XPathLearner: An On-Line Self-Tuning Markov Histogram for XML Path Selectivity Estimation

Lipyeow Lim[1]* Min Wang[2] Sriram Padmanabhan[2] Jeffrey Scott Vitter[1]† Ronald Parr[1]

[1]Department of Computer Science
Duke University
Durham, NC 27708, USA.
{lipyeow, jsv, parr}@cs.duke.edu

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

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 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, 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 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 several real data sets.

* Work was done when the author was visiting IBM T. J. Watson Research Center.

† Support was provided in part by the National Science Foundation through research grant CCR–9877133 and by the Army Research Office through grant DAAD19–01–1–0725.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 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 [18], 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.,

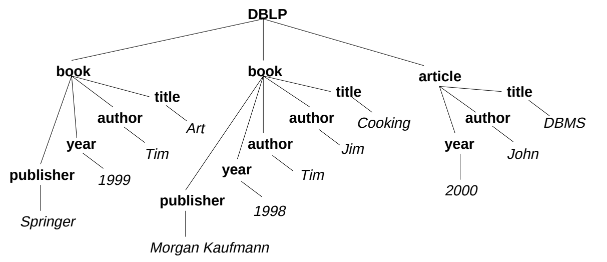


Figure 1: An example XML data tree. Tag names are in bold and data values are in italics.

`//book/title`) are called *simple path expressions.* Path expressions ending in a data value (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 *multivalue 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. Previous work is oblivious to workload distribution and consequently wastes 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 nor efficient because of the huge 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 expression. Our XPathLearner assumes a Markov model [14, 1] of path selectivities and learns this model from query feedback using error reduction strategies. Two such strategies are presented: the heavy-tail rule and the delta rule. Learning from a batch of query feedback tuples is also explored. Our XPathLearner overcomes limitations of previous work and has the following properties:

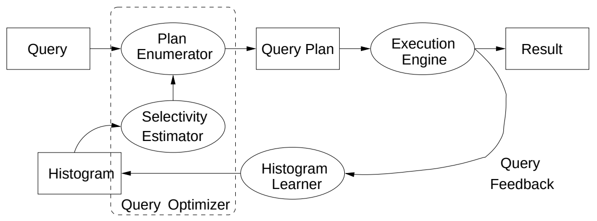


Figure 2: Workflow of XPathLearner. The top path is for query processing and feedback loop is for workload-driven selectivity estimation processing.

- 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 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 it 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 formulates the on-line XML path selectivity estimation problem. Section 4 presents our approach: the representation, the learning strategies, and the data structures. 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 1 of Section 4.1). The paths stored include both tags and data values, but no further summarization is performed. Since the number of possible data values in a big XML repository can be very large, the number of distinct paths with data values can be extremely large. The space requirement of the statistics used in the Lore system is therefore prohibitive for large repositories. Our XPathLearner overcomes this problem by using two data structures: one for the selectivities of paths involving tags only and the other for the selectivities of paths involving data values. For the paths involving data values, those with very high selectivity values are stored exactly and the rest are aggregated into buckets resulting in significant savings in space[1].

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 and is therefore an improvement over the method used in the Lore system because it reduces the space requirement. One limitation of the Markov table method is that paths with data values are not considered (i.e., it can only estimate selectivity of the simple path expression). 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 *path tree*[1] for estimating the selectivity of XML paths without data values. A path tree is a summarized form of the XML data tree. (We will define path trees formally in Section 3.) Tree pruning and aggregation techniques are proposed to manage the space requirement of the statistics stored in a path tree. Their experiments show 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. Subtree queries containing data values and substrings of the data values are considered. 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. show that their Markov table method is superior in accuracy [1].

The XML path selectivity estimation methods proposed in [1, 6, 14] are essentially summarization techniques for statistics that have been gathered after 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 distributions 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, learning a Markov model is 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. A Markov model of order $m - 1$ assumes that the selectivities of all the paths whose lengths are less than or equal to m capture all the required

[1]The way we store the statistics for paths involving data values is similar to a compressed histogram in traditional RDBMSs [17].

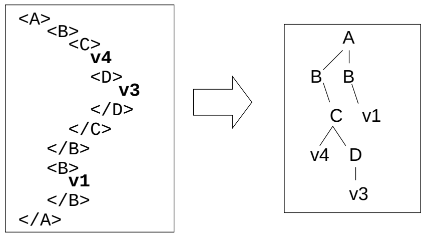


Figure 3: An XML document and its corresponding tree representation.

