Distributed Structural Relaxation of XPath Queries

Georgia Koloniari and Evaggelia Pitoura

Computer Science Department, University of Ioannina, Greece
{kgeorgia,pitoura}@cs.uoi.gr

***Abstract*— Due to the structural heterogeneity of XML, queries are often interpreted approximately. This is achieved by relaxing the query and ranking the results based on their relevance to the original query. Query relaxation over distributed XML repositories may incur large communication costs, since partial result lists from different sites need to be gathered and ranked to assembly the overall top-k results. To process such queries efficiently, we propose using a *distributed clustered index* to group documents based on their structural similarity. The clustered index proves to be very effective in reducing the sizes of the partial lists that need to be combined. Furthermore, it can be used as the basis of a *pay-as-you-go* approach, where clusters of documents are accessed gradually providing the user with increasingly improving results. To reduce the cost of constructing and maintaining the clustered index, we use a compact data structure that trades-off accuracy for storage and communication efficiency. The index is also used for selectivity estimation so that query relaxation is geared towards the most promising structural transformations. Our experimental results show that our approach significantly reduces the communication cost for retrieving the top-k results, while maintaining a low construction cost for the clustered index.**

***Index Terms*— ignore**

I. Introduction

XML has evolved as a standard for data representation and exchange in the Internet. Due to the structural heterogeneity of XML data, queries are often interpreted approximately. Usually, this is achieved by relaxing the query and then, ranking the results based on their relevance to the original query [1]. Most previous research addresses the problem of processing approximate or top-k queries on XML data in a centralized setting [1], [2], [3], [4]. Our focus in this paper is on approximate processing over collections of XML documents that are *distributed* among a number of sites.

A problem with distributed processing is that a site cannot determine how much to relax a query without contacting the other sites. A straightforward solution would be to forward the query to all sites. Then, each site would relax the query independently to produce a list with the locally best k results and send it to the query origin, which would produce the final list of top-k results. However, the number of sites involved and thus, the communication overhead would be very large. Furthermore, some of the sites may not support approximate query processing capabilities.

To address the lack of processing capabilities and achieve scalability, we propose: (a) partitioning the documents into groups and assigning a coordinator (or superpeer) to each group to be responsible for computing the top-k results and (b) along the spirit of [5], instead of processing queries on actual data, maintaining appropriate indexes (i.e., data summaries) and relax queries based on these indexes.

Furthermore, we argue that, if, instead of partitioning the documents randomly, the index entries of similar documents are assigned to the same superpeer, then the number of sites and documents that need to be considered during relaxation will be reduced. We call the distributed index produced this way *clustered index*. In a sense, our clustered index is analogous to the clustered indexes used in centralized applications to minimize the I/O cost of query evaluation, by placing similar data close to each other in disk. Our clustered index aims at reducing communication and processing costs by placing similar data at the same superpeer. Let us substantiate this claim.

Threshold-based Processing. The process that is followed to retrieve the top-k results proceeds in phases. When a query is issued by a site, the site propagates it to a superpeer. In the first phase (*Local Query Evaluation*), the superpeer forwards the query to the other superpeers. In parallel, each superpeer evaluates the query against its part of the index to attain k results. Note that, with results, we refer to index entries and not to the actual data that are stored at remote sites. These results are sorted locally at each superpeer, in non-decreasing order, according to their distance from the original query.

By following a procedure commonly used in a family of distributed top-k evaluation algorithms (*Threshold-based Algorithms* [6]) for reducing communication costs, in a second phase (*Elimination Phase*), superpeers exchange the distance scores of their k-th result (the result in the list with the largest distance from the original query). Upon receiving these distance scores, each superpeer discards the results with distance score larger than the smallest of all the distance scores it has received. In the third phase (*Result Construction*), each superpeer forwards the remaining results to the superpeer from which it has received the query. After the initial superpeer has gathered all lists, it merges them to construct the final result list. Then, the site that issued the query contacts the sites in the ranked list to retrieve the actual data. If some sites lack the required processing capabilities, they forward their data to the initial superpeer to perform the actual query processing.

The elimination phase of the routing process can consist of more than one round for further pruning the result list and

thus reducing the size of the data that needs to be transmitted. In the second round of this phase, the superpeers exchange the distance score and the position in their ranked list of the last result (in ascending distance order). Let *m* be the smallest distance score among all the distance scores sent. Each superpeer compares *m* to its own list and finds all results with a distance score larger than *m*. If the sum of the position of such a result with the position of *m* is greater than k, then this result can be safely discarded, since it does not belong to the final result list. The elimination phase can then proceed with the next round taking into account the newly acquired pruned lists in a similar manner to further reduce the number of results.

Assume N superpeers, each maintaining a part of the distributed index U_i, $1 \leq i \leq N$. Let us denote each iteration (round) of the elimination phase of the query processing algorithm as $round_j$. We define the *pruning degree* to measure the number of results that are eliminated in each round of the elimination phase as follows:

Definition 1 (Pruning Degree): For each part U_i of the distributed index and each $round_j$ of the elimination phase, the pruning degree $(PD_j(U_i))$ is defined as:

$$PD_j(U_i) = \frac{Eliminated_{ij}}{k}, \tag{1}$$

where $Eliminated_{ij}$ is the number of eliminated results in the partial results list of superpeer i responsible for part U_i. We consider the average pruning degree (PD_j) for each $round_j$:

$$PD_j = \frac{\sum_{i=1}^{N} \frac{Eliminated_{ij}}{k}}{N} = \frac{\sum_{i=1}^{N} Eliminated_{ij}}{N * k} \tag{2}$$

The larger the pruning degree, the more efficient the elimination phase, since less data needs to be transferred to the superpeer that initiated a query. Furthermore, a large pruning degree indicates that less processing is required for constructing the final result.

Motivation for a Clustered Index. Using a clustered index increases the pruning degree and thus reduces the number of documents that need to be considered during relaxation. To see this, consider the following simple example. Assume a set $\mathcal{D}$ of XML documents that can be classified in N categories (C_i) each having x documents, such that if a query q matches a document $d' \in C_t$, $1 \leq t \leq N$, it does not match any documents belonging to any other category. Also, assume that there are more than k exact matches for q. We partition the documents to N superpeers: (a) uniformly at random, where x/N documents of each category fall into each superpeer and (b) by assigning the documents of each category to a different superpeer. In (a), the results of q are expected to be distributed uniformly among the superpeers while in (b), the results reside at a single superpeer. For simplicity, we assume that all non exact matches have the same distance from q. While in (a), all exchanged distance scores have the same value and no pruning is possible, in (b), the superpeer of C_t has a k-th distance score equal to 0 and all other superpeers can prune their entire lists. This simple example shows that a clustered index can increase the pruning degree, when there is similarity among the documents.

Using a distributed clustered index provides another advantage. In particular, it enables an alternative query evaluation strategy that follows the principles of a *pay-as-you-go* approach. That is, the query evaluation proceeds incrementally to reduce the time that a user has to wait to obtain results. The pay-as-you-go strategy exploits the locality of the entries at each index partition. With the clustered index, query processing may start with the most relevant to the query cluster (for instance, with the cluster with the most similar to the query entries) and proceed gradually to access the remaining clusters. This way, we "prune" the number of sites that need to be considered for processing the query by excluding the potentially irrelevant ones. The processing and communication cost increase, while users receive more responses to their queries, i.e, gradually paying for the results they get.

In a nutshell, our clustered-index approach to relaxation works as follows. An index entry (or summary) is constructed for each document. The indexes are clustered and assigned to superpeers. The superpeers cooperate to relax the query by either pruning the result lists or the number of sites to be considered.

