XML Summaries for Routing in P2P Systems

Marina Drosou
University of Ioannina
mdrosou@cs.uoi.gr

Abstract

Peer-to-peer systems are large-scale computer networks through which computers share resources and exchange data. A common way to describe such data is the XML language. Summarizing data lowers the time required to search for specific data items. One summarization option is the use of Bloom Histograms. Those Bloom Histograms can also be merged and exchanged among the participating computers and used to route queries among them.

Keywords: Peer-to-beer, Bloom Histograms, Routing

1. Introduction

Peer-to-peer (P2P) systems are computer networks formed by large numbers of computers, called nodes. Each node is connected with a small fraction of other nodes which we call its neighbors. Through these neighbors it can reach many other nodes in the network. P2P systems utilize the computer power of all those nodes to complete various tasks. The participating nodes are usually simple personal computers which do not have significant computational power on their own. However, through the sharing of resources among the nodes of the P2P system, they are able to speed up the time required to complete their tasks. P2P systems are also widely used for data exchange and this is one of the main reasons for their growth. Since the participating nodes can vary significantly in terms of the operating system they use and the way they interact with other computers, it is essential to find a universal way to describe the data they exchange. A widely spread solution for this problem is the use of the XML language (eXtensible Mark-up Language) [XML].

The problem we focus in this work is how to efficiently search for specific XML data in a P2P system. The volume of the data contained in all the nodes of such a system can be very large and therefore searching all of it is not practical. We instead turn our

attention to data summarization techniques since searches in summarized collections of data can be performed much faster.

To summarize the XML data that each node makes available to the network we use Bloom Histograms. A Bloom Histogram [Wang et al. (2004)] is a data structure that can summarize data and also give an estimation of the *frequency* of each data item (by the term frequency we mean the number of times that each data item is found in the initial data collection). XML data form tree structures. Bloom Histograms group paths of such trees together according to their frequency. More specifically, each path is placed in one group (or *bucket*) in such a way so that paths that are placed in the same bucket have similar frequencies. The paths of each group are combined and stored as a single bloom filter [Bloom, B. H. (1970)]. Thus, we end up with a 2-dimentional matrix $H(p, v)$ where p is a bloom filter and v is an estimation of the frequencies of the paths stored in p. Algorithms for the estimation of the frequency of a specific path and for the construction of minimal-error Bloom Histograms have been proposed in the past.

The estimation error of the frequency returned from a Bloom Histogram for a specific path depends on two factors: the number of buckets of the histogram and the size of its bloom filters. We experiment with these two factors and find out that the estimation error can be decreased by increasing the size of the bloom filters and/or by increasing the number of buckets. However, since the available size for the histograms is typically limited we cannot increase both of those sizes simultaneously. According to our experiments, there is a best combination of those two parameters for each specific data collection which depends on the number of XML paths it contains and the number of their distinct frequencies. Bloom Histograms however are generally small in size.

We also consider merging Bloom Histograms together. In this way we can combine pre-existing Bloom Histograms which reside in different nodes of a P2P system in a new, single Bloom Histogram. Merged Bloom Histograms can be used to combine information about distinctive data collections. We propose two merging algorithms: *Frequency-merge* and *Filter-merge*. The frequency-merge algorithm combines pairs of buckets of the pre-existing histograms based on the frequencies of those buckets while the filter-merge algorithm uses the similarity of the corresponding bloom filters instead. We experiment with merged histograms and observe that while the estimation error is larger than that of a regular (minimal-error) Bloom Histogram (as expected) it is comparable to it. Frequency-merge performs better when the size of the bloom filters increases. When that size is small, filter-merge out-performs frequency-merge.

The main advantage of merged Bloom Histograms is that all that is required for their construction are the two pre-existing histograms and not the whole data collections from which those two emerged. For this reason they can be used in unstructured P2P

systems as routing indices since their small size and ease of construction leads to efficient communication between the nodes of the network.

The rest of this paper is organized as follows. In Section 2 we present some background information on Bloom Histograms, their properties and Routing Indices. In Section 3 we discuss the applications of Bloom Histograms in P2P systems and present our merging algorithms. In Section 4 we present our experimental evaluation of regular and merged Bloom Histograms and discuss our findings. Finally, Section 5 concludes this paper.