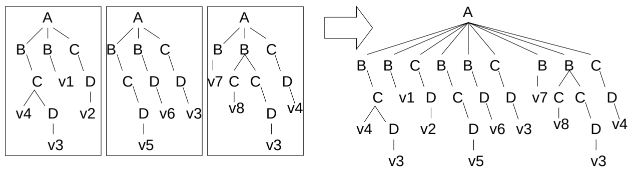


Figure 4: The XML data tree (right) is constructed from a repository of three XML documents (left). Alphabets in upper case denote tag names, 'v' followed by a number denotes a data value. The selectivity of the simple path expression //B/C/D is 3, the selectivity of the single-value path expression //B/C/D=v3 is 2, the selectivity of the multivalue 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.

statistics. The experiments in [1] show that, in practice, first-order 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 scanning the repository, (2) handles the three types of common query path expressions, and (3) is workload-aware.

3 Problem Formulation

In this section, we introduce basic terms and formulate 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 $|p|$ is a sequence of tags $\langle t_1, t_2, \ldots, t_{|p|}\rangle$, $t_i \in \Sigma$, 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 $//t_1/t_2/\ldots/t_{|p|}$ (e.g., `//book/title`). We will use as shorthand, where there is no confusion, the string $t_1t_2\ldots t_{|p|}$ to represent $//t_1/t_2/\ldots/t_{|p|}$.

A *multivalue path expression* is a simple path expression where values are associated with one or more tags in the path expression[2]. A special case of a multivalue path expression is a *single-value path expression* where only the last tag in the path is associated with a value.

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 multivalue path expression, and the path //B/C/D=v3 is a single-value path expression.

We denote the selectivity of a (simple, multivalue, 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 multivalue 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[3]. 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.

We now define our problem more precisely:

Problem Statement: Estimate the selectivity of a given path expression (simple, single-value, multivalue), given that statistics can only be obtained from query feedback and given only a fixed amount of space (memory) to store these statistics.

[2]A multivalue path expression is an example of a twig [6].
[3]We are investigating the extensions for more complicated wildcards in ongoing work.

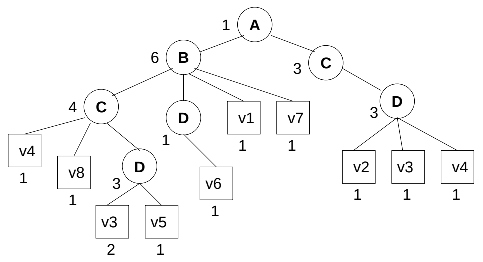
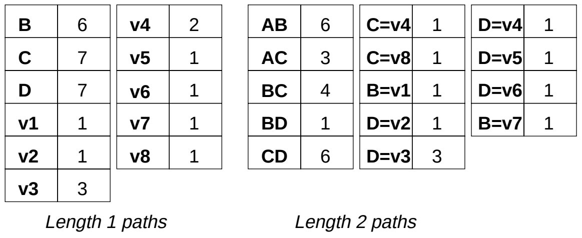


Figure 5: The path tree corresponding to the XML data tree in Figure 4. Circle nodes denote tag names and square nodes denote values.



B	6	**v4**	2
C	7	**v5**	1
D	7	**v6**	1
v1	1	**v7**	1
v2	1	**v8**	1
v3	3		

Length 1 paths

AB	6	**C=v4**	1	**D=v4**	1
AC	3	**C=v8**	1	**D=v5**	1
BC	4	**B=v1**	1	**D=v6**	1
BD	1	**D=v2**	1	**B=v7**	1
CD	6	**D=v3**	3		

Length 2 paths

Figure 6: The first-order Markov histogram corresponding the path tree from Figure 5.

4 Learning Path Selectivities

Solving the on-line XML selectivity estimation problem involves choosing a representation for the statistics and designing algorithms to update/refine the representation using information from query feedback. Our XPathLearner uses the Markov histogram representation. In this section we describe in detail the Markov histogram representation, the data structures it requires, and the on-line refinement algorithms. We also describe the path tree representation, because it is used in query workload generation and it can be viewed as an intermediate representation when transforming an XML data tree to a Markov histogram representation.