Paper Roadmap and Contributions. The main premise of this paper is that: *a clustered index can improve the performance of distributed approximate top-k processing.* To deploy our clustered index for distributed top-k processing of XPath queries, we need to resolve a number of important issues. In addressing them, the paper also makes the following contributions:

- We provide a distance measure for ranking the results in the top-k list based on structural transformations. These structural transformations are utilized by our relaxation algorithms. The novel aspect of these algorithms is that they direct the relaxation process towards those transformations that would result to the most results based on selectivity estimations provided by the distributed index. The selectivity-based relaxation algorithms are introduced in Section II.
- We propose a compact hash-based index structure, termed *Depth Bloom Histogram*, that provides selectivity estimations, can be updated incrementally and can be used for clustering. This index is described in Section III.
- We integrate the different components to work together efficiently with a threshold-based query evaluation procedure in a distributed setting. Their deployment is described in Section IV.

The other sections of the paper are organized as follows. Section V includes our experimental evaluation, while Section VI refers to related research. Extensions of the basic approach are discussed in Section VII and a summary is given in Section VIII.

II. Selectivity Driven Relaxation

XML employs a tree-structured model for representing data. In particular, we can model an XML document as a node-

labeled tree $Tree(V, E)$. Each node $e_i \in V$ corresponds to an XML element with a label assigned from some string literals alphabet that captures element semantics. Edges $(e_i, e_j) \in E$ capture the containment of element e_j under e_i. Any subtree of the XML tree is called an *XML fragment*.

We focus on queries that belong to an XPath subset, XPath$^{\{/,//,*\}}$, consisting of expressions of the form: $a_1p_1a_2p_2a_3...a_np_n$, where $p_i \in V \cup \{*\}$, $*$ is the wildcard operator and a_i is the child or descendant-or-self axis ("/" and "//"). Such path expressions form the building blocks of more advanced querying. We also handle twig queries (e.g. tree-pattern queries) by decomposing them to appropriate path expressions in XPath$^{\{/,//,*\}}$ and process them separately. A path expression is evaluated sequentially by finding an element p_1 anywhere in the document and nested within it an element p_2, and so on, until p_n is encountered. Its result is the set of fragments rooted at the p_n nodes found in the given XML tree. We say that a query q *matches* a collection of documents $\mathcal{D}$, if the result set of the evaluation of q against the documents in $\mathcal{D}$ is non-empty. We denote as *result(q, D)* the fragments included in the result set.

A. Distance Measure and Structural Transformations

There has been a lot of work on defining non exact-match semantics for XML [1], [2], [3]. To show the benefits of a clustered index, in this paper, we focus on structural relaxations. We first define a distance measure for ranking the results. Such a measure should not assume any knowledge about the global data distribution, since this is not realistic in a distributed setting. To this end, we rely on a variation of the edit-distance that counts the number of mismatching element tags between two path expressions of the same length. To compare two path expressions with different lengths, we define a function $ap^*(p, n, j)$, which given a path expression p of length l, and a number n ($n \geq 0$), appends n path steps at the $j + 1$ position in p with the wildcard as their element tag. That is: $ap^*(p, n, j) = //p_1/p_2/\ldots/p_j/p_{j+1}/\ldots/p_n/\ldots/p_l$, where $p_i =$"*", for $j < i \leq j + n$. To compare two path expressions p and p' of length l and l' respectively with $l < l'$, it suffices to use the ap^* function on p, with $n = l' - l$ and $j = l$. If p contains the "//", we set j equal to the position of the "//" and n equal to l'. If $l = l'$, then $ap^*(p, 0, j)$ just substitutes "//" with "/". In the computation of distance, we consider that the wildcard operator does not match any element tag.

Definition 2 (Distance Measure): The distance, $dist$, between two path expressions p and p', with lengths l and l' respectively, is defined as:

$$dist(p, p') = \begin{cases} dist(ap^*(p, l' - l, l), p') = \sum_{i=1}^{l'} diff(p'_i, p_i)/l', \\ \text{if } l \leq l' \\ dist(p', ap^*(p', l - l', l')) = \sum_{i=1}^{l} diff(p'_i, p_i)/l, \\ \text{if } l > l' \end{cases}$$

where p_i and p'_i denote the elements in position i in p and p' respectively, and $diff(p_i, p'_i) = 0$ if $p_i = p'_i$ and 1 otherwise.

To acquire approximate results for a query, we generalize the original query, by applying a number of structural transformations to it. Our transformations are in the spirit of [1], but simplified for linear path expressions.

Definition 3 (Structural Transformations): Let P be the set of path expressions. A structural transformation T is a function $T : P \to P$ that maps a path expression $p = //p_1/p_2/.../p_n$ into a new path expression $T(p)$. In particular, we define the following three structural transformations:

- truncation of the last element of p: $Trunc(p) = //p_1/p_2/.../p_{n-1}$
- replacement of an element tag at position i of p with the wildcard operator: $RepTag(p, i) = //p_1/p_2/.../p_{i-1}/*/p_{i+1}.../p_n$.
- replacement of a "/" axis in position i with the "//" axis: $RepAxis(p, i) = //p_1/p_2/...p_{i-1}//p_i/.../p_n$.

We associate a *cost* with each structural transformation, evaluated by calculating the distance between the original path expression p and the relaxed path expression $T(p)$.

Definition 4 (Transformation Cost): The cost of each structural transformation is defined as:

- $cost(Trunc, p) = dist(Trunc(p), p) = 1/length(p)$.
- $cost(RepTag, p) = dist(RepTag(p, i), p) = 1/length(p)$.
- $cost(RepAxis, p) = dist(RepAxis(p, i), p) = dist(//p_1/p_2/... /p_n, /p_1/p_2/\ldots/p_{i-1}/ *_1 /.../ *_x /p_i/.../p_n) = x/length(p)$.

The cost of the last transformation is defined so that "//" is equivalent to inserting a specific number of simple steps in the path expression with the wildcard operator.

Note that cost is not defined for all pairs of path expressions. For example, it holds that $dist(a/*/b, a/c/b) = dist(a/c/b, a/*/b) = 1/3$. However, we can determine only the cost for transforming *"a/c/b"* to *"a/*/b"* which is 1/3, and not vice versa.

Let $T^t(p)$ be a sequence of t transformations applied to p. That is, $T^t(p) = T(T^{t-1}(p))$. We define the cost for a sequence of transformations as: $cost(T^t, p) = cost(T^{t-1}, p) + cost(T, p)$. If we denote as q^t the result of a sequence of transformations T^t applied to q, it is straightforward that: $dist(q, q^t) = dist(q, q^{t-1}) + cost(T, q^{t-1})$, where T denotes the t-th transformation in the sequence. A direct consequence is the following property:

Property 1 (Monotonicity): The transformation functions are *monotonous* with respect to the distance measure $dist$. That is, $dist(q, q^i) \leq dist(q, q^{i+1})$.

A second important property guarantees that if a an XML fragment f matches the original query q, it also matches any query that is derived after the application of any possible combination of transformations on q. In particular:

Property 2 (No-loss Guarantee): If fragment $f \in result(q^{i-1}, D) \Rightarrow f \in result(q^i, D)$.

This is because the transformation functions guarantee that if a query q matches a document d, then $q' = T(q)$ also matches d (detailed proof in [7]). This property ensures that by relaxing a query, we do not lose any results that may exist. Furthermore, it holds that: $|result(q^i, D)| \geq |result(q^{i-1}, D)|$.

B. Relaxation Algorithms

The *relaxation* algorithm takes as input a query and gradually applies a combination of the available transformations

to it so as to attain the user-specified number of results. A central issue is the termination condition, that is, how much we need to generalize the query to attain the required number of results. We propose deriving the termination condition by exploiting the distributed index. In particular, both the order and the number of transformations that the algorithm applies is driven by two factors: the quality of the results and index selectivity estimations. Note that our index ensures that queries with results in the data have non-zero estimated selectivity.

We propose three different variations of the relaxation algorithm that vary on the portion of the total search space that each one explores.

Dynamic Programming Relaxation Algorithm. We introduce an exhaustive relaxation algorithm that applies at each iteration the transformation that will result in a query with the smallest possible distance from the original one. The index is consulted so as to consider only transformations with non-zero estimated selectivity. Furthermore, for transformed queries with the same distance from the original query (e.g. when we apply *RepTag* or *Trunc* to the same query), the index is used to select the one with the largest estimated selectivity.

