Distributed Approximate XPath Processing

Georgia Koloniari and Evaggelia Pitoura
Computer Science Department, University of Ioannina
Ioannina, Greece
{kgeorgia, pitoura}@cs.uoi.gr

Abstract

Supporting top-k queries over distributed collections of schemaless XML data poses two challenges. While XML supports expressive query languages such as XPath and XQuery, these languages require the user to know the schema of the data so as to write an appropriate query which may not be feasible in distributed systems with autonomous and dynamic sources. Thus, there is a need for supporting approximate query processing. Furthermore, retrieving the top-k most relevant results in a distributed system incurs large communication and processing cost, since partial result lists from a number of different sites need to be combined and ranked to assembly the top-k answers. To address both of these issues, we present an approach for approximate XPath processing over distributed collections of XML documents that relies on the use of a clustered path index, where documents are grouped based on structural information. Our method gradually generalizes a query by applying a set of structural transformations to it and the retrieved results are ranked with a distance measure based on the edit distance between two path expressions. A new compact indexing data structure, the Multi-Level Bloom Histogram, is utilized to reduce the index construction cost. Our experimental results show that our approach can significantly reduce the communication cost for retrieving the top-k results, while maintaining a low construction cost for the clustered index.

1 Introduction

XML [18] has evolved as the new standard for data representation and exchange in the Internet. Its flexibility in describing heterogeneous data makes it very popular for distributed applications and systems where the data are either native XML documents or XML descriptions of data or services. XML query languages (XQuery, XPath) address both the structure and content of the data, conveying rich semantics to queries. These query languages require knowledge of the data schema so as to phrase the appropriate queries. However, in systems with autonomous and dynamic sources, each user may be unaware of the schema remote data follow but still requires querying it.
For example let us consider two sources A and B that maintain conference and journal articles regarding databases. Source A maintains the articles sorted by author following the XML schema depicted in the document of Fig. 1(a) while B sorts the articles by year following the schema of the document in Fig. 1(b). Now consider a user that is interested in all articles published by a specific author ‘X’ in year ‘Y’. To retrieve the corresponding articles from source A, the user needs to submit a query such as $q_1=’/\text{author}=’X’/\text{year}=’Y’/\text{publication’}$, while from B the query needs to be rewritten as $q_2=’/\text{year}=’Y’/\text{author}=’X’/\text{article’}$. Thus, there is a need for supporting approximate processing of XPath queries, so that users can still retrieve a satisfying number of results without needing to know the exact schema of the data. Let us consider a user interested in $k$ articles of the specific author. If the exact matches to a query in the distributed documents collection is smaller than $k$, the system should provide approximate answers that though not exact matches to the query, they are still relevant. Thus, assuming the user submits query $q_1$ in our example and does not retrieve $k$ results from source A, the system should be able to provide more relevant answers from source B by identifying the relevance of the query with the source’s data. The query thus needs to be generalized or “relaxed”, and the results to the new generalized query can be retrieved for the user. The use of a similarity measure is usually required to evaluate the quality of the approximate results, i.e. their relevance to the original query. The main problem in this scenario is how much do we need to generalize the original query so as to attain $k$ results. If the query is not generalized enough we may end up again with less than $k$ results, while over-generalizing a query will provide us with more than $k$ results but of very low quality.

We address this problem with the use of an index that provides enough information so that the system can determine when the query has been generalized enough to satisfy the user request for $k$ results. By using an approach that gradually generalizes a query, the index is looked up for each new generalized query and an estimation of the number of results of the given query is retrieved. Thus, when this number is $k$ or larger the generalization stops. In a centralized setting where all data is indexed in a single server this approach is applied easily.

However, if the data is distributed among multiple nodes then finding the $k$ most relevant results becomes more complicated. We assume that each node maintains its own local index. A straightforward approach requires sending and evaluating the query locally at each node in the distributed system. Each node produces its own local list of the top-$k$ relevant results by following a procedure similar to the centralized scenario. To evaluate the final top-$k$ answers from all nodes, all local lists are forwarded to one node which is responsible for merging and sorting the partial list to acquire the final result list. Therefore, this approach requires a large communication cost among the nodes and also a large processing cost for the node responsible for constructing the final result list. The problem becomes even more difficult when all nodes are not able to locally generalize the query, due to lack of processing or power capabilities. In this case, each possible generalized query should be produced at one node able to handle
the relaxation process and subsequently sent to each other node for evaluation against its local index, resulting in further increase in the communication cost.

To improve performance a method to decrease the communication cost and the processing effort of the site responsible for creating the final result is required. To this end, we present an approach for supporting approximate XPath processing over large and dynamic distributed collections of schemaless XML data. For supporting approximate query results over XML data we present an algorithm that gradually relaxes a query. The algorithm applies a number of structural transformations to a query by choosing at each step the transformation that results in the lowest loss of the quality of results. To measure the quality of results we use a distance metric based on the edit distance between two path expressions.