4.1 Representation

Path trees. A path tree [1] 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 (see Figure 5 for an example). With respect to the selectivity of simple path expressions and single-value path expressions, a path tree is a lossless summary of the XML data tree, but with respect to subtree selectivity (e.g., multivalue path expressions), a path tree is lossy. For example, consider paths ABC and ABD in Figure 5. It is not possible to determine whether a subtree AB{C,D} exists in the original XML data tree by looking at the path tree only.

Using a path tree, we can estimate the selectivity of a simple or single-value path expression p by summing the counts of all the nodes whose root-to-node paths contain p as a suffix. Observe that the counts of different nodes (whose root-to-node paths contain p as a suffix) contribute to the selectivity of the path expression p. Given a query feedback tuple, we want to use it to learn/refine the counts of all relevant root-to-node paths. However, the learning process for XML data is much more difficult than that for numeric attributes addressed in [2, 4]. For numeric attributes, when a query feedback (in terms of a range query and its corresponding real selectivity) is given, we know for certain that all the buckets overlapping the query range may contribute to the estimation error (i.e., the difference between the real selectivity and the estimated selectivity). The count(s) of these buckets can be refined in a straightforward way using heuristics [2, 4]. For XML data, it is not clear which root-to-node paths contribute to the estimation error when we try to update or refine the path tree using query feedback. For example, consider Figure 5. Suppose our XPathLearner predicts that the selectivity of the path CD is 2, but the query feedback is (CD, 6). How much of this estimation error of $6 - 2 = 4$ is due to the root-to-node path ABCD and how much of it is due to the root-to-node path ACD?

Markov Histograms. The Markov model has been used for XML path selectivity estimation in [14, 1]. The histogram that we use also assumes the Markov model and we call it the *Markov histogram*. A Markov histogram of order $(m - 1)$ is a table storing a set of distinct paths in the XML data up to length m along with their corresponding selectivities. For simplicity of presentation we consider $m = 2$ in this paper; however, the methods we describe can be easily generalized to higher-order Markov models. A first-order Markov histogram is conceptually a summary graph of the path tree where all nodes with the same tag are merged into one node and counts are associated with the incoming edges of the summary graph. The first-order Markov histogram is the edge list representation of this summary graph (see Figure 6). A Markov histogram summarizes the path tree using the assumption that the occurrence of a particular tag in a path expression is dependent only on the $m - 1$ tags that occur before it in that path expression. This assumption has been shown to hold for many real XML data sets for $m = 2$ or $m = 3$ [1, 6, 10].

Using a first-order Markov histogram, the selectivity of a simple path expression $p = t_1 t_2 \dots t_n$ can be estimated by

$$\widehat{\sigma}(t_1 t_2 \dots t_n) = \left(\prod_{i=1}^{n-1} P[t_i | t_{i+1}] \right) \cdot P[t_n] \cdot N, \qquad (1)$$

where N is the total number of nodes in the XML data tree, $P[t_i|t_{i+1}]$ is the probability of tag t_i occurring before tag t_{i+1} conditioned upon observing t_{i+1}, and $P[t_i]$ is the probability of tag t_i occurring in the XML data tree. Since $P[t_i|t_{i+1}] = f(t_i, t_{i+1})/f(t_{i+1})$ and $P[t_i] = f(t_i)/N$, where $f(p)$ is the count of path p ($|p| \leq m$) maintained by our Markov histogram, we can rewrite (1) as

$$\widehat{\sigma}(t_1 t_2 \dots t_n) = \left(\prod_{i=1}^{n-2} \frac{f(t_i, t_{i+1})}{f(t_{i+1})} \right) \cdot f(t_{n-1}, t_n). \qquad (2)$$

Note that in the off-line case, the count of a length-1 path $f(t_i)$ is the sum of the counts of all the length-2 paths having t_i as a suffix:

$$f(t_i) = \sum_{\alpha \in \{\text{tag/value names}\}} f(\alpha, t_i). \tag{3}$$

This equation is not true in the on-line case, because not all length-2 paths have been learned at a given time and for those that have been learned, the corresponding $f(\alpha, t_i)$ values are not necessarily the true values.

Our method assigns by default a selectivity of 1 to a negative query path, namely, a query path that contains a length-2 path not in the Markov histogram.