If all one-step transformations result in queries with zero selectivity, we proceed by applying a second relaxation step to the already relaxed queries. When there are transformed queries that have no results with regards to the index, but have a smaller distance from the original query than that of the best relaxed query with non-zero selectivity, the algorithm continues to apply transformations to them. The process stops only when each candidate query has either non-zero selectivity or no more transformations can be applied to it or its distance is larger than that of a query with non-zero selectivity. Thus, the algorithm ensures that the selected transformations are the ones that will result in the smallest loss of quality, that is, the transformations that will lead to a query with the smallest distance from the original query. It can be shown that (formal proof in [7]):

Property 3 (Correctness): The relaxation algorithm chooses at each iteration the transformation that has the smallest possible distance from the original query among those that have not been applied yet.

To improve the complexity of the relaxation algorithm, we use a dynamic programming technique (Alg. 1). The intermediate distances calculated at each iteration are stored in auxiliary tables ($qlist, qw, dw, q_1$) to avoid recomputing them at each iteration. An instance of Alg. 1 is shown in Fig. 1.

Property 3 ensures that at each iteration the returned results have greater (or equal) distance from the input query than the results of the previous iteration. This can be exploited to stop processing before attaining k results. In particular, each superpeer that has computed a number of k/x results, ($x \geq 1$) can forward the distance score of the last result to the other superpeers. After comparing their distance scores, each superpeer can decide whether it needs to continue the relaxation process or not. This way we move the pruning phase of the query evaluation earlier in the method, thus saving on processing cost at each superpeer. Note that if a random index is utilized, we do not expect to have a large gain, but with a clustered index, superpeers responsible for clusters irrelevant to the query stop their processing much earlier.

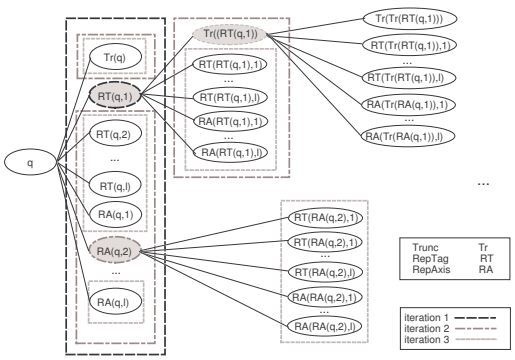

Fig. 1. Dynamic programming algorithm. The rectangles include the candidate states (transformation sequences) considered at each iteration. The shaded oval is the best transformation at each iteration.

Greedy Relaxation Algorithm. We introduce an alternative *greedy* relaxation algorithm. The main difference with the dynamic programming algorithm is that at each iteration, only a single query is considered (Fig. 2(left)). In particular, at each iteration the index is consulted to select among all one-step transformations the one that yields the largest number of results. This relaxed query is then used as input to the next iteration, until the required number of results is attained.

The algorithm may relax a query up to the most general query (i.e. "//*"), especially for large values of k. To avoid this, we use a simple heuristic in which the algorithm stops relaxing the selected query when its distance from the original query exceeds a threshold ($MaxDist$). It then uses backtracking to return to the iteration after which the largest increase in distance so far was observed. Then, the process continues by applying a different transformation to the relaxed query of that iteration.

Unlike the dynamic programming algorithm, the greedy one does not guarantee that the returned results are the actual top-k results. Thus, the greedy algorithm trades-off correctness for efficiency, since it applies a much smaller number of transformations and index look-ups by reducing the search space it considers.

Random Walks Relaxation Algorithm. To improve the quality of the results of the greedy algorithm without reaching the complexity of the dynamic programming one, we introduce a hybrid technique that considers M relaxed queries at the first iteration and applies the greedy technique in parallel to each of them (Fig. 2(right)). After each iteration, the number of all returned results is compared to k to determine the termination of the process. Instead of selecting M queries randomly, we bias the walk by selecting the M queries with the largest pairwise distances. The goal is to drive relaxation towards queries that lead to a larger number of non-overlapping results. The random walks approach also does not ensure correctness. However, we expect that it provides results with better quality, since it explores a larger part of the search space.

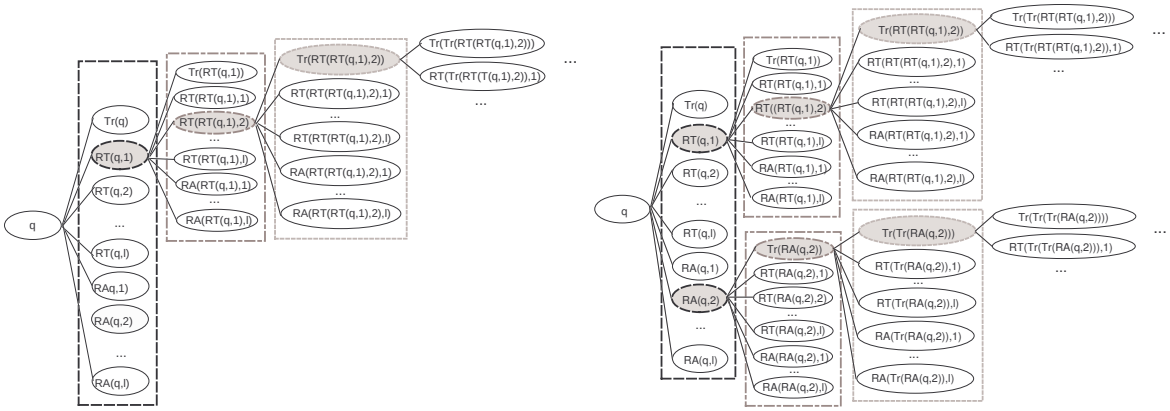

Fig. 2. (left) Greedy and (right) random walks with $M = 2$. Graphical notation as in Fig. 1.

Algorithm 1 DynamicProgRelax(I, k, q)

I: Index, k: # of results, q: Query

```
qlist = {(q, 0)} //Input query list with pairs of the form (expr, dist(expr, q))
q_MIN = NULL //Selected query for this iteration
qw, dw, q_can, q_1 //Lists for storing intermediate results
NUM = 1 //Number of queries in qlist currently
K = 0 //Number of retrieved results so far
result_list = NULL
while (1) do
    for j = 1 to NUM do
        q_1[j].expr = Trunc(qlist[j].expr)
        q_1[j].dist = qlist[j].dist + 1
        len =Number of path steps in qlist[j].expr
        for i = 1 to len do
            qw[j, i].expr = RepTag(qlist[j], i)
            qw[j, i].dist = qlist[j].dist + 1
            dw[j, i].expr = RepAxis(qlist[j], i)
            qw[j, i].dist = qlist[j].dist + x
        end for
    end for
    q_can:exprs with min dist value in qw, dw, q_1
    q_MIN:The query in q_can with the largest EstRes according to the index
    K = K + EstRes(I, q_MIN)
    Add q_MIN to result_list
    if K < k then
        NUM = NUM + 1
        qlist[NUM] = q_MIN
    else
        RETURN result_list
    end if
end while
```

Path	Count
/a	10
/b	20
/c	100
/d	250
/a/b	20
/b/c	100
/b/d	250
/a/b/c	100
/a/b/d	250

DBH (buckets=2, levels=3)

DBH_0	BF(/a,/b)	15	b_1
	BF(/c,/d)	175	b_2
DBH_1	BF(/a/b)	20	b_1
	BF(/b/c,/c/d)	175	b_2
DBH_2	BF(/a/b/c)	100	b_1
	BF(/a/b/d)	250	b_2

Fig. 3. (left) Full path count table and (right) the corresponding DBH

III. Index Structure

We assume a collection of documents distributed in a number of sites. To achieve scalability, instead of contacting all sites to process a query, documents are summarized. The summaries of the documents are aggregated and these aggregated summaries are assigned and maintained by special sites that play the role of *superpeers*. The indexes or summaries must (a) be compact, so that the cost of transmitting the entries among the various sites is small, (b) representative of the document structure and capable of providing selectivity estimations, and (c) incrementally updatable through a cost-effective procedure. They should also lead to an effective aggregation that increases the pruning degree of the relaxation procedure. To this end, we use a data structure based on Bloom filters.

