrasts sharply with the summarized Markov tables of [1], where the average absolute error for negative workloads can be as high as 250.

Positive query workloads are generated from the path tree [1] of the given XML data set. Recall that a path tree summarizes an XML data tree by aggregating every sibling having the same tag into a single node annotated by a count of the number of occurrences in the original XML data tree. Figure 7 shows an example of a path tree corresponding the XML data tree in Figure 4. We generate positive path queries as follows. All the root-to-leaf paths in the path tree are first enumerated. A query path is generated by randomly choosing a root-to-leaf path and then randomly choosing a starting level and a path length that are within limits of the length of the chosen root-to-leaf path. The random query path of the chosen length is then output starting from the chosen level in the chosen root-to-leaf path.

These root-to-leaf paths are not chosen uniformly, but from a distribution weighted according to their selectivities,

$$P[\text{choosing root-to-leaf path } p] = \frac{\sigma(p)}{\sum_{r \in R} \sigma(()r)}, \tag{28}$$

where R is the set of all root-to-leaf path of the given path tree. The reason for choosing the root-to-leaf path in this way is to prevent the query workload from having too many query paths with very small selectivities.

Comparisons. We compare the performance of the off-line method and several versions of XPathLearner. The labeling convention for the different versions are as follows: **xpl-o1-dt-lb** denotes a first order ("o1") XPathLearner

[9] A positive workload of 1000 query paths was used as the training workload for that experiment.

using the delta rule ("dt") and the compressed histogram approach for storing paths with leaf values ("lb" for leaf buckets); **xpl-o1-ht-lb** denotes a first order XPathLearner using the heavy-tail rule ("ht") and the compressed histogram approach; **xpl-o1-dt-ca** denotes a first order XPathLearner using the delta rule and the cache-based approach to storing paths with leaf values ("ca"); **xpl-o2-dt-ca** denotes a second order ("o2") XPathLearner using the delta rule and the cache-based approach. The **off-line** method differs from the XPathLearner in that the Markov histogram is constructed by scanning the repository. It differs from the Markov table method [1] in that (1) no summarization is done of the tag-tag counts, (2) the tag-value counts are stored, and (3) the tag-value counts are summarized using the method described in Section 4.3.

Initial Condition. We assume that we do not know anything about the workload distribution at the start of each experiment; that is, we start with an empty Markov histogram. More sophisticated ways of obtaining an initial Markov histogram are possible, but an empty initial histogram represents a reasonable worst case.

Counting Memory. An order $(m-1)$ XPathLearner using the compressed histogram approach consists of m tables and a compressed histogram data structure. Each table i stores entries of the form $\langle path_i, count \rangle$, where $path_i$ is a simple path of length i. A compressed histogram stores k entries of the form $\langle path_m, count \rangle$ where $path_m$ is a single-value path of length m, and some number of aggregated entries of the form $\langle path_{m-1}, feature, sum, num \rangle$, where $path_{m-1}$ is a simple path of length $m-1$. Each length i path requires i integers. All other fields take one integer each. Each integer takes four bytes. (For a first order example see Figure 6.)

An order $(m-1)$ XPathLearner using the cache-based approach consists of just m tables, each table i storing entries of the form $\langle path_i, count, counter \rangle$, where $path_i$ is a simple or single-value path of length i. Each length i path again requires i four-byte integers, each count requires one four-byte integer, and each counter requires one byte.

The memory requirements of each method is accounted for by counting the number of entries in each table and compressed histogram, and multiplying by the corresponding memory requirement of each type of entry.

Parameters. We set the learning rate γ to 1 for the heavy-tail rule update strategy and 0.1 for the delta rule update strategy. For the cache-based approach, we set the threshold for evicting low count entries to 30, i.e., entries

with count less than 30 are candidates for eviction. These values were found to be reasonably good by experimentation.

5.1 Accuracy vs Space

In this experiment, we measure the estimation error under varying memory constraints. Two different query workloads from the DBLP data set are used: one as the training set and the other as the testing set. Each workload consists of 4096 query paths, of which about 3100 paths are distinct. The average true selectivities of the training and testing workloads are 2034 and 2296 [10], respectively.

The goal of this experiment is to see how our on-line Markov histogram performs on a workload that is different from its training workload. We define a workload difference measure with respect to a first-order Markov histogram in order to quantify the difference between two workloads.

Workload Diff. Given two workloads A and B, we construct for each workload the set of length-2 paths of all the query paths in the workload. Let the set of length-2 paths of A and B be S_A and S_B, respectively. The workload difference measure of A and B is

$$\text{workload\_diff(A,B)} = 1 - \frac{|S_A \cap S_B|}{|S_A \cup S_B|}. \tag{29}$$

Intuitively, the workload_diff measures how different the first-order Markov models of the two given workloads are.