Furthermore, instead of relying on the nodes local indexes, we propose the use of a distributed clustered path index over the distributed data collection. We consider a system where a small number of nodes that have good stability properties and processing capabilities are assigned the role of index servers, i.e. they are responsible for the maintenance of the clustered index and query processing. By restricting the relaxation task to the index server sites we first, alleviate the load of other nodes with limited processing capabilities and second, we reduce the communication cost required for the construction of the top-k results, since the communication required to assembly the list is limited to a small number of index servers and not all the nodes in the system. In addition, the use of more than one server avoids overloading a single server with both the index maintenance and query processing tasks as in a centralized scenario.

The clustered path index groups data based on structural information, so that each cluster includes documents with similar structural properties. The index is partitioned and each cluster (or set of clusters) is assigned to a different index server. The relaxation algorithm relies on the clustered index so as to transform a query suitably for each of the clusters in the index. Therefore, each server runs its own instance of the relaxation algorithm so as to maximize the retrieved results and stop the relaxation process as soon as possible.

To further improve the performance of our system, we rely on an idea presented in [5]. The index servers exchange the distance score of their k-th results so as to determine whether they can prune from their local result lists answers that will not be part of the final top-k results. Thus, the lists that are sent to the site responsible for sorting the final results are much smaller and require less processing effort from the site. We argue that the use of a clustered index is able to decrease the size of the lists that need to be sent to the final site much further than a randomly partitioned distributed index would, thus compensating for its higher construction and maintenance cost. In addition, we introduce the use of a new indexing structure the Multi-level Bloom Histogram, that is used to minimize the construction cost of the clustered index while supporting efficiently the processing of the approximate XPath queries.

To sum up, the contributions of our paper are:

- a relaxation algorithm for approximate XPath processing and a suitable
distance measure for evaluating the quality of the results that are both suitable for use upon distributed collections of schemaless XML data since they do not require any knowledge of the data scheme.

- a distributed approach for processing approximate top-k queries over a clustered index that is able to significantly reduce the communication costs and the processing required for retrieving the top-k results and

- a distributed low cost construction procedure for the clustered index that relies on the use of a new indexing structure the Multi-Level Bloom Histogram.

The rest of the paper is organized as follows: in Section 2, we summarize related work. In Section 3, we outline the data and query language model of the system and present our technique for relaxing an XPath query along with our distance measure. In section 4, we describe the new indexing structure. In Section 5, we present the distributed procedure for constructing the clustered index along with the query routing and results retrieval processes. Section 6, includes our experimental results. We conclude in Section 7, with a summary and directions for future work.

## 2 Related Work

The problem of approximate query processing over distributed collections of XML data has not been addressed adequately. Most efforts deal with the evaluation of top-k queries over relational or text-based distributed data in an effort to address the impacts the distribution of the data incurs but do not focus on the specific properties of XML data such as their inherent structure. On the other hand, approximate processing of XPath queries has only been addressed in centralized scenarios where the data is concentrated in a single server and their schema is known. In this paper, we adapt results from both research directions in order to tackle the overall problem of dealing with distributed XML data collections by providing both an efficient relaxing algorithm for XPath processing.
and an efficient distributed procedure for retrieving the top-k results with a low communication cost, based on the use of a clustered path index.

**Distributed Top-k Query Evaluation.** Recently much attention has been attracted to the distributed evaluation of top-k queries in the context of distributed web search engines, such as Minerva [3] and CHORA [6] that are layered upon a dynamic distributed directory, such as a distributed hash table. In Minerva, the DHT 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 actually clustered so that similar information is assigned to the same superpeer. When a query cannot be satisfied locally, it is forwarded to a chosen, small set of remote peers. A new routing algorithm (IQN) [11] exploits statistical synopses published in the DHT overlay to iteratively choose the best peers for forwarding a query. CHORA’s basic characteristic is that it exploits a peer’s web viewing history to recommend useful web sites to queriers. The queries supported by both, are keyword-based and so is the model used for indexing and ranking.

KLEE [12] is an algorithmic framework for distributed top-k queries evaluation that allows a peer to trade-off result quality and expected performance. The system assumes that index lists for text terms or data attributes are distributed across peers. For query evaluation statistical information about the indexed data is exploited so as to improve performance. Again both the auxiliary structures maintaining statistical information and the scoring functions are tailored for a flat data model and are not suitable for XML data.