A. Depth Bloom Histogram

Bloom filters [8] are compact data structures for probabilistic representation of a set that support membership queries. Consider a set $A = \{a_1, a_2, ..., a_n\}$ of n elements. The idea is to allocate a vector v of s bits, initially all set to 0, and then choose m independent hash functions, $h_1, h_2, \ldots, h_m$, each with range 1 to s. For each element $a \in A$, the bits at positions $h_1(a), h_2(a), \ldots, h_m(a)$ in v are set to 1. Given a membership query for c, the bits at positions $h_1(c), h_2(c), \ldots, h_m(c)$ are checked. If any of them is 0, then certainly $c \notin A$. Otherwise, we conjecture that c is in the set, although there is a certain probability that we are wrong (false positive). To support updates, we maintain for each location i in the bit vector a counter of the number of times the corresponding bit is set to 1.

In [9], we have proposed an extension of a Bloom filter, called Depth Bloom Filter, that supports testing whether an XPath query matches an XML document. A *Depth Bloom Filter* (DBF) with l_b levels for an XML tree with L levels is a set of l_b Bloom filters $\{\text{DBF}_0, \text{DBF}_1, \text{DBF}_2, \ldots, \text{DBF}_{l_b-1}\}$, $l_b \leq L$, where in each DBF_i we hash all paths of the XML tree that have length i, (i.e., all paths with i + 1 nodes). A path is hashed by considering its path expression as a string. Depth Bloom Filters do not provide selectivity estimations.

Bloom Histograms [10] are designed for supporting selectivity estimations for XPath expressions. The idea is to use a histogram to summarize the frequency of all paths of an XML tree, while using a Bloom filter to summarize the corresponding values, i.e. the paths that fall into each bucket of the histogram. To construct a Bloom Histogram *BH* with b buckets for an XML tree, the set of paths in the tree is partitioned into b disjoint sets of paths, $path_i$. The *Bloom Histogram* is a two-column table $BH(BF_i, val_i)$, $1 \leq i \leq b$, where BF_i is a Bloom filter of all paths in $path_i$ and val_i is a representative value of the frequencies of all paths in $path_i$.

We enhance Depth Bloom Filters to provide selectivity estimation as follows. Instead of using a simple Bloom filter for each level i, we use a Bloom Histogram constructed by taking as input all paths of length i. The resulting *Depth Bloom Histogram* (DBH) with l_b levels and b buckets for an XML tree with L levels is a set of Bloom Histograms $\{\text{DBH}_0, \text{DBH}_1, \text{DBH}_2, \ldots, \text{DBH}_{l_b-1}\}$, $l_b \leq L$, where each DBH_i is a Bloom Histogram with b buckets that includes all paths of length i. An example is shown in Fig. 3, where Fig. 3(left) depicts a full path count table that reports the frequency of each distinct path in the tree and Fig. 3(right) the corresponding DBH.

B. Selectivity Estimation

Assume a DBH with l_b levels and b buckets and a query q of length l_q. To estimate the selectivity of q using the DBH, q is split to all possible subpaths of length 0 (single elements) up to length $l_m - 1$, where $l_m = minimum\{l_b, l_q\}$. Each subpath of length i is then hashed and checked against the b Bloom filters of the corresponding DBH, that is of DBH_i. If we have a match in more than one filter, the average of the values in the corresponding buckets is returned as the selectivity estimation for the corresponding path. After all subpaths are evaluated, a set of selectivity estimations, one for each of the subpaths, is retrieved. For the query to have a match in the data, it is required that all subpaths appear in the filter. Thus, we take as the selectivity estimation for q the minimum of the set of the selectivity values that are retrieved.

When q contains either the wildcard or the "//" operator, the query is split at its position and the resulting subqueries are evaluated separately by using the procedure described above. The selectivity estimation for q is given by the minimum estimation returned for any of the subqueries. The same applies for queries containing branching.

Complexity. Assume a DBH with l_b levels and b buckets. For a query q with length l_q, firstly, l_q subpaths of length 0 are matched against the b Bloom filters of DBH_0, then $l_q - 1$ subpaths of length 1 are matched against the b Bloom filters of DBH_1 and so on, until we match 1 path of length l_q or the length of the subpaths in question is no longer contained in the DBH ($l_q > l_b$). Thus, we have: $b * l_q + b * (l_q - 1) + b * (l_q - 2) + \cdots + b * 1 = O(b * l_q * l_m)$ lookups for each query, where $l_m = minimum\{l_b, l_q\}$.

False Positives and Estimation Error. The probability for a false positive in a Bloom filter is equal to $(1 - e^{-mn/s})^m$, where n is the number of elements inserted in the filter, s its size, and m the number of hash functions. For an XML tree $Tree$ with L levels and maximum out-degree dg, for the number of elements n_i inserted at each level i of a DBH, it holds: $n_i \leq \sum_{j=i}^{L-1} dg^j$ (i.e., is smaller than the number of non distinct paths with length i). Assume that the probability that a path in $Tree$ has frequency that belongs to bucket j ($1 \leq j \leq b$) is $P_f(j)$ with $\sum_{j=1}^{b} P_f(j) = 1$. Then, the number of elements inserted at each bucket j of each level i of the filter is at most: $P_f(j) * n_i$. Thus, the false positive probability for a single look-up in a filter corresponding to a bucket j of the i level of DBH is given by: $P_l(i,j) \leq (1 - e^{-m * P_f(j) * n_i / s})^m$. Note, that we consider that all filters are of the same size. If there is information about the frequency distribution of the paths, we can adjust the size for the filter of each bucket accordingly. The query evaluation algorithm performs b checks for each subpath of length i. Thus, the false positive probability for a subpath of length i is: $P_{sub}(i) = 1 - \prod_{j=1}^{b} (1 - P_l(i,j))$, since for not having a false positive, all the b checks must not return one.

To estimate the corresponding estimation error for any subpath lookup we rely on [10]. Let $1 < val_j \leq M$, $1 \leq j \leq b$ and let V_* denote the actual number of appearances of a path in the XML tree. The absolute error for a returned value val_j is then given by $e_j = |val_j - V_*|$. For $V_* > 0$, the estimated error for a subpath of length i that was inserted in bucket b_* of DBH_i ($1 \leq b_* \leq b$) is bounded by:

$$E[e] < \prod_{j=1, j \neq b_*}^{b} (1 - P_l(i,j)) E[|val_{b_*} - V_*|] +$$

$$(1 - \prod_{j=1, j \neq b_*}^{b} (1 - P_l(i,j))) M$$

The first term refers to the error due to the histogram when the correct bucket b_* is located, while the second term is the error for falsely reporting the existence of the path in multiple buckets. When $V_* = 0$, the error is bounded by: $E[e] < P_{sub}(i) M$.

Let us consider the simplest case where none of the subpaths in the query belongs to documents in the DBH. Then, for a false positive to occur, at each level i of the filter, we must have a false positive match against all $l_q - i + 1$ subpaths of the query, in at least one of the b buckets it is checked against. If we consider these probabilities of false matches to be independent, then this probability is: $P(i,q) = \prod_{q' \in q} P_{sub}(i)$, where q' are all subpaths of length i extracted from q. Taking into account all levels of the filter, the overall false positive probability is: $P(q) = \prod_{i=1}^{l_m} P(i,q)$, where $l_m = minimum\{l_b, l_q\}$ and l_b the number of levels of the DBH. Let us now also consider the case in which $r+1$ elements $0 \leq r \leq l_q$ exist in the document and form a correct subpath of length r. Then, at each level i of the filter, the correct subpaths that exist are at least: $r - i + 1$ and the ones that do not exist are at most $l_q - r$. To compute the false positive, $P(i,q)$ for $i \leq l_q - r$, we should consider only these subpaths.