We experimented on a large number of training-testing workload pairs and we present a typical result set in Figure 8. The workload_diff of the training and testing workload we present is 88.4%. For the XPathLearner using the compressed histogram approach, we measure the estimation error as k varies from 32 to 4096. The k values (for the top k values) are then converted to memory usage in bytes and the estimation errors are plotted against memory usage. Our experiments show that in terms of absolute errors our on-line XPathLearner (**xpl-o1-ht-lb**, **xpl-o1-dt-lb**, and **xpl-o2-dt-ca**) is more accurate than the off-line version. Amongst the two on-line update strategies, the delta rule is usually more accurate than the heavy-tail strategy. In terms of relative errors, the second order **xpl-o2-dt-ca** is the most accurate under tight memory constraints. The performance of **xpl-o1-ht-lb** and **xpl-o1-dt-lb** are within 10% of the **off-line** method.

[10] Since the total number of nodes in the XML data tree is $N = 261,256$, these selectivities correspond to 0.77 % and 0.87 %.

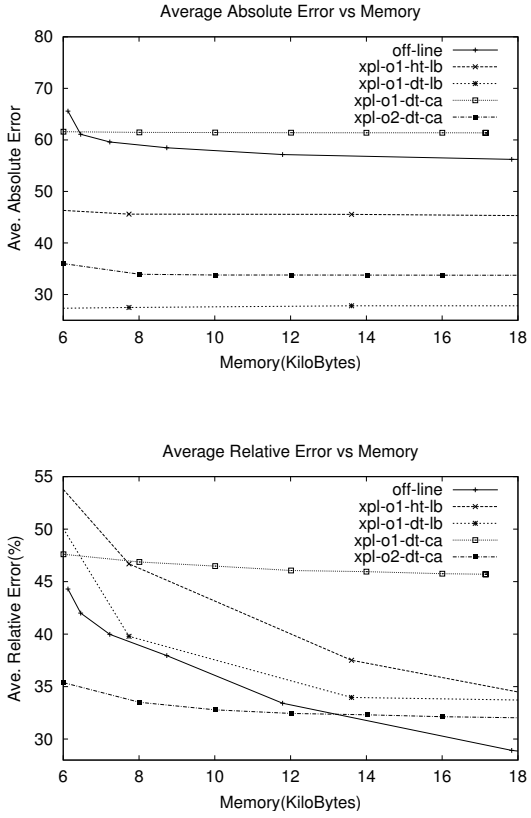


Fig. 8. **Accuracy vs Memory for DBLP data set.** Accuracy of the on-line Markov histogram method on a testing workload that is 88.4% different from the training workload. Both workloads contain 4096 single-value query paths, about 3100 of which are distinct.

The relationship between k and the memory usage in bytes for the off-line and on-line XPathLearner (using compressed histogram approach) is graphed in Figure 12. We note that, for fixed k, the memory requirement of the off-line method is more than that of the on-line method; for $k = 512$, the off-line method requires 2947 bytes and the on-line method only 1934 bytes. The off-line method has to store statistics for the entire XML repository while the on-line method only needs to store the statistics of the workload.

We also show our results for two workloads (training and testing) consisting of simple path expressions only (no value nodes involved). Both workloads consist of 1000 simple path expressions, and although the two workloads are different, their workload_diff is zero [11] . This property arises because the set of length-2 paths entailed by both workloads are the same. The estimation error rates are tabulated in Table 2.

[11] Since the number of possible tags is small, a workload of 1000 paths captures most of the length-2 paths.

Method	a.a.e.	a.r.e.(%)	Memory
xpl-o1-dt-lb	0.086	0.197	764 Bytes
off-line	0.110	0.331	796 Bytes
xpl-o1-ht-lb	1.198	0.243	764 Bytes
xpl-o1-dt-ca	87.9	19.535	760 Bytes

Table 2
Accuracy of various methods for simple path expression queries on the DBLP data set. Estimation error of the **off-line** method, the on-line **xpl-o1-ht-lb**, **xpl-o1-dt-lb**, and **xpl-o1-dt-ca** methods for a workload consisting only of simple path expressions (tag-only path expressions). The on-line **xpl-o1-dt-lb** outperforms the others.