2. Background Material

2.1 Bloom Histograms

A Bloom Filter [Bloom, B. H. (1970)] is a data structure that can represent a set of items $S = \{s_1, s_2, \ldots, s_n\}$ via the use of a bit vector of length m and k hash functions h_1, h_2, …, h_k. The bits of the bit vector can be set either to 0 or 1 and all the hash functions hash their input uniformly in $[1, m]$. Initially, all the bits of the filter are set to 0. To add an item s in the filter we apply every one of the hash functions on it and then set the bit at position $h_i(s)$ to 1, $1 \le i \le k$. We also say that an item r *matches* a specific bloom filter if after applying every one of the hash functions on it, all the bits at positions $h_i(r)$, $1 \le i \le k$, have a value equal to 1. It is possible for an item r to match a bloom filter even if it does not belong in the original set S from which the bloom filter has been created. However, if an item r does not match the bloom filter, it is not possible for it to belong in S. It has been proven [Bloom, B. H. (1970)] that the probability of such false positives is equal to:

$$\varepsilon = \left(1 - \left(1 - \tfrac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k} \quad \textbf{(1)}$$

where n is the number of items in S.

Bloom Histograms were introduced by Wang et al in [Wang et al. (2004)]. They are data structures that can be used for data summarization and selectivity estimation. We define as *frequency* of an XML path the number of times it can be found in a data collection. For a given data collection D, a Bloom Histogram is a 2-dimentional matrix $H(p, v)$ where p is a bloom filter representing a group of XML paths which can be found in D and v is a representative value of the frequencies of the paths in p. Thus, given a path $q \in D$, we can find a row i in H such that $q \in H_i.p$. We can then say that q's frequency is equal to $H_i.v$.

For example, assuming we would like to insert the data shown in Figure 1(a) into a Bloom Histogram, we'd end up with a histogram like the one shown in Figure 1(b) (where we denote as $BF(S)$ the bloom filter created from S).

As discussed in [Wang et al. (2004)], the estimation error of a Bloom Histogram for a specific XML path depends on two factors: First, the fact that the value $H_i.v$ returned is just a representative value for the whole group of paths and not just the path in question and second, the fact that the use of bloom filters adds an additional error. It can be proved though that the estimation error is bounded.

Wang et al also proposed a construction algorithm that, given a specific number of buckets, can produce a minimal-error Bloom Histogram.

Path	Frequency
/a	10
/a/b	10
/a/f/c	99
/a/e	101
/a/z	995
/a/s	1000
/a/o	1005

Figure 1(a). *Data collection*

Bloom Filter	Frequency
BF(/a, /a/b)	10
BF(/a/f/c, /a/e)	100
BF(/a/z, /a/s, /a/o)	1000

Figure 1(b). *Bloom Histogram*

2.2 Routing Indices

Routing Indices were introduced by Crespo and Garcia-Molina in [Crespo, A., Garcia-Molina, H. (2002)]. They provide the necessary means for a P2P node to choose the set of neighbors to which it should forward a specific query. For example, in Figure 2, if a query about a data item stored in *C* reaches *A*, then *A* only needs to forward the query to *B* and not *E* or *G*. There are many data structures that can be used as Routing Indices. What they all have in common is that they can summarize the data that is accessible through every outgoing link of a node. To achieve this, every node maintains two indices: A local index to summarize the data that is stored in the node itself and a routing index to summarize the data that is stored in parts of the network it can reach through its neighbors.

3. Bloom Histograms in P2P

3.1 Merged Bloom Histograms

Let $BH_1(p_1, v_1)$ and $BH_2(p_2, v_2)$ be two Bloom Histograms that represent two data collections D_1 and D_2 respectively. Let s_1 and s_2 be the size of BH_1 and BH_2 respectively. We would like to construct a new Bloom Histogram, say *BH*, which will

represent the data collection $D = D_1 \cup D_2$ and will occupy no more space than $s_{max} = \max\{s_1, s_2\}$.