Computing the selectivity of a single-value path expression $p = //t_1/t_2/\ldots/t_{n-1}{=}v_{n-1}$ is the same as that for simple path expression except that the $f(t_{n-1}, t_n)$ term in Equation 2 should be replaced by the tag-value count $f(t_{n-1}, v_{n-1})$. The selectivity of a multivalue path expression $p = //t_1{=}v_1/t_2{=}v_2/\ldots/t_n{=}v_n$ can be estimated by

$$\begin{aligned}\hat{\sigma}&(t_1{=}v_1, t_2{=}v_2, \ldots, t_n{=}v_n) \\ &= \hat{\sigma}(t_1, t_2, \ldots, t_n{=}v_n) \cdot \left(\prod_{i=1}^{n-1} \frac{f(t_i, v_i)}{\sum_v f(t_i, v)} \right), \end{aligned} \tag{4}$$

where $f(t_i, v_i)/\sum_v f(t_i, v)$ can be viewed as the probability of the value v_i occurring after tag t_i, conditioned on observing t_i.

4.2 Handling Data Values

Our XPathLearner uses the Markov histogram representation. For first-order Markov histogram, the counts of all paths up to length 2 are stored. These length-2 paths can be tag-tag pairs or tag-value pairs. The counts of tag-value pairs need to be stored differently because the number of distinct data values is typically very large compared with 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. This translates to 841 possible tag-tag pairs and 2,664,462 possible tag-value pairs. Since the number of distinct tags is usually small, we store the counts of the tag-tag pairs exactly if there is enough memory. Otherwise, pruning or aggregation techniques in [1] can be used. For the counts of tag-value pairs, we use the bucketing technique described next.

Storing the counts of the tag-value pairs efficiently is similar to the problem addressed in [12]; however, we adopt a simpler approach, because we found empirically that most of the probability mass is concentrated in a very small number of tag-value pairs. This suggests an approach similar to [17] for storing the counts of tag-value pairs.

1. For the top k tag-value pairs with the largest counts, their counts are stored exactly, where k is a tunable parameter. Each entry in the "top k" data structure stores the tuple $\langle tag, value, count \rangle$.
2. Tag-value pairs with a count smaller than the minimum count among the top k tag-value pairs are aggregated into buckets. A bucket is a tuple $\langle feature, sum, numpairs \rangle$, where the field *numpairs* is the number of tag-value pairs it represents, the field *sum* is the sum of the counts of those tag-value pairs assigned to that bucket, and the field *feature* is the feature corresponding to that bucket. A tag-value pair is assigned to a bucket based on some feature of the tag-value pair.

For example, if a first-order Markov histogram is used and the first letter of a value is used as the bucket assignment feature, the data structure for storing the counts of tag-value pairs consists of at most k tuples of the form $\langle tag, value, count \rangle$ and at most $36 \cdot |\Sigma|$ tuples (where Σ is the set of tags) of the form $\langle feature, sum, numpairs \rangle$, 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 tag-value pair requires first searching through the top k entries. If the required pair is not found, the feature of the given tag-value pair is used to locate the corresponding bucket and *sum*/*numpairs* of the bucket is returned.

Update. Given a tag-value pair and an updated count, the top k entries are searched first and if a matching tag-value pair is found, its count is updated. Otherwise, we check if the updated count of the given tag-value pair 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 feature of the displaced entry. If it is smaller, the count of the given tag-value pair is added to the *sum* field of the corresponding bucket, and the *numpairs* field of the same bucket is incremented. Each bucket therefore encodes the average selectivity of all the current instances of the tag-value pairs belonging to that bucket.

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

4.3 An Example

Consider our earlier example in Figure 6. Figure 7 gives the corresponding representation using our Markov histogram with count parameter $k = 1$. We show how the selectivities of simple path expressions, single-value path expressions, multivalue path expressions, and simple path expressions with wildcard are computed.

tag	count
A	1
B	6
C	7
D	7

(a) tag counts

tag	tag	count
A	B	6
A	C	3
B	C	4
B	D	1
C	D	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

Figure 7: A 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'.

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

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

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

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

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

$$\widehat{\sigma}(//B/C = v4/D = v3) = \frac{f(BC)}{f(C)} \cdot \frac{f(CD)}{f(D)} \cdot f(D = v3) \cdot \frac{f(C = v4)}{\sum_v f(C = v)} = 0.735,$$

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

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

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

4.4 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[4].

Our two update algorithms follow the high-level steps outlined in Figure 8 and differ in the update equations used in line 7. 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.