5.2 Convergence

We want to investigate how well the on-line method converges to a given workload distribution. One query workload of 1000 query paths (840 distinct) from the DBLP data set is used in this experiment. We measure the average absolute and relative errors over the entire workload as the histogram learner processes each query path in the same workload. Since the Markov histogram is initially empty, the first few error measurements will be large, and as the Markov histogram converges to the workload distribution, the measured error will be small. The error measurements over each iteration or update of a Markov histogram are plotted in Figure 9 and Figure 10. All the methods are given the same amount of memory (7.7 KBytes or $k = 512$). The results in Figure 9 and Figure 10 show that the accuracy of our XPathLearner reaches very acceptable levels within the first 100 iterations. Figure 11 shows how the memory constraint (governed by k when the compressed histogram approach is used) affects the convergence properties of XPathLearner. Our results show that XPathLearner can still be very accurate even when little memory is allocated. The spike in the **xpl-o1-dt-lb** plot of Figure 9 is due to the occurrence of a path that violates the Markov assumption.

5.3 Adapting to Data Distribution Change

This experiment investigates how the on-line Markov histogram will adapt to a workload that has its first 1000 query paths generated from the original DBLP path tree and the next 1000 query paths generated from a modified DBLP path tree with random perturbation to the counts at each node. The perturbation is intended to simulate the DBLP data changing over time. We introduce the perturbation by generating a random number $U \in [1-\delta, 1+\delta]$ for each node r in the path tree. The count associated with node r is then

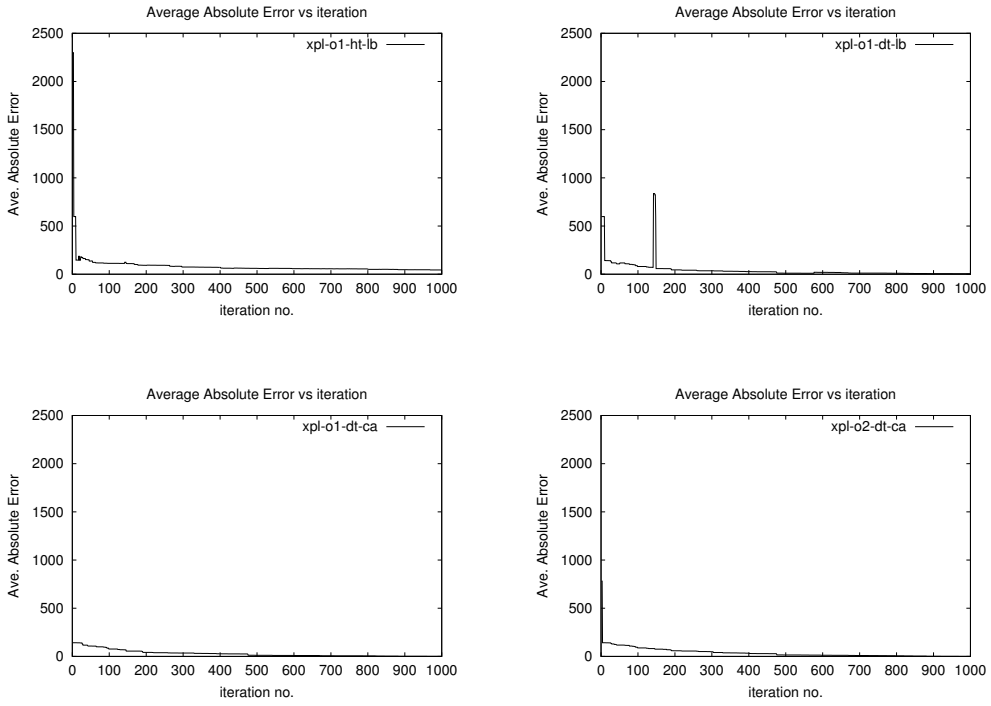


Fig. 9. **Convergence of** *a.a.e.* **(fixed memory) on the DBLP data set.** The absolute error averaged over an entire workload at each iteration of the learning process.

scaled by the random number U,

$$count(r) \leftarrow \max\{1,\ U \cdot count(r)\} \tag{30}$$

We realize that these perturbations are simplistic and a finer model of the changes in XML data is part of our future work.

The modified path tree that we generated using $\delta = 0.7$ has a Kullback-Liebler divergence of 0.129299 bits. The Kullback-Liebler (KL) divergence of a modified path tree f_1 with respect to the original path tree f_0 is defined as

$$KL(f_0|f_1) = \sum_{y \in \{\text{root-to-node paths}\}} f_0(y) \log \frac{f_0(y)}{f_1(y)}. \tag{31}$$

The KL divergence is a common difference measure of distributions [11].