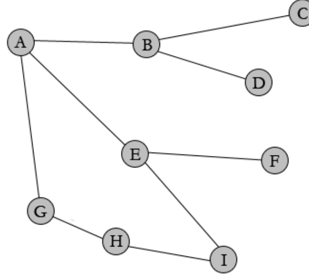

Figure 2. *Unstructured P2P system*

What we are most interested in is to keep the estimation error of the new histogram at a low level. To achieve this, our intuition tells us to try and combine similar pairs of buckets, one from each pre-existing histogram. The way we measure similarity can vary and will be further explained later. Bearing this in mind, we will now propose two merging algorithms: the *frequency-merge* and the *filter-merge* algorithms.

3.2 Merging Algorithms

The main concept behind the frequency-merge algorithm is to find pairs of buckets whose frequency values v_1 and v_2 are as close as possible, that is the value of $| v_1 - v_2 |$ is small.

When combining two buckets, the new bucket has a bloom filter $BF = (BF_1)$ BOR (BF_2), where BOR is the binary-OR operator, and a frequency value $v = (v_1 + v_2) / 2$. Pseudo-code for this algorithm is shown below (b_1 and b_2 are the number of buckets in BH_1 and BH_2 respectively).

```
Algorithm 1: Frequency-merge(BH1, BH2)
1: int min, target
2:  for i = 1 to b2 do
3:   min = +∞;, target = 0;
4:   for j = 1 to b1 do
5:    if abs(BH2.value[i] - BH1.value[j]) < min then
6:     min = BH2.value[i] - BH1.value[j];
7:     target = j;
8:    end if
9:   end for
10:   MergeBuckets(i, target);
11:  end for
```

The MergeBuckets function combines the BH_1's bucket at position *i* and BH_2's bucket at position *target* in a single bucket and adds it to the new histogram.

Instead of relying on the similarity of the frequency values in order to combine buckets, we can use the similarity of their bloom filters instead. Let B_1 and B_2 be two bloom filters of length *m*. We use $B[i]$ to symbolize the *i*-th bit of a bloom filter *B*. We define the *distance* $dist(B_1, B_2)$ between the filters B_1 and B_2 to be equal to the number of bits in which they differ. So,

$$dist(B_1, B_2) = |B_1[1] - B_2[1]| + |B_1[2] - B_2[2]| + \ldots + |B_1[m] - B_2[m]| \quad \textbf{(2)}$$

We then define the *similarity* between B_1 and B_2 as the quantity

$$similarity(B_1, B_2) = m - dist(B_1, B_2) \quad \textbf{(3)}$$

The higher this quantity is, the more similar the two bloom filters are.

When merging histograms using the filter-merge algorithm we try to combine pairs of buckets with similar bloom filters. We do this in order to produce new filters with few 1s in them, something which will lower the probability of the new buckets falsely reporting that various paths match with them and thus keeping the estimation error at a low level. The filter-merge algorithm is shown below.

```
Algorithm 2: Filter-merge(BH1, BH2)
1: int max, target
2:  for i = 1 to b2 do
3:   max = -∞;, target = 0;
4:   for j = 1 to b1 do
5:    if similarity(BH2.BF[i], BH1.BF[j]) > max then
6:     max = similarity(BH2.BF[i], BH1.BF[j]);
7:     target = j;
8:    end if
9:   end for
10:   MergeBuckets(i, target);
11:  end for
```

Since Merged Bloom Histograms can be constructed without the whole data collection being available, they can be used as Routing Indices in unstructured P2P systems.

3.3 Optimizations

There are a number of optimizations that can be used to increase the efficiency of the frequency-merge and filter-merge algorithms.

Firstly, using the average of the frequencies v_1 and v_2 of the two pre-existing buckets as the frequency v of the new bucket may work but is not an optimal solution. If for example v_2 is much larger than v_1, then v is shifted towards v_2 even if the number of paths in the second bucket is much lower than the number of paths that have been inserted into the first bucket. In this case, a value for v closer to v_1 would be desirable instead. One way to deal with such cases is to use a weighted average of v_1 and v_2 when calculating the value of v. To do this we need to maintain some extra information along with our Bloom Histograms, namely the number of paths that have been inserted into each bucket. Let k_1, k_2 be the number of paths inserted into the first and second bucket respectively. The frequency value v of the new bucket will then be:

$$v = \frac{k_1 \cdot v_1 + k_2 \cdot v_2}{k_1 + k_2} \quad \textbf{(3)}$$

The number of paths inserted into the new bucket will now be equal to $k = k_1 + k_2$.