[4]Batch updates have also been studied and are described in our full paper.

```
UPDATE(Mhistogram $f$, Feedback $(p,\sigma)$, Estimate $\widehat{\sigma}$)
1  if $|p| \leq 2$ then
2    if not exists $f(p)$
       then compress-add entry $f(p) = \sigma$
3    else $f(p) \leftarrow \sigma$
4  else
5    for each $(t_i, t_{i+1}) \in p$
6      if not exists $f(t_i, t_{i+1})$
         then compress-add entry $f(t_i, t_{i+1}) = 1$
7      $f(t_i, t_{i+1}) \leftarrow$ update
         /* depends on update strategy */
8    endfor
9  for each $t_i \in p, i \neq 1$
10   if not exists $f(t_i)$
       then compress-add entry $f(t_i)$;
11   $f(t_i) \leftarrow \max\{f(t_i), \sum_\alpha f(\alpha, t_i)\}$
12 endfor
```

Figure 8: The high-level pseudocode for updating the Markov histogram given a query feedback. The symbol '$\leftarrow$' denotes updates to existing entry

4.4.1 The Heavy-tail Rule

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

$$\epsilon = \sigma(p) - \widehat{\sigma}(p). \quad (5)$$

Recall from (2) that the selectivity of path p is computed as

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

We need to refine all the $f(t_i, t_{i+1})$ terms in this product based on the observed error ϵ. By (3), the updates to the $f(t_{i+1})$ terms are dependent on the $f(t_i, t_{i+1})$ terms 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:

$$f_{l+1}(t_i, t_{i+1}) \leftarrow f_l(t_i, t_{i+1}) + sign(\epsilon)\,(\gamma|\epsilon|)^{w_i / \sum_{j<n} w_j}, \tag{6}$$

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 $f_l(\cdot)$ and $f_{l+1}(\cdot)$ are the counts before and after the update, respectively. The weights we used are

$$w_i = 2^i, \qquad i = 1, 2, \ldots \tag{7}$$

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:

$$\prod_{i=1}^{n-1} (\gamma|\epsilon|)^{w_i / \sum_j w_j} = \gamma|\epsilon|. \tag{8}$$

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 overreacting to an estimation error and smoothes the error reduction process. The $f(t_i)$ terms are updated as

$$f_{l+1}(t_i) \leftarrow \max\{\sum_j f_{l+1}(t_j, t_i),\ f_l(t_i)\}, \tag{9}$$

since by (3) the sum of the counts for all length-2 paths ending in t_i must be a lower bound on the true $f(t_i)$. The term $f_l(t_i)$ could be greater than $\sum_j f_{l+1}(t_j, t_i)$, if XPathLearner previously encountered a query feedback with a length-1 query path t_i.

As an example, suppose that the Markov histogram maintained by XPathLearner is in the state as shown in Figure 7. Further suppose that the feedback for path ACD is $(ACD, 6)$ and its estimated selectivity is $\widehat{\sigma} = 3 \cdot 6 \div 7 \approx 3$. The observed error is $\epsilon = 6 - 3 = 3$, and using $\gamma = 1$, the following updates are made:

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

The estimated selectivity of the path ACD after the update is $4 \cdot 8 \div 8 = 4$. The estimation error is thus reduced by 1.

4.4.2 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. [16].

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

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

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

$$\begin{aligned}
w_{\alpha\beta}^{(l)} &= f_l(\alpha, \beta), &(11)\\
W_{\beta}^{(l)} &= f_l(\beta) = \sum_{\alpha \in \Sigma} w_{\alpha\beta}^{(l)}, &(12)\\
\sigma &= \sigma(p), &(13)\\
\widehat{\sigma} &= \widehat{\sigma}(p). &(14)
\end{aligned}$$