This experiment is performed as follows. Let the workload of 1000 query paths generated from the original path tree be Q_{old} and the workload of 1000 query paths generated from the modified path tree be Q_{new}. Let Q_{mix} be the concatenation of Q_{old} and Q_{new}. We let our XPathLearner ($k = 32$) learn the Markov histogram from this mixed workload. For the first 1000 iterations we measure the average absolute error over Q_{old} after each update and for the

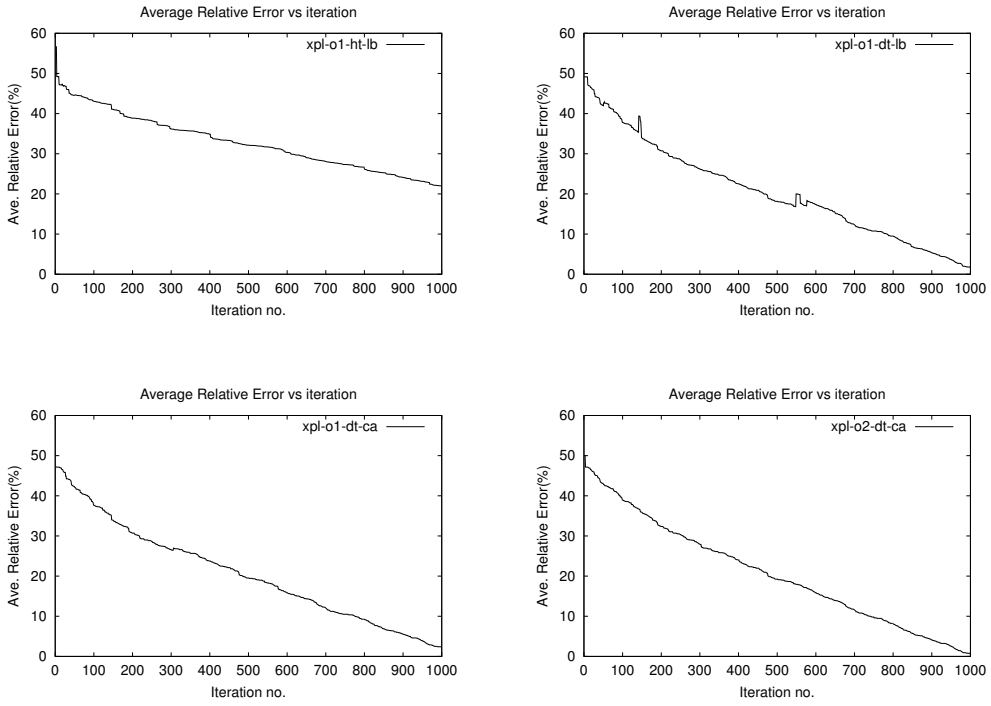


Fig. 10. **Convergence of** *a.r.e.* **(fixed memory) for the DBLP data set.** The relative error averaged over an entire workload at each iteration of the learning process.

next 1000 iterations we measure the average absolute error over Q_{new} after each update. The average absolute error at the end of each iteration is plotted in Figure 13. The spikes near iteration 320 in the plot for **xpl-o1-dt-ca** is probably an artifact of the cache-based approach. The spike at iteration 1001 is expected and shows the transition from workload Q_{old} to workload Q_{new}. Since the distribution underlying Q_{old} is different from that underlying Q_{new}, the average absolute estimation error with respect to Q_{new} at iteration 1001 is very large. About 100 iterations after the transition, the on-line method has adapted to Q_{new}.

5.4 On-line Accuracy vs Space

How does XPathLearner perform in an on-line setting? We measure the on-line estimation errors of XPathLearner under different memory constraints and plot the results in Figure 14. A query workload of 10,000 queries generated from the XMark data set is used. Recall that the on-line average error measures the error of each query in the workload while allowing the estimation method to update itself after each query. The on-line average error therefore measures the performance of XPathLearner while it is "in action".

In terms of *on-line a.r.e.*, the XPathLearner using the delta rule and compressed histogram is the most accurate among the four methods. In terms of *on-line a.a.e.*, the cache-based approach (with delta rule) seems to be more accurate.