IV. Distributed Deployment

In this section, we focus on the creation and maintenance of the clustered index. In brief, each site creates a DBH for each

Algorithm 2 EvaluateSim(H, clH)

H, clH: DBHs with l_b levels and b buckets

1: $sim = 0$
2: **for** $i = 0$ to l_b **do**
3: $mBF_i = clH.DBH_i(BF_1)$ BOR $clH.DBH_i(BF_2)$ BOR ... BOR $clH.DBH_i(BF_b)$
4: $sim_i = 0$
5: **for** $j = 0$ to b **do**
6: $sim_i = sim_i + H.DBH_i(val_j) * Jaccard(H.DBH_i(BF_j), mBF_i)$
7: **end for**
8: $sim = sim + sim_i$
9: **end for**
10: **RETURN** sim

of its documents. The DBHs of the documents of all sites are merged based on their similarity into a number of clusters to produce a single DBH for the cluster, denoted *cluster DBH* or $clDBH$, that provides a summary of all documents in the cluster. Cluster formation takes place incrementally as sites and documents are inserted, updated or deleted.

Cluster formation and maintenance is coordinated by superpeers. Superpeers correspond to sites that are stable and have increased capabilities. For simplicity, we assume a one-to-one mapping between clusters and superpeers, that is, there is exactly one superpeer per cluster. However, this mapping is virtual, in that, in practice, one superpeer may be responsible for more than one cluster, or many superpeers may collaborate for the maintenance of one cluster.

Clustered Index Construction. When a site joins the system, it constructs one DBH for each of its documents and sends them to a superpeer. The superpeer forwards these DBHs to the other superpeers. Each superpeer compares each one of them against the content of its cluster to determine their similarity. We derive the structural similarity between a new document having a DBH, H, and the content of a cluster having a cluster DBH, clH, from the similarity between the corresponding DBHs, that is, between H and clH. Similarity is computed using the $EvaluateSim$ algorithm (Alg. 2), which is based on the Jaccard distance between the corresponding simple Bloom filters of the two DBHs. Each document is assigned to the most similar superpeer (cluster). The selected superpeer merges H with clH to produce the new cluster DBH. We describe next how the merging of two DBHs is achieved.

Let the DBHs H and H' with l_b levels and b buckets be the indexes for document d and d' respectively. We want to aggregate (merge) the two DBHs to create a new DBH H'' with l_b levels and b buckets that summarizes both documents d and d'. A nice property of Bloom-based structures is that to construct H'', there is no need to have access to the actual documents d and d'. Instead, there is a simple procedure to construct H'' based on H and H'. The two DBHs H and H' are merged per level, that is level i of H is merged with level i of H' to create level i of H''. For each level i, we merge one bucket of H with one bucket of H'. We start by merging the two buckets that have the most similar frequencies, that is, the ones with the most similar value at the val column, then the ones with the next most similar frequencies and so on, until all b buckets of level i are merged. The frequency (i.e., val column) of the resulting merged bucket is set equal to the average of the frequencies (i.e., val columns) of the buckets being merged. The Bloom filter (i.e., BF column) of the merged bucket is computed by taking the bitwise OR of the Bloom filters (i.e., BF columns) of the two buckets being merged. Clearly, the same merge procedure may be used to merge DBHs that summarize more than one document.

Lastly, let us briefly discuss our simple technique for bootstrapping the system. Bootstrapping is centralized. We consider as our basic input parameter the number of clusters C. When an appropriate sample of documents D_{in} has been inserted, a single superpeer is selected to gather all DBHs and to apply *K-Means* on them. *K-Means* is very sensitive to the initial sample of documents and a different sample could lead to a completely different result. For instance, even if all documents in D_{in} belong to the same semantic category, *K-Means* may still partition them into C clusters. To deal with this problem, we enforce additional constraints. After finding the C centroids (i.e, clDBHs), we check their pairwise distances and if this is lower than a value $\epsilon > 0$, we merge the respective clusters. Any empty clusters will be filled with new documents, as they enter the system, if their distance with the existing clusters is large enough.

Clustered Index Maintenance. For supporting local updates, as in [10], each site maintains an auxiliary full path count table on which it first applies the update and then reconstructs the DBH taking it as input. Then, it sends both the original DBH H and the updated one, H', to the corresponding superpeer. The superpeer removes H from its clDBH and inserts the new H' to it through merging.

Depending on the distribution that the documents entering the system follow, some clusters may become much larger than others. The index of large clusters becomes less accurate and the superpeer responsible for it may become overloaded. In this case, the cluster needs to be partitioned. Partitioning can be handled locally by the superpeer responsible for the given cluster without disrupting the other clusters. In particular, when either the number of documents assigned to a cluster becomes too large or the accuracy of the clDBH becomes too low, a cluster split procedure is triggered. The sites that have documents in this cluster resend their DBHs to the superpeer. The superpeer computes all pair-wise similarities and chooses the two DBHs with the smallest similarity to initialize the two new clusters. Then the rest of the DBHs are merged with the cluster they are most similar to. Finally, a new superpeer is assigned to one of the two clusters. Similar steps are taken to merge two clusters, when their load becomes too low.

Overall Processing. Each query is assigned to a coordinator superpeer. All superpeers process each query in parallel by relaxing it against their part of the index until they attain k results. Then, they exchange the distance scores of their k-th result and prune their result list according to these distance

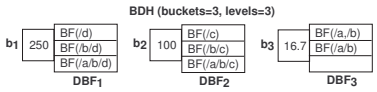

Fig. 4. BDH of the full path count table of Fig. 3

TABLE I
INPUT PARAMETERS

Parameter	Default Value	Range
# of documents	500	50 - 500
# of clusters	8	2 - 24
k	20	10%-50% of the results
# of data categories	8	-
Size of document	2KB-120KB	-
DBH size	320KB	5 - 320KB
DBH depth	3	-
DBH hash functions	4	-

scores. Finally, they send the remaining entries in the list to the coordinator, which after receiving all lists, proceeds in constructing the final top-k list by merging them.

Alternatively, we may employ a pay-as-you-go approach, in which the clusters are processed one-by-one based on their relevance to the query. This can be achieved as follows. Each query can be represented as a tree, thus we construct the DBH for the query. We rank the clusters based on the similarity between their cluster index, clDBH, and the query DBH and then visit them in descending order of similarity.

V. EXPERIMENTAL EVALUATION

We use real data sets from the Niagara Project [11] that belong to eight predefined categories as shown in Table II. There are two pairs of categories that share structural similarities, namely, the Sigmod Record data with the bibliographical data and the actors with the movies data, while there is little to none overlap among the rest. For query generation, we use the zipf distribution to select paths from the documents. Then, we replace a random element with a foo element, or insert a subpath of random length in them. The queries have a minimum length of 4 location steps. To tune the number of buckets and their boundaries, we apply the techniques presented in [10]. We use 4 buckets per level. We configure the Depth Bloom filters parameters (i.e, the BF size and the number of hash functions) based on [9]. We limit the levels to 3 since experimental results in [9] show that 3 levels are adequate for a false positive ratio below 5%.

We use four approaches for constructing the distributed index. The first two approaches use a simple path count table (PCT) as the building block of the index, while the latter two use the Depth Bloom histogram (DBH). Using the two index structures, we construct both a random index, where the data is assigned to superpeers uniformly ($randPCT$ and $randDBH$) and a clustered index ($clustPCT$ and $clustDBH$). Table I summarizes our parameters.