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

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

$$E = (\sigma(p) - \widehat{\sigma}(p))^2. \tag{15}$$

The delta rule states that the update to term w should be proportional to the negative gradient of E with respect to w:

$$w^{(l+1)} \leftarrow w^{(l)} - \gamma \frac{\partial E}{\partial w}, \tag{16}$$

where γ is the proportionality constant or learning rate. By (2), our update equation for the term $f(t_{n-1}, t_n)$ is

$$w_{\alpha\beta}^{(l+1)} \leftarrow w_{\alpha\beta}^{(l)} + 2\gamma\epsilon \frac{\widehat{\sigma}}{w_{\alpha\beta}^{(l)}}, \tag{17}$$

where $\alpha = t_{n-1}$ and $\beta = t_n$. Similarly, from (2) and (3) and the quotient rule for differentiation, the update equation for each $f(t_i, t_{i+1})$ term, for $i < n - 1$, is

$$w_{ab}^{(l+1)} \leftarrow w_{ab}^{(l)} + \frac{2\gamma\epsilon\widehat{\sigma}\left(W_b^{(l)} - w_{ab}^{(l)}\right)}{w_{ab}^{(l)} W_b^{(l)}}, \tag{18}$$

where $\alpha = t_i$ and $\beta = t_{i+1}$. The learning rate parameter 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 7. Assume again that the feedback for path ACD is $(ACD, 6)$ and its estimated selectivity is $\widehat{\sigma} = 3 \cdot 6 \div 7 \approx 3$. The observed error is $\epsilon = 6 - 3 = 3$ and using $\gamma = 0.5$, the following updates are made:

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

The estimated selectivity for path ACD after the updates is $5 \cdot 8 \div 9 \approx 4$. The estimation error is thus reduced by 1.

4.5 Update Overhead

Let the time needed to access an entry in the Markov histogram be $O(m)$, where m is the size of the data structure used to implement the Markov histogram. Let the query path in question be $p = t_1 t_2 \dots t_n$. The update equations for the heavy-tail rule method (see (6)) and the delta rule method (see (17) and (18)) both take $O(m)$ time. There are $O(n)$ iterations of the two loops in lines 5–8 and lines 9–12 of the update algorithm (Figure 8). Each iteration of the loop in lines 9–12 requires $O(m)$ time, since the summation in line 11 is over at most all the length-2 paths in the Markov histogram of size m. Therefore each update takes $O(nm)$ time, where n is the query path length and m is the size of the Markov histogram. Since n is bounded by the height of the XML data tree and m by the small amount memory allocated to store the Markov histogram, m and n are practically constants. Therefore the update overhead is a constant.

5 Experiments

We implemented our XPathLearner in C using the XML Parser Toolkit [7]. We describe in this section the experiments that we have done to validate our method. We describe briefly the data sets used, the query workloads, the performance measures, and the goals of the different sets of experiments.

Performance Measures. Two error metrics were used: the average relative error and the average absolute error. The average relative error ($a.r.e.$) and the average absolute error ($a.a.e.$) are defined as

$$a.r.e. = \frac{1}{n}\sum_{i}^{n} \frac{|\sigma_i - \widehat{\sigma}_i|}{\sigma_i}, \quad a.a.e. = \frac{1}{n}\sum_{i}^{n} |\sigma_i - \widehat{\sigma}_i|,$$

where n is the number of the query paths in the workload, σ_i is the selectivity of the ith query path in the workload, and $\widehat{\sigma}_i$ is the estimated selectivity for the ith query path.

Data Set. We performed our experiments on several real data sets: DBLP [13], Swiss protein[5], and Shakespeare[6]. For brevity, we present only the results from the DBLP data set in this paper. The XML data tree for the DBLP data set consists of 261,256 nodes. The corresponding path tree has a depth of 5 levels with 57 tag nodes and 109,741 value nodes. There are 29 distinct tags and 91,878 distinct values.

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 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[7]. (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 of the given XML data set. 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,

$$P[\text{choosing root-to-leaf path } p] = \frac{\sigma(p)}{\sum_{r \in R} \sigma(r)}, \qquad (19)$$

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.

The off-line method. The off-line method that XPathLearner is compared with differs from the on-line method in that the Markov histogram is constructed by scanning the repository. The off-line method 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 4.2.

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. There are four data structures in a first-order Markov histogram (see the example in Figure 7). These data structures store the counts of distinct tags (length-1 tag paths), the counts of distinct tag-tag pairs, the top k counts of distinct tag-value pairs and the aggregated counts of distinct tag-feature pairs. Each tag, value, count, and feature is stored as a four-byte integer. The count of a length-1 tag path requires two integers; the count of a tag-tag or tag-value pair requires three integers, and the count of a tag-feature pair requires

[5]http://www.expasy.ch/sprot

[6]http://metalab.unc.edu/bosak/xml/eg/shaks200.zip

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

four integers. The memory requirements of each method is accounted for by counting the number of entries in each of these data structures 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. These values were found to be reasonably good through experimentation.