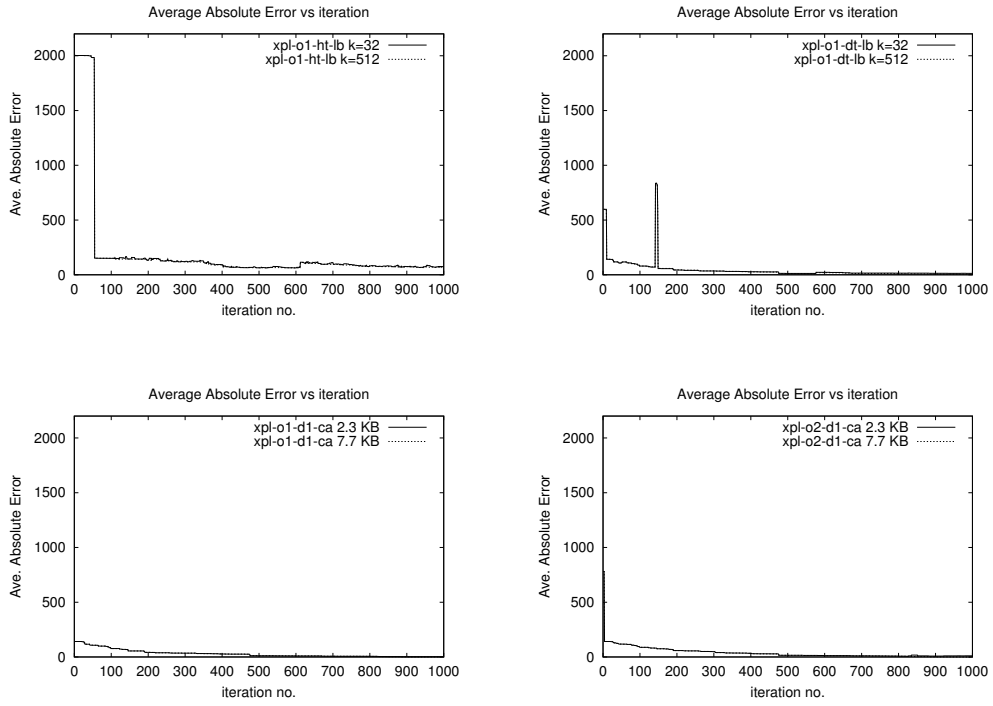


Fig. 11. **Convergence of** *a.a.e.* **(variable memory) for the DBLP data set.** Each plot shows the absolute error convergence curves for two different memory constraint and for most of the time, the two curves are indistinguishable. These plots show that convergence is not sensitive to the memory constraint.

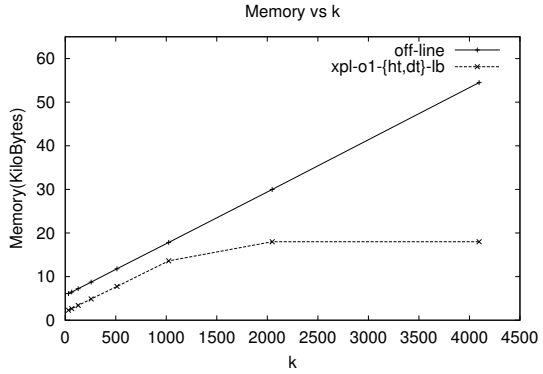


Fig. 12. Memory vs k for the DBLP data set.

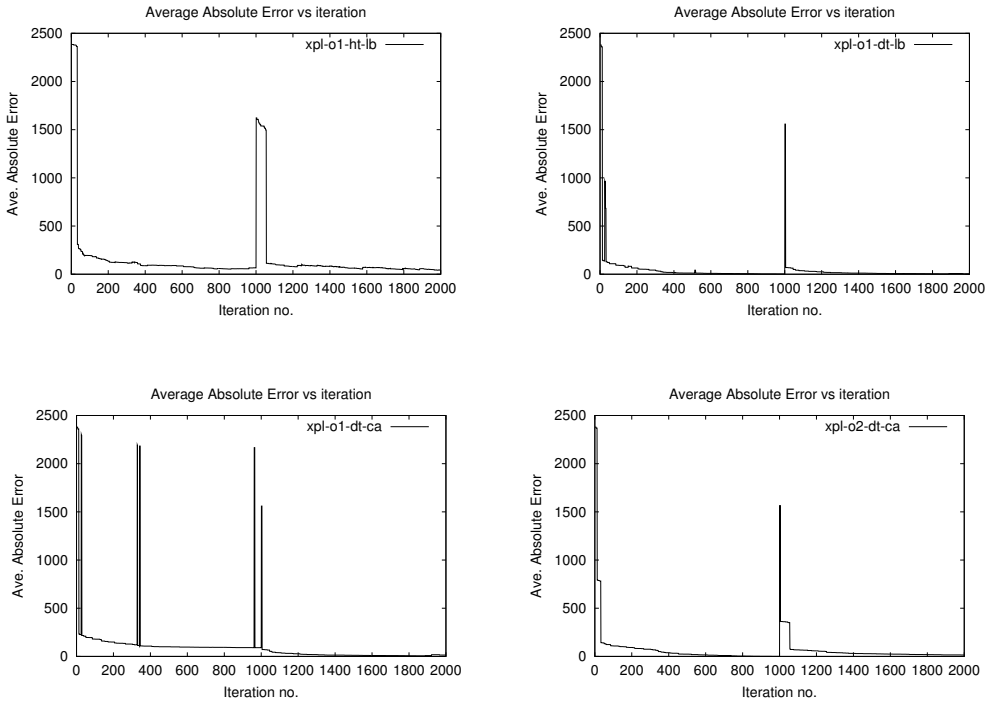


Fig. 13. **Adaptability (DBLP data set).** Average absolute error averaged over Q_{old} for iteration 1-1000 and averaged over Q_{new} for iteration 1001-2000.