A. Summaries vs Full Path Indexes

In the first set of experiments, we evaluate the performance of our summaries. In addition to the DBH, we also consider an alternative way of combining histograms and Depth Bloom Filters (DBFs). Instead of using a Bloom Histogram for each of the levels of a DBF, we use a DBF for each bucket of the Bloom Histogram. In this case, we first assign all paths of the input XML document into buckets according to their frequencies. Then, we build a DBF for each bucket by splitting the paths in the bucket based on their length. Assume that the paths of the XML tree are split into b sets of paths, $path_i$, $1 \leq i \leq b$. The *Bloom Depth Histogram* (BDH) with l_b levels and b buckets is a two-column table $BDH(DBF_i, val_i)$ where each DBF_i is a DBF with l_b levels of the paths with frequency val_i. An example is shown in Fig. 4.

SELECTIVITY ESTIMATION. We compare the false positive ratio and the estimation error of the DBH and BDH indexes with those of a Bloom Histogram (BH) index of the same size. The space that each index occupies is expressed as a percentage of the space occupied by a full path count table (PCT) for the same documents (this is between 2% to 10% of the PCT size). All documents summarized by each of the indexes belong to one of our categories, i.e. they are expected to be assigned to a single cluster. This is an indication of the expected performance of a clDBH index. Figure 5(center-left) shows that both DBH and BDH improve the error of the BH up to 30% for the same space overhead.

We repeat the same experiment using documents selected uniformly at random from all categories in the system. For comparison, we use the same sizes we used for the clDBH. However, note that these do not represent the same percentage of the path count table that would be constructed in this case, since this path count table is larger due to the variety in the path expressions. As Fig. 5 shows, the comparative performance of the three indexes is similar with that in our first experiment. However, all three indexes perform worse with random documents than with similar ones. For instance, the simple Bloom Histogram does not achieve a false positive ratio lower than 40% (Fig. 5(center-right)), while for the clustered index, it achieves a ratio below 20%.

The two enhanced indexes (DBH and BDH) perform similarly for both cases. In general, their performance depends on the data characteristics. The first structure splits paths primarily according to their length, while the second one according to their frequencies. Therefore, for data with paths with equal frequencies, BDH is expected to perform worse than DBH. For the data sets in our evaluation, their performance is comparable, and we use the DBH as the index structure in the rest of our experiments.

CONSTRUCTION COST. We measure the construction cost of the clustered index, as the total size of messages required, while varying the number of clusters (Fig. 6(left)) and the number of documents (Fig. 6(center)). The deployment of the DBH reduces the construction cost of the clustered index from GBs to only a few MBs from the case when PCT is used (Fig.

TABLE II
DATA SETS

Set	Sigmod Record '03	bibliographical data	movies	actors	linux docs	company personnel	nasa	club members
Doc Size	100K	5K	2.2K	2-10K	20-120K	2K	2-14K	8K

6(left)) without damaging performance significantly, since the pruning degree is only slightly decreased (Fig. 7(left)).

SCALING. To evaluate the scaling capability of our index, we keep a fixed size for the DBH and increase the number of documents maintained. As Fig. 6(right) illustrates, our index structures scale gracefully, since the pruning degree does not decrease significantly.

Summary of Results.

- Both the Depth Bloom Histograms and the Bloom Depth Histograms outperform a same size Bloom Histogram up to 30%.
- All index structures produce better estimations in the case of clustering, i.e., when similar documents are assigned to them.
- The Depth Bloom Histograms reduce the construction cost of the clustered index from GBs to MBs.
- The Depth Bloom Histograms scale well with respect to the number of documents.

B. Benefits of a Clustered Index

PRUNING DEGREE. We compare the performance of a clustered and a non-clustered (random) index using dynamic programming (DP) relaxation. This clearly depends on the number of clusters. Thus, we measure the average pruning degree varying the number of clusters from 2 to 24. Figure 7(left) shows that for all numbers of clusters, the pruning degree is improved in the case of clustering as expected. Specifically, as we assign multiple categories to the same cluster (2 and 4 clusters), the pruning degree increases for the clustered index, while it remains almost constant for the random one. This is because if documents from one category are distributed among different clusters, relaxation at each superpeer produces results with similar quality. Using as many clusters as the document categories or less, alleviates this problem.

Increasing the number of clusters, reduces the load at each superpeer, but increases the communication cost among them. To demonstrate this trade-off, we measure the maximum processing cost among all superpeers (Fig. 7(center)) and the average communication cost during query processing (Fig. 7(right)). As processing cost, we measure the lookups performed in the index during relaxation and the number of entries that are processed for the construction of the final result list. While for a small number of clusters the first cost is the prominent one, i.e., 90% of the total cost, as the number increases the second cost reaches up to 56% of the total cost.

GREEDY AND RANDOM WALKS RELAXATION. We repeat the same set of experiments using greedy (GR) and random walks (RW) relaxation. For random walks, we set $M = 3$, so that 3 different queries are relaxed. Figure 8(left) shows that the pruning degree is again larger with clustering for both approaches. Compared to the performance of the DP approach, pruning degree is decreased by 20% for the greedy one, and by 6% for random walks. Due to this decrease in the pruning degree, the communication cost increases correspondingly for both approaches (Fig. 8(center-right)). However, their advantage over the DP approach is that the processing cost required is reduced by around 24% for the GR and 15% for the RW approach (Fig. 8(center-left)).

This gain in the processing cost results in a loss in the accuracy. We evaluate this loss by measuring the recall of the greedy and the random walks algorithms against the DP one (i.e., the percentage of results returned by each method that belong to the actual top-k results). For the random walks approach we used two different configurations with $M = 3$ and $M = 4$. As Fig. 8(right) shows, the RW approach exhibits the best recall when $M = 4$ almost 99%, while the greedy approach has the lowest recall of all (46% in the worst case).

MULTIPLE ROUNDS IN THE ELIMINATION PHASE. We want to evaluate the benefits of using multiple rounds in the elimination phase. We measure the pruning degree at each round using all three relaxation algorithms (Fig. 9(right)). The largest pruning degree is achieved on the first round, while the other rounds prune only a small number of results. If we consider the time/communication cost trade-off, it is clear that we do not need to use more than two rounds as the response time of the evaluation would only increase significantly without bringing a considerable gain in the communication and processing cost.

PAY-AS-YOU-GO EVALUATION. In this experiment, we evaluate the performance of a *pay-as-you-go* strategy in which clusters are visited gradually, starting from the "best" one for each query. We measure the recall of this strategy against a strategy in which all clusters are visited (i.e. the percentage of results returned by the pay-as-you-go strategy that belong to the actual top-k results), while gradually increasing the number of clusters visited. We follow two approaches to determine the order in which the clusters are considered. In the first approach, the clusters are considered in ascending order of the average distance of their results to the query. This can be achieved for instance by caching information related to which superpeer provided the best results for previous queries. Figure 9(left) shows that pay-as-you go works well with clustering. The clustered index achieves much higher recall than the random one in which recall increases proportionally to the clusters that processed the query. In the second approach, we rely on the similarity between the DBH of the query and the DBH of the cluster index (clDBH) to determine the order according to which the clusters are considered. As shown in

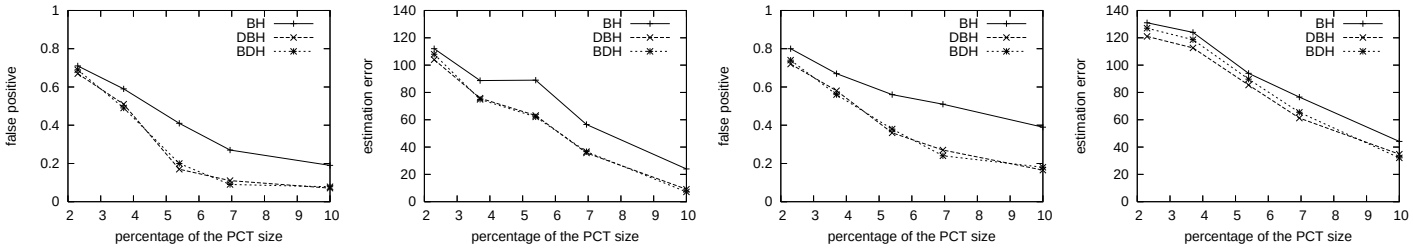

Fig. 5. False positive ratio and estimation error for (left) and (center-left) clustered documents, and (center-right) and (right) randomly selected ones