Secondly, there is the case of the two pre-existing buckets having identical bloom filters. For example, let us assume that the two Bloom Histograms we would like to merge are the ones shown in Figure 3 and that our method of merging is frequency-merge.

	Bloom Filter	**Frequency**
A_1	BF(/a)	1000
A_2	BF(/b)	4000
A_3	BF(/c)	7000
A_4	BF(/d)	9000

	Bloom Filter	**Frequency**
B_1	BF(/a)	2000
B_2	BF(/y)	5000
B_3	BF(/w)	8000
B_4	BF(/z)	11000

Figure 3. *Pre-existing histograms*

In this case, buckets A_2 and B_2 will be combined together and the new bucket will have frequency 4500. This is a good estimation since a random path in the new bucket can be either */b*, which has frequency 4000, or */y*, which has frequency 5000. In a similar way buckets A_1 and B_1 will be combined together and the new bucket will have frequency 1500. However, all paths in this new bucket are */a* paths and according to the two pre-existing histograms we know that there are 1000 */a* paths in the first data collection and another 2000 */a* paths in the second data collection. We thus know that we have 3000 */a* paths in our data and can give this frequency value to the new bucket.

4. Experiments

4.1 Experimental Setup

To see how we can affect the estimation error of Bloom Histograms and evaluate the performance of Regular and Merged Bloom Histograms we implemented the algorithms proposed in [Wang et al. (2004)] as well as the two merging algorithms of Section 3. We used an XML parser [Xerces] to extract all the paths in a given XML data collection and then construct the corresponding Regular Bloom Histogram. The basic steps of this procedure are shown in Figure 4.

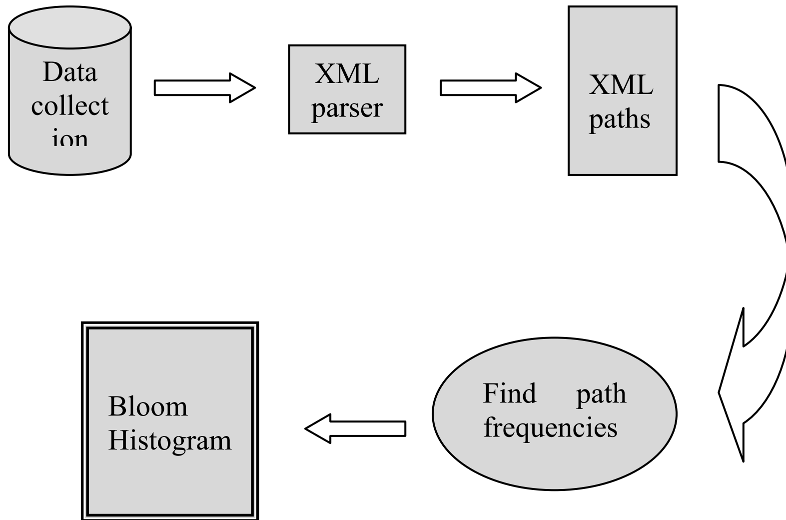

Figure 4. *Constructing Bloom Histograms*

We use three quite different data collections for our experiments:

- The first one is "SIGMOD record" [XML Data Repository], an index of articles in the Sigmod Record journal. It has only a handful of distinct paths which have high frequencies.
- The second one is "SwissProt" [XML Data Repository], a data collection which contains information about the molecular structure of various proteins. It is a large and complex data collection which contains wide-spread frequencies.
- For our third data collection, we used the XML data generator called XMark [XMark]. We created a collection which contains a high number of distinctive paths.

The main characteristics of the three data collections are summarized in Table 1.

Table 1. Data collections

	Size	# paths	Max frequence difference	# distinct frequencies	Min frequency	Max frequency
SIGMOD record	483 KB	31	3670	3	67	3737
SwissProt	112130 KB	335	566307	85	1	566308
XMark	75501 KB	1747	40924	250	1	40925

To measure the estimation error of a Bloom Histogram, the metric we use is the *absolute error*: Let us assume that we submit n queries to the histogram about various XML paths. Let us also assume that that the real frequency of the i-th path is X_i and that the estimation returned by the histogram for this path is V_i, $1 \leq i \leq n$. Then, the absolute error is equal to $\frac{1}{n}\sum_{i=1}^{n}\left|X_i - V_i\right|$.