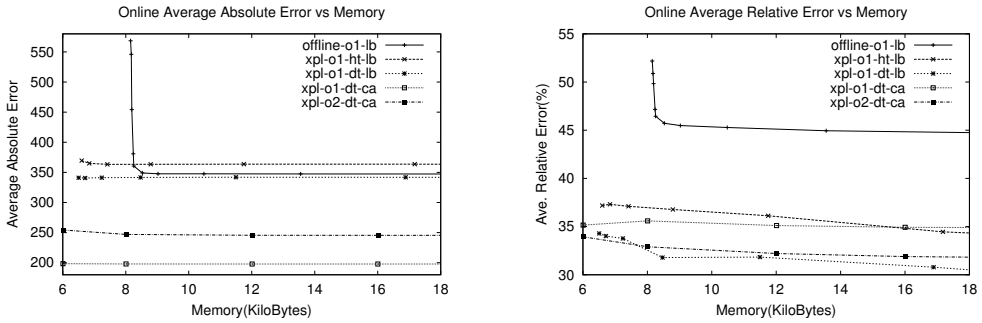


Fig. 14. **On-line error vs memory for the XMark data set.** The on-line estimation performance of XPathLearner under varying memory constraints on a workload of 10,000 queries drawn from the XMark data set.

5.5 On-line Errors vs Selectivity

We further analyze the on-line estimation errors (absolute errors) by partitioning the errors into 40 bins according to the true selectivity of the queries. The errors in each bin are averaged and the average error for each bin is plotted in Figure 15. The on-line errors we used are for the same workload of 10,000 queries from the XMark data set used in Section 5.4. Each XPathLearner method is given about 7.3 KBytes of memory. The plots show that the

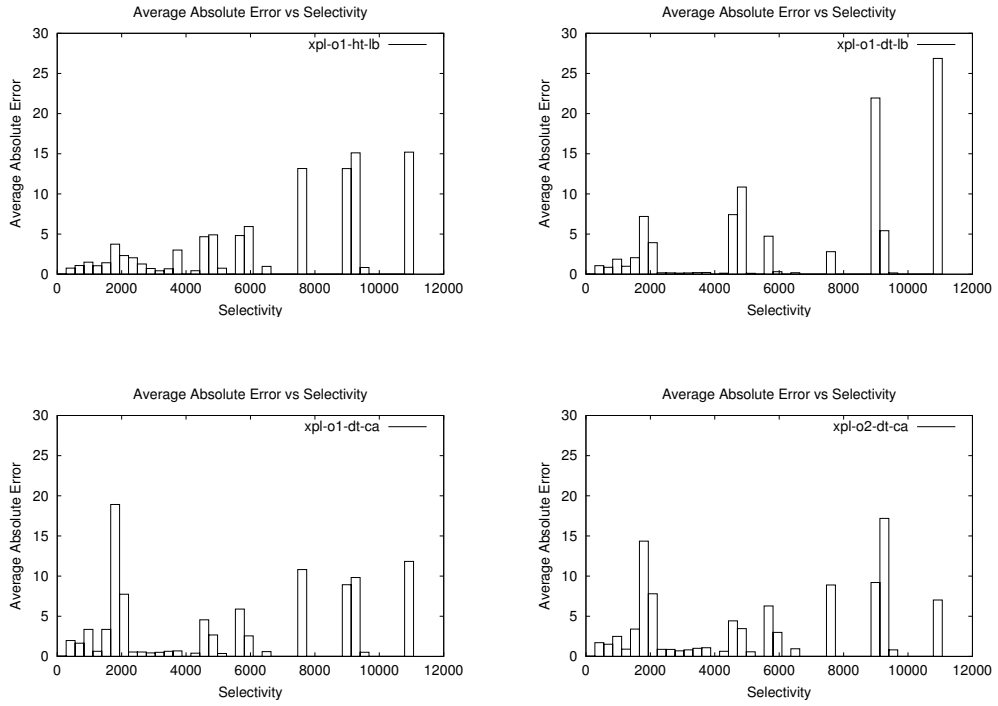


Fig. 15. **On-line error distribution across selectivities for the XMark data set.**

compressed histogram approach tend to produce larger errors for larger selectivities compared to the cache-based approach. This result is consistent with the on-line accuracy results in Section 5.4 where the compressed histogram approach is more accurate than the cache-based approach in terms of relative error, even though the cache-based approach is better in terms of absolute error.