5.1 Accuracy versus Space

In this experiment, we measure the estimation error under varying memory constraints. Two different query workloads 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[8], respectively.

The goal of this experiment is to see how our online 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

$$\text{workload\_diff(A,B)} = 1 - \frac{|S_A \cap S_B|}{|S_A \cup S_B|}. \tag{20}$$

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 9. The workload_diff of the training and testing workload we present is 88.4%. We measure the estimation error as the count parameter k varies from 32 to 4096. The k values (for the top k values) are then converted to memory usage in bytes and we plot the the estimation errors against memory usage. Our experiments show that in terms of absolute errors our on-line XPathLearner is more accurate than the off-line version. Among the two on-line update strategies, the delta rule is usually more accurate than the heavy-tail strategy. In terms of relative errors, the differences among the on-line methods and off-line method are within 10%.

The relationship between k and the memory usage in bytes for the off-line and on-line XPathLearner is graphed in Figure 10. For fixed k, the memory requirements 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[9]. 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 1.

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) is used in this experiment, and we measure the average

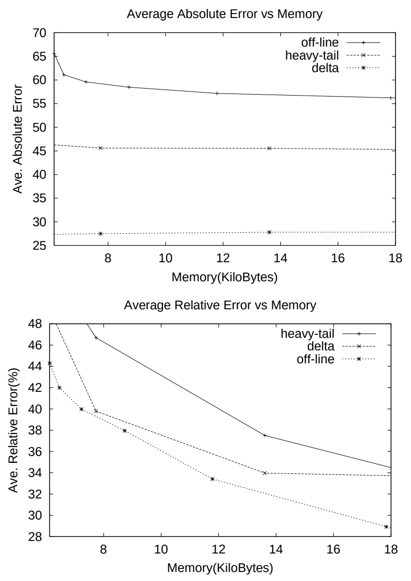


Figure 9: **Accuracy versus Memory.** Accuracy of the on-line Markov histogram method on a testing workload that is 88.4% different from the training workload. Both workloads contain 4096 single-value query paths, about 3100 of which are distinct.

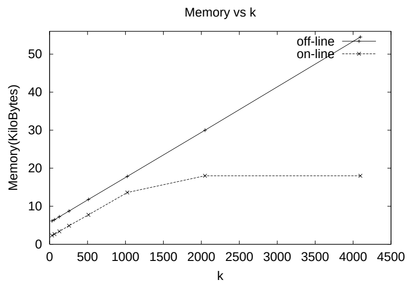


Figure 10: **Memory versus count parameter k.**

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

[9]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
On-line delta	0.086	0.197	764 Bytes
Off-line	0.110	0.331	796 Bytes
On-line heavy-tail	1.198	0.243	764 Bytes

Table 1: **Accuracy of various methods for simple path expressions.** Estimation error of the off-line method, on-line heavy-tail method and on-line delta rule method for workload consisting only of simple path expressions (tag-only path expressions). The online delta rule outperforms the others.

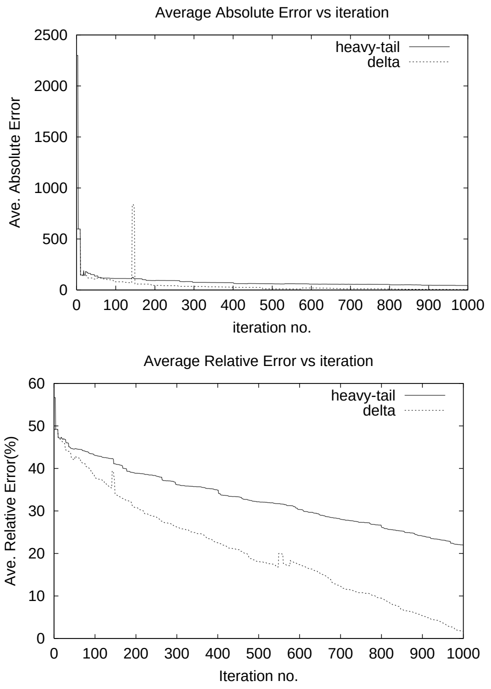


Figure 11: **Convergence with fixed count parameter $k = 512$.** The estimation error averaged over an entire workload at each iteration of the learning process. This experiment shows how the estimation error is reduced as our XPathLearner learns the path selectivities from query feedback

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 where all the tag-value counts are stored exactly ($k = 512$) are plotted in Figure 11. The results in Figure 11 show that the accuracy of our XPathLearner reaches very acceptable levels within the first 100 iterations. Figure 12 shows how the memory constraint governed by k affects the convergence properties of the heavy-tail and the delta rule update strategies. Our results show that XPathLearner can still be very accurate even when little memory is allocated for the tag-value counts. The spikes in Figure 11 and Figure 12 are due to bad paths[10].