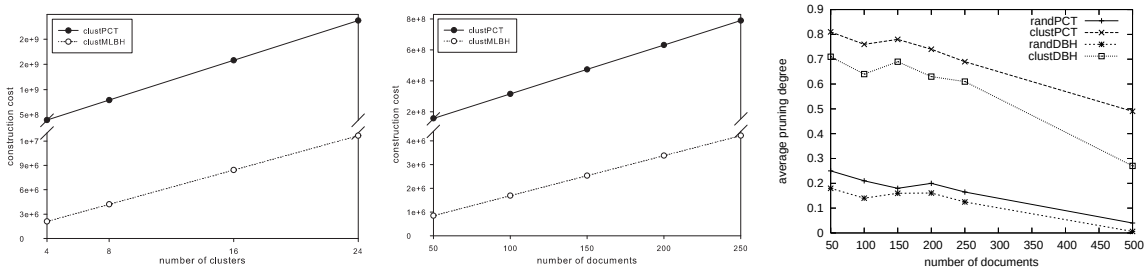

Fig. 6. Construction cost with respect to (left) varying clusters, (center) number of documents and (right) scaling

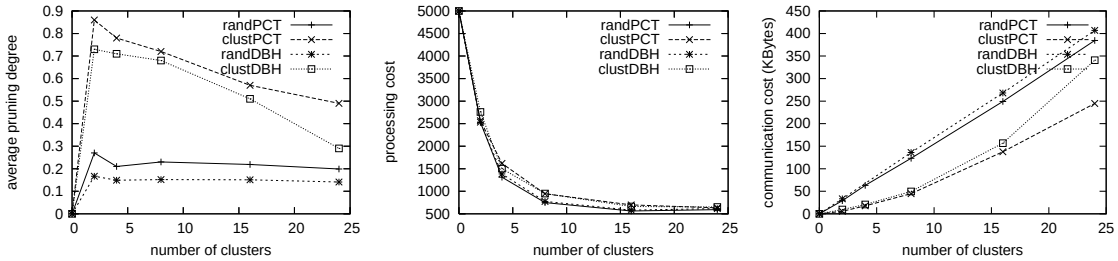

Fig. 7. (left) Pruning degree, (center) processing and (right) communication cost when using dynamic programming

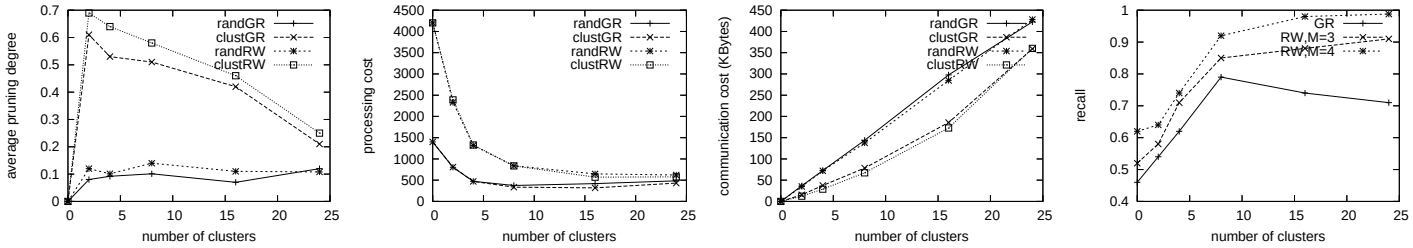

Fig. 8. (left) Pruning degree, (center-left) processing cost, (center-right) communication cost with the greedy and random walks approach and (right) comparison of the three methods

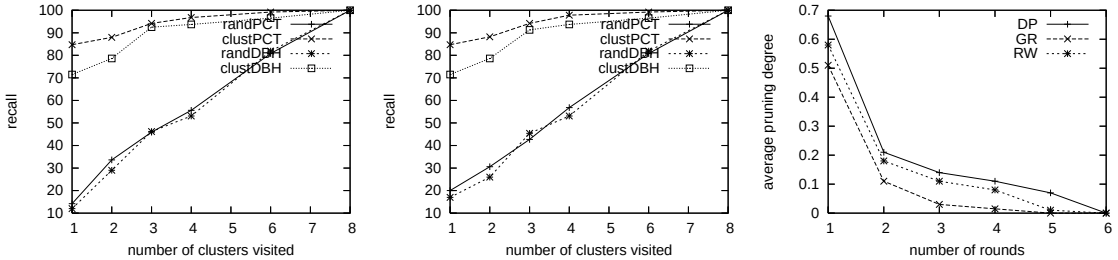

Fig. 9. Pay as you go based on (left) average distance of results and (center) query similarity, and (right) pruning degree with multiple rounds

Fig. 9(center), the ordering based on the similarity of the query to the clDBHs almost reaches the performance of the first approach. Thus, this simple similarity measure can be used as an efficient criterion for cluster selection enabling us to avoid having to process a query against all clusters.

Summary of Results.

- Using a clustered index improves the pruning degree up to 250% compared to a random index.
- The largest pruning degree is achieved by the dynamic programming approach that also requires the lowest communication cost. The greedy and the random walks relaxation approaches result in worse performance in terms of pruning degree and communication cost, but both reduce the maximum processing cost (around 24% the greedy and 15% the random walks). Further, despite not considering the whole search space, both approaches are still effective, achieving recalls up to 79% the greedy and 98% the random walks with $M = 4$.
- The clustered index can be effectively used as the base for a pay-as-you-go query evaluation, achieving a high recall. Query similarity to the clustered index can be used effectively for determining the order of visiting the clusters in a pay-as-you-go evaluation.

C. Other Issues

INFLUENCE OF *k*. It is clear that the value of k (i.e., the requested number of results) affects performance. The determinant factor is how similar the k-th result of the relaxed query is to the results of the original query. If sufficient results with a small distance from the original query do not exist, relaxation leads to queries with a very large distance from the original one, which match data at various clusters. To demonstrate this, for a distance x and a query q, we express k as a fraction of the available results with distance from q lower or equal to x. We fix *x* and measure the average pruning degree as the number of available results with at most this distance from the queries vary. For larger values of k the pruning degree decreases as low quality results, which are located at various clusters, are retrieved (Fig. 10(left)).

DYNAMIC BEHAVIOR. The communication cost for propagating a local update to the clustered index depends solely on the size of the DBH. Thus, we focus on how well the DBH's estimation accuracy is maintained after a series of updates. To this end, we consider two approaches. In the first, we apply updates on our DBH using the update propagation procedure, while in the second one, we reconstruct a new DBH with the updated data (let H and H' be the resulting DBHs, respectively). Next, we perform a batch of queries and measure the recall of H compared to that of H', i.e. we measure the percentage of results provided by H that are also provided by H'. Reconstruction may be required, when recall decreases. Our results (Fig. 10(center-left)) show that updates do not significantly affect performance and thus, reconstruction may not necessarily be frequent. Note that 20 updates correspond roughly to $1/3$ of the cluster documents being updated.

CLUSTER SPLIT. The communication cost for a cluster split depends on the number of documents that are indexed by the cluster before the split and the size of the DBH. We focus on the performance of the split process. In particular, we measure the false positive ratio and the estimation error in a clDBH as new documents are inserted into it, until a split is triggered (in this experiment after 50 insertions) and then after applying the split we show the average estimation error and false positive ratio for the two new clusters that were produced as new documents are still inserted and accordingly placed into one of the two clusters. From Fig. 10(center-right) and Fig. 10(right), we see that the split process can deal with the overloading of a single cluster and balance the document load between the two new clusters as the system continues to evolve.

Summary of Results.

- The pruning degree depends on the similarity of the available results to the original query. For large values of k, when sufficient similar results are not available, it decreases.
- Even after $1/3$ of the documents have been updated, the update procedure maintains recall up to 90%. Cluster split deals effectively with the problem of cluster overloading.