5.6 The Frequency of Online Errors

We investigate how often on-line errors of a certain (relative) magnitude occurs. The on-line relative errors we used are for the same workload of 10,000 queries from the XMark data set used in Section 5.4. Each XPathLearner method is given about 7.3 KBytes of memory. The on-line relative errors are partitioned into 40 bins by their magnitude and the normalized count of the number of errors in each bin is plotted in Figure 16. Our results show that the relative estimation error is very small most of the time. Relative errors larger than 200% occur less than 2 % of the time.

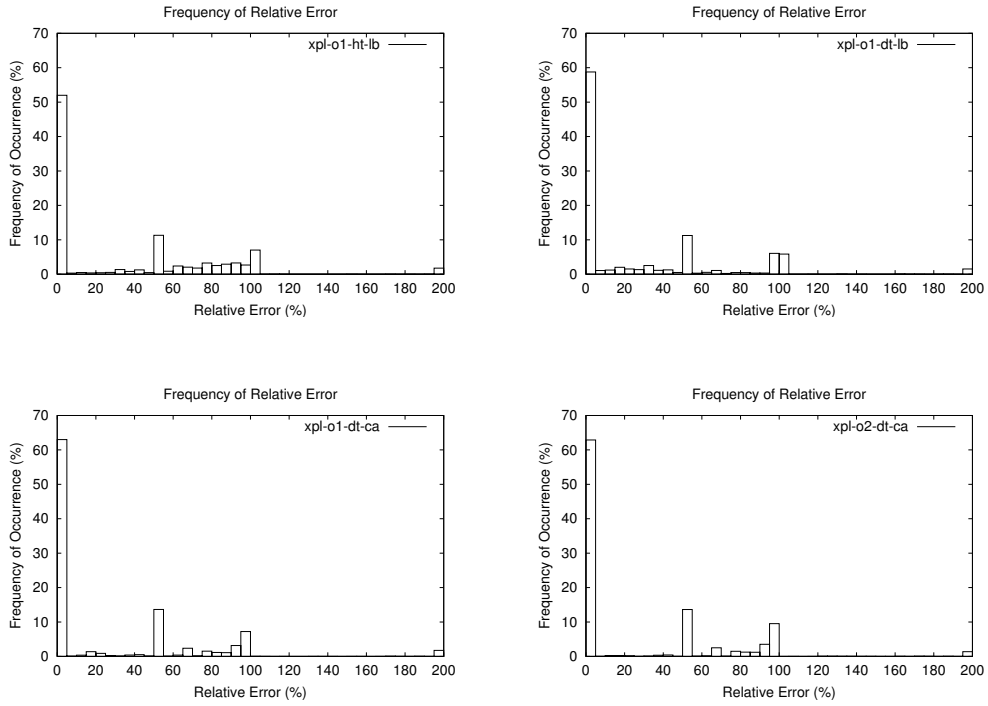
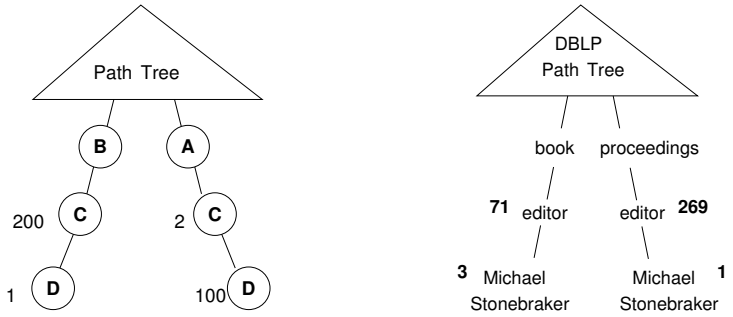


Fig. 16. **Frequency of on-line relative errors for the XMark data set.** The occurrence frequency of different ranges of on-line relative errors for an XMark query workload of 10,000 queries are plotted. More than half the errors that XPathLearner makes are relative errors of less than 5%.

5.7 Discussion

Our experiments have shown that XPathLearner is very accurate in terms of the traditional error measures as well as the on-line error measures we proposed. Moreover, we have shown that most of the the errors that XPathLearner makes are very small relative errors. XPathLearner adapts readily to changing data distributions and converges to very low error rates even under tight memory constraints.