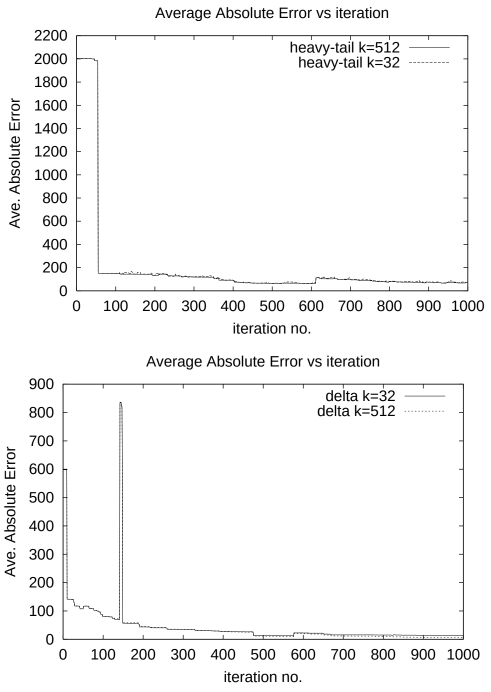


Figure 12: **Convergence versus count parameter k.** Estimation error averaged over an entire workload at each iteration of the learning process for two different values of k. The plot at the top is for the heavy-tail rule and the bottom one is for the delta rule. When $k = 512$, all the tag-value counts are stored exactly. When $k = 32$, only 32 tag-value counts are stored exactly.

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 scaled by the random number U,

$$count(r) \leftarrow \max\{1,\ U \cdot count(r)\}. \tag{21}$$

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

$$KL(f_0|f_1) = \sum_{y \in \{\text{root-to-node paths}\}} f_0(y) \log \frac{f_0(y)}{f_1(y)}. \tag{22}$$

[10]Please refer to our full paper for the definition of bad paths.

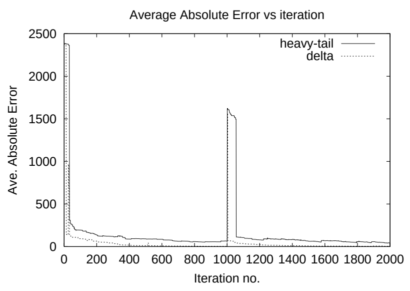


Figure 13: **Adaptability.** Average absolute error averaged over Q_{old} for iteration 1-1000 and averaged over Q_{new} for iteration 1001-2000.

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 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 spike at iteration 1001 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. However, about 100 iterations after the transition, the on-line method has adapted to Q_{new}.

Our experiments show that XPathLearner is very accurate and converges to very low error rates even under tight memory constraints. It adapts to changing data distributions and can even be more accurate than the costly off-line method.

6 Conclusions

In this paper, we presented XPathLearner, a new method for estimating the selectivities of path expressions (simple, single-value, or multivalue) 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 a method to deal with the large number of tag-value pairs 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. Our method can be generalized to arbitrary fixed-order Markov models. As future work, we plan to extend the current fixed-order Markov model to a more general variable-order Markov model.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB'01*, 591–600. Morgan Kaufmann, 2001.

[2] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999*, 181–192. ACM Press, 1999.

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

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

[5] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. *W3C Working Draft*, June 7, 2001.

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

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

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

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

[10] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, 249–260. ACM Press, 1999.

[11] M. J. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. E. Shapire, and L. Sellie. On the learnability of discrete distributions. *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, 273–282, 1994.

[12] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In H. V. Jagadish and I. S. Mumick, editors, *SIGMOD 1996*, 282–293. ACM Press, 1996.

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

[14] J. McHugh and J. Widom. Query optimization for XML. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99*, 315–326. Morgan Kaufmann, 1999.

[15] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2), 27–33, June 2001.

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

[17] 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 *SIGMOD 1979*, 23–34, 1979.

[18] Xyleme home page, 2001.