VI. RELATED WORK

Approximate XPath processing has been addressed in centralized scenarios where data is located at a single server and their schema is known. [3] uses a relaxation technique [1], [2] based on edge generalization, leaf deletion and subtree promotion. Our approach is based on the same principles, but we exploit a clustered index to appropriately order the transformations so that efficiency is improved. For ranking in [3], the tf*idf measure is extended to account for predicates both on content and structure. Each query node is assigned to a server maintaining a priority queue of partial matches. For each match at the head of its queue, a server computes a set of extended matches with their scores and updates the set of top-k matches maintained by the system. TopX [12] is a query engine that can support approximate top-k results using probabilistic score estimations, but unlike our approach relies on relaxing values rather than structure and uses the latter only to reduce the processing cost. Compact structural summaries for XML data such as [4], [13] that provide selectivity estimations for twig queries can be used to support approximate query processing. However, though both structures produce lower estimation errors than Depth Bloom histograms, they require larger space overhead and therefore would incur larger communication costs.

Distributed evaluation of top-k queries has been addressed in the context of distributed web search engines, such as Minerva [14] that is layered upon a distributed hash table, which holds compact, aggregated information about the peers local indexes. Every peer is responsible for a randomized subset of the global directory, unlike our approach where the index is clustered. Queries are forwarded to a chosen, small set of peers. KLEE [5] extends [6], and allows a peer to trade-off result quality and expected performance. The system assumes that index lists for text terms are distributed across peers. Query evaluation exploits statistical information about

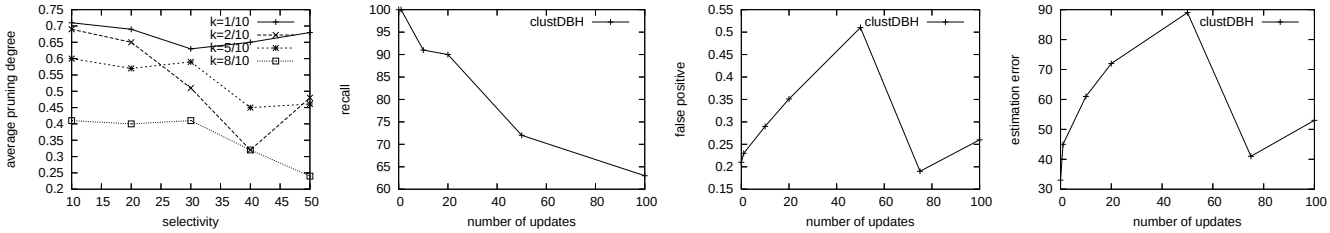

Fig. 10. (left) Influence of k, (center-left) Update propagation, (center-right) cluster split influence on false positives and (right) estimation error

the indexed data. Both approaches assume keyword-based data models. Other Bloom-based structures have been proposed recently for distributed XML search [15]. These structures were designed for cost-efficient structural joins and unlike our enhanced filters, do not provide selectivity estimations or support approximate query processing.

A preliminary version of this paper appears in [16] as poster.

VII. Discussion and Extensions

The main claim of this paper is that clustering improves the performance of relaxation over distributed data collections. This is achieved either by reducing the number of results exchanged in threshold-based distributed top-k processing or/and by considering only data in selected clusters. This is a general argument, which is showcased in this paper using structural relaxation on distributed collections of XML documents.

Structure vs Content. In general, approximate query processing and ranking for XML may include other characteristics of the documents such as values or keywords. This is an important and well studied problem where various techniques including ontologies and thesaurus are used for relaxation and tf*idf measures or other IR techniques for ranking (for example, [3], [12], [17]). One way to combine structure and other characteristics is to consider them as complementary and treat them independently. In this case, we can maintain our structural clustering and use auxiliary indexes for values. For ranking, an aggregated distance measure based on both structure and value can be used. Structural clustering would still improve pruning, but perhaps in a lesser degree than with pure structural relaxation. Another approach would be to construct the clustered index based on both values and structure using appropriate index structures and clustering methods. Deriving a compact, summary data structure similar to the Depth Bloom Histogram for this case is an interesting problem for future research. This is a challenging task, since such an index should be compact, merge-able, update-able and provide good selectivity estimations with no false negatives. Clearly, which of the two approach works best depends on the data distribution. If many query results comply to a specific DTD or XML schema, then structural clustering may be appropriate. On the other hand, if documents relative to a specific query comply to numerous different schemes, then a combined approach might be more appropriate.

Hierarchical Clustering. Our clustered index corresponds to a partition of the input documents, so that each superpeer (cluster) is responsible for a disjoint part of the data space. An interesting extension is to build a hierarchy of superpeers. One way to achieve this is by clustering similar superpeers, which effectively corresponds to a hierarchical agglomerative clustering procedure. This will make locating the most similar cluster faster by using a top-down procedure starting from the most general cluster and proceeding downwards to the appropriate sub-cluster. However, this would make maintenance harder, since it would involve updates at multiple levels. Another drawback would be the loss of accuracy at the higher layers. A related issue is the overlay topology among the superpeers. In our experiments, we have assumed a fully-connected superpeer overlay, where each superpeer is connected to all other superpeers. This is an issue orthogonal to our approach in the sense that it does not affect pruning during relaxation. However, the choice of an appropriate overlay would improve the communication cost among the superpeers.

VIII. Summary

We have addressed the problem of the efficient evaluation of XPath approximate queries over dynamic distributed collections of XML data. We have focused on reducing the communication cost required for evaluating top-k queries in such settings, by introducing a distributed clustered path index which enables the sites responsible for query evaluation to prune the number of candidate results they need to consider.

REFERENCES

[1] S. Amer-Yahia, L. Lakshmanan, and S. Pandit, "Flexpath: Flexible structure and full-text querying for xml," in *SIGMOD*, 2004.

[2] S. Amer-Yahia, S. Cho, and D. Srivastava, "Tree Pattern Relaxation," in *EDBT*, 2002.

[3] A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava, "Adaptive Processing of Top-k Queries in XML," in *ICDE*, 2005.

[4] N. Polyzotis, M. Garofalakis, and Y. Ioannidis, "Approximate XML Query Answers," in *SIGMOD*, 2004.

[5] S. Michel, P. Triantafillou, and G. Weikum, "KLEE: A Framework for Distributed Top-k Query Algorithms," in *VLDB*, 2005.

[6] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," in *PODS*, 2001.

[7] G. Koloniari and E. Pitoura, "Distributed Approximate XPath Processing," DCS 2007-08, Univ. of Ioannina, 2007.

[8] B. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *CACM*, vol. 13, no. 7, 1970.

[9] G. Koloniari and E. Pitoura, "Content-based Routing of Path Queries in Peer-to-Peer Systems," in *EDBT*, 2004.

[10] W. Wang, H. Jiang, H. Lu, and J. Yu, "Bloom Histogram: Path Selectivity Estimation for XML Data with Updates," in *VLDB*, 2004.

[11] "Niagara Project. http://www.cs.wisc.edu/niagara/data."

[12] M. Theobald, R. Schenkel, and G. Weikum, "An Efficient Versatile Query Engine for TopX Search," in *VLDB*, 2005.

[13] A. Aboulnaga, A. Alameldeen, and J. Naughton, "Estimating the Selectivity of XML Path Expressions for Internet Scale Applications," in *VLDB*, 2001.

[14] M. Bender, S. Michel, G. Weikum, and C. Zimmer, "The MINERVA project: Database selection in the context of P2P search," in *BTW*, 2005.

[15] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun, "XML processing in DHT networks," in *ICDE*, 2008.

[16] G. Koloniari and E. Pitoura, "A Clustered Index Approach to Distributed XPath Processing," Poster in ICDE, 2008.

[17] N. Fuhr and K. Grossjohann, "XIRQL: An Extension of XQL for Information Retrieval," in *SIGIR*, 2001.