What may be surprising is that XPathLearner can be even more accurate than the costly off-line method. We would expect the off-line method to be more accurate than the on-line method because the Markov histograms are constructed by scanning the data itself. The main reason for the on-line method being more accurate than the off-line is that the off-line method attempts to model all the data using the limited amount of given memory, whereas the on-line method uses the same amount of memory to model the portion of the data that is frequently queried in the workload. A secondary reason is that in constructing a Markov histogram over all the data, the off-line method can be affected by paths that violate the Markov assumption. Recall that for an order-1 Markov chain, the frequency of the next tag depends on the frequency



(a) An artificial example. (b) An example from the DBLP data.

Fig. 17. Two examples of a pair of paths that violate the order-1 Markov assumption.

of the current tag only. We illustrate effect these Markov-violating paths with an example.

Example of paths violating the order-1 Markov assumption. Consider the path tree in Figure 17(a) and the two paths `//A/C/D` and `//B/C/D` in the path tree. Assume that the tags `A, B, C, D` do not occur anywhere else in the path tree. The two paths `//A/C/D` and `//B/C/D` violate the order-1 Markov assumption because the count of `D` conditioned on being at `C` (via `B`) is 1/200 and the the count of `D` conditioned on being at `C` (via `A`) is 50. The violation is due to the large difference between the two conditional counts. In the off-line method, these two conditional counts will be aggregated and the entries $f(AC) = 2$, $f(BC) = 200$, and $f(CD) = 101$ will be stored. Hence, the selectivity of path `//B/C/D` will be computed as $\hat{\sigma}(BCD) = 200 \times 101 \div 202 \approx 100$ which has an absolute error of 99. For the on-line method, in the best case when the workload does not contain any path with `//A/C` or `//C/D`, the on-line method with delta rule will learn that $f(BC) = 200$ and $f(CD) = 1$. XPathLearner with delta rule will then give a more accurate estimate of `//B/C/D`.

The effect of these Markov-violating paths on the on-line method depends on whether the Markov-violating paths occur in the workload and how they are interleaved in the workload. The bottom line is that the on-line method has a chance of avoiding these Markov-violating paths, while the off-line method does not.

6 Conclusions

In this paper, we presented XPathLearner, a new method for estimating the selectivities of path expressions (simple, single-value, multi-value) without examining the XML data. Our method relies on the feedback from the query

execution engine to construct and refine a Markov histogram of the underlying path selectivity statistics. We also proposed two approaches to deal with the large number of single-value paths that allows us to estimate the selectivity of paths containing data values using our Markov histogram. We presented two update or refinement strategies—the heavy-tail rule and the delta rule—and evaluated their performance experimentally. Our experiments show that our method is accurate under modest memory requirements. As future work, we plan to extend the current fixed-order Markov model to a more general variable-order Markov model.

References

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

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

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (2nd edition). *W3C Recommendation*, October 6, 2000.

[4] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: a multidimensional workload-aware histogram. In Walid G. Aref, editor, *SIGMOD 2001*, pages 211–222. ACM Press, 2001.

[5] Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, Jerome Simeon, and Mugur Stefanescu. XQuery 1.0: An XML query language. *W3C Working Draft*, June 7, 2001.

[6] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting twig matches in a tree. In *ICDE 2001*, pages 595–604, 2001.

[7] James Clark. expat—XML parser toolkit, 2000.

[8] James Clark and Steve DeRose. XPath 1.0: XML path language. *W3C Recommendation*, November 16, 1999.

[9] Roy Goldman, Jason McHugh, and Jennifer Widom. From semistructured data to XML: Migrating themlore data model and query language. *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[10] H. V. Jagadish, Raymond T. Ng, and Divesh Srivastava. Substring selectivity estimation. In *PODS 1999*, pages 249–260, 1999.

[11] Michael J. Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Shapire, and Linda Sellie. On the learnability of discrete distributions. *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, pages 273–282, 1994.

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

[13] Michael Ley. DBLP XML records, 2001.

[14] Jason McHugh and Jennifer Widom. Query optimization for XML. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of 25th Intl. Conf. on Very Large Data Bases*, pages 315–326. Morgan Kaufmann, 1999.

[15] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, June 2001.

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

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

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald 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.

[19] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of 28th Intl. Conf. on Very Large Data Bases*, pages 974–985, Hong Kong, China, August 2002.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, 1979.

[21] XMark 100 MB standard dataset., 2002. http://www.xml-benchmark.org.

[22] Xyleme home page, 2001. http://www.xyleme.com.