4.2 Regular Bloom Histograms

This error is due to two factors. A part of it is due to the error associated with the bloom filters and the rest is due to the estimation error of the buckets. The goal of our first set of experiments is to quantify the percentage of error attributed to each of these factors.

Our experiments show that the total estimation error of a Bloom Histogram decreases as we increase the length of its bloom filter or the number of its buckets. Due to lack of space we'll present some of our most characteristic findings. For a complete evaluation refer to [Drosou, M. (2006)]. In Figures 5 and 6 we can see the total estimation error for two Bloom Histograms. We use histograms with bloom filters of length equal to 128 and 512 bits to summarize the “SwissProt” and “XMark” data collections respectively. The y-axis is presented in a logarithmic scale.

We see that the percentage of the total error that is due to the estimation error of the buckets decreases as the number of buckets increases. There is a point beyond which this percentage is almost zero. We also observe that this happens when the number of buckets is larger than the number of distinct frequencies in the data collection (see Table 1).

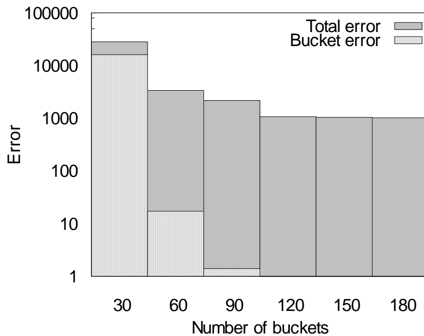

Figure 5. *SwissProt*

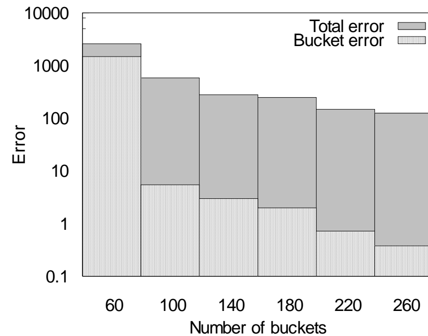

Figure 6. *XMark*

However, since the size available for a bloom histogram is typically limited, we cannot increase both the length of the bloom filters and the number of buckets to minimize the total estimation error. Therefore, we would like to know which factor is more important. To do this, we constructed a set of Bloom Histograms for each of the three data collections. The total size allocated for all the histograms is fixed. This means that as the number of buckets increases, the length of the bloom filters decreases. The space dedicated to the "SwissProt" and the "XMark" data collections is 320 bytes and 7.5 KB respectively. In Figures 7 and 8 we can see the percentage of the total error that is due to the bloom filters and the number of buckets.

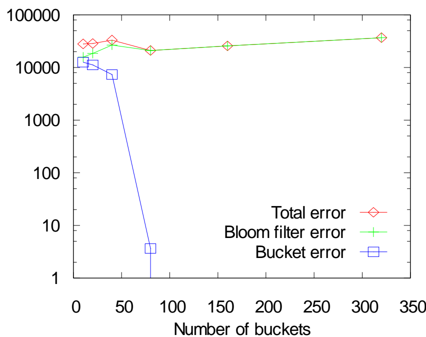

Figure 7. *SwissProt*

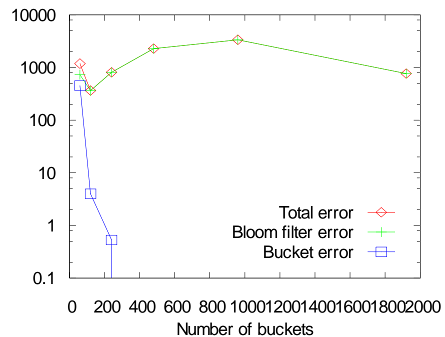

Figure 8. *XMark*

We see that as the number of buckets increases, the error due to them decreases and becomes equal to zero. However, as the number of buckets increases, we are forced to decrease the length of the bloom filters. Their false-positive probability is thus increased, something that increases the part of the error that is due to them. It is not clear which parameter we should generally increase further to minimize the total estimation error for every data collection. However, for each one of them we can initially construct a number of histograms and see what the optimal combination is.

For example, in Figure 7 we see that the best parameter combination for a 7.5 KB "XMark" Bloom Histogram is to have 180 buckets and devote the rest of the space to its bloom filters.

4.3 Merged Bloom Histograms

In the next set of experiments we compare the accuracy of Regular and Merged Bloom Histograms. More specifically, for each data collection we first construct a minimal-error bloom histogram *BH*. We also cut the collection in two halves and for each half we construct a minimal-error bloom histogram, BH_1 and BH_2. We then merge BH_1 and BH_2 into BH_b using frequency-merge. We also merge them into BH_f using filter-merge. In this way, *BH*, BH_b and BH_f all summarize the whole data collection but only *BH* has been constructed directly from it. We also implemented the optimizations discussed in Section 3.

For the rest of this section we use the following abbreviations: *MinErr* for Regular Bloom Histogram, *Freq* for Frequency-merge without optimizations, *FreqOpt* for Frequency-merge with optimizations, *Fil* for Filter-merge without optimizations and *FilOpt* for Filter-merge with optimizations.

In Figures 9 and 10 we see the see the total absolute error for "SwissProt" and "XMark". We observe that our optimizations work and lower the total error by 10%. The merged histograms have a larger estimation error that the regular ones, something we expected due to the way they are constructed. The estimations they provide however are comparable to the ones provided by the regular histograms. Their main advantage is that they are constructed without the use of the whole data collection. They are also small in size (like all Bloom Histograms). These features make them suitable to be used as Routing Indices in P2P systems because they can be easily exchanged between nodes and be merged in any of them. Frequency-merge seems to out-perform filter-merge, unless the resulting merged histogram has bloom filters of high false-positive probability. In those cases, filter-merge is preferable.

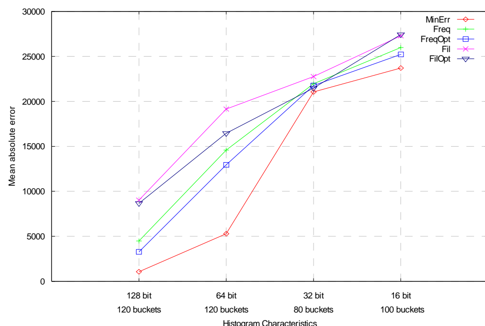

***Figure 9**. Frequency-Merge*

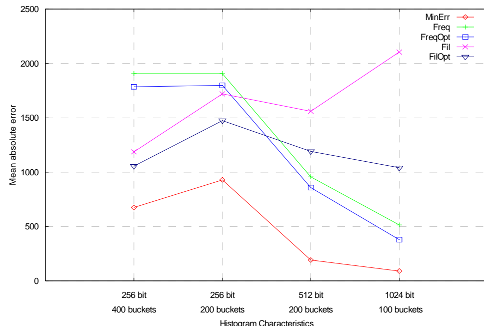

***Figure 10**. Filter-Merge*

5. Conclusion

Bloom Histograms are an interesting data structure that can be used in P2P systems to summarize data. We can also merge them together and create histograms containing information about many data collections without the need for those data collections to be locally available. Such Merged Bloom Histograms can be also used as Routing Indices among the nodes of an unstructured P2P system. In the future, it would be interesting to search for specific rules to associate the characteristics of a data collection with the efficiency of each of the two merging algorithms.

Acknowledgement

The work reported in this paper was performed as a part of the author's Bachelor Thesis [Drosou, M. (2006)] under the supervision of Associate Professor E. Pitoura.

References

Bloom, B. H. (1970), Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, vol. 13(7) pages 422-426

Crespo, A., Garcia-Molina, H. (2002), *Routing Indices For Peer-to-Peer Systems*, International Conference on Distributed Computing Systems

Drosou, M. (2006), Επεξεργασία και Δρομολόγηση XML Ερωτήσεων σε Συστήματα Ομότιμων Κόμβων, Bachelor Thesis, University of Ioannina

Wang, W., Jiang, H., Lu, H., Yu, J. X. (2004), *Bloom Histogram: Path Selectivity Estimation for XML Data with Updates*, Proceedings of the 30th VLDB Conference

Xerces, http://xml.apache.org/xerces-c/

XMark, http://monetdb.cwi.nl/xml/

XML, http://www.w3.org/XML/

XML Data Repository, *http://www.cs.washington.edu/research/xmldatasets/*