**Approximate Match Queries over XML Data.** Most work on supporting approximate match queries over XML data is limited in a centralized scenario. In [10], a relaxation technique [2] based on edge generalization, leaf deletion and subtree promotion is used. The tf*idf measure is extended so as to account for predicates both on data content and structure. Although our relaxation technique is also based on the same principles, the order upon which we apply the relaxation steps exploits information about the peers data maintained at the clustered index so as to attain results more efficiently. Different execution plans are deployed for different partial matches. Each node in a query pattern is assigned to a server. The server responsible for the root of the query initializes all partial matches. Each of the other servers, maintains a priority queue of partial matches. For each match at the head of its queue, a server computes a set of extended matches along with their scores. A set of top-k (partial or complete) matches with their scores is maintained. A newly computed partial match either updates the score of an existing one in the set, or replaces an existing match, or it is discarded. The partial matches generated, are routed to the next server that needs to process them.

TREESKETCHes [15] are compact structural synopses of XML data that enable low-error approximate answers for twig queries. Results are ranked by a distance metric that quantifies the differences between two XML trees. While, TREESKETCHes efficiently evaluate approximate answers, they are designed for a centralized setting. In our approach they can be used locally at each node to summarize its content, but it is not an easy task to determine how they can
be merged to construct the clustered index. Furthermore, the authors do not deal with the update of this structure which is crucial if the structure is to be used in a dynamic system.

**XML Clustering.** Related research on clustering XML documents in centralized settings, which relies mostly on structural information, cannot be directly applied when the documents are distributed among multiple dynamic nodes. For the classification of schema-less data, the authors of [16] combine text terms, structural information in the form of twigs and paths and also ontological knowledge to construct more expressive feature spaces that are then used for the classification. S-GRACE [9] is a hierarchical algorithm for clustering XML documents with a distance metric based on the notion of a structure graph, which is a minimal summary of edge containment in the data. In contrast with the previous methods that can be applied to schema-less data, XClust [8] addresses clustering when schema information in the form of DTDs is available. XClust clusters DTDs based on the semantics, immediate descendants and leaf-context similarity of DTD elements. XRules [19] assigns the documents to categories through a rule-based classification approach that relates the presence of a particular structural pattern in an XML document to its likelihood of belonging to a particular category.

## 3 Approximate Match Queries

### 3.1 Data Model and Query Language

XML employs a tree-structured model for representing data. In particular, we can model an XML document as a node-labeled tree \( T(V, E) \) (Fig. 1(c)). Each node \( e \in V \) corresponds to an XML element with a label assigned from some string literals alphabet that captures element semantics. Edges \((e_i, e_j) \in E\) are used to capture the containment of element \( e_i \) under \( e_j \). Any subtree of \( T \) is called an XML fragment.

Queries in XML query languages specify patterns of selection predicates on multiple elements that have some specified relationship. In this paper, we concentrate on queries that use simple XPath expressions involving only the child and descendant-or-self axes (i.e. ‘/’ and ‘//’) and the wildcard operator (*). Although leaf elements in \( T \) typically contain values, in this paper, we ignore values and mainly focus on the label structure of an XML tree. Such expressions form the building blocks of more advanced querying. They are represented as a sequence of element tags and describe a navigation pattern through the XML tree. An XPath expression \( //t_1/t_2/\ldots/t_n \) is evaluated sequentially by finding an element \( t_1 \) anywhere in the document and nested within it an element \( t_2 \), and so on until we find \( t_n \). The result of the XPath expression is the set of fragments rooted at the \( t_n \) nodes found in the given XML tree. We say that a query \( q \) matches a document \( D \) if the result set of the evaluation of \( q \) against \( D \) is non-empty and denote as \( \text{result}(q, D) \) the fragments included in the result set.
3.2 Relaxation

Let us consider a distributed system where users issue queries expressed by simple XPath expressions and are interested in retrieving a specified number of results \( k \). If the exact matches to a query in the system are not enough to satisfy the requested number of answers, i.e. \( |\text{result}(q, D)| < k \), the system should try to find approximate answers to the query. In particular, the system tries to find fragments that are relative to the query, i.e. structurally similar, though they do not constitute exact answers to it. In this paper, we deal with the problem of locating these results efficiently. More formally:

**Problem Formulation:** Given a path query \( q \) and a set of documents \( D \), find \( k \) XPath fragments \( f_1, f_2, \ldots, f_k \in D \), such that given a distance measure \( d \) between two XPath expressions, \( d(q, f_1) \leq d(q, f_2) \leq \ldots \leq d(q, f_k) \) and \((\neg\exists)f' \in D \) with \( d(q, f') < d(q, f_k) \).

The distance measure \( d \) is used to rank the approximate results according to their quality, i.e. according to their distance from the initial query \( q \). The greater the distance of a result fragment from the initial query, the lower its quality. To this end, we use a simple distance measure between two linear path expressions based on the edit distance between string sequences. The distance measure evaluates the number of mismatching element tags between two simple path expressions. To compare two path expressions with different lengths, we define a function \( \text{ap}(p, l') \), which given a path expression \( p \) of length \( l \), and a number \( l' \), appends \((l' - l)\) path steps in \( p \) with the wildcard as their element tag. That is: \( \text{ap}^*(p, l') = /p_1/p_2/\ldots/p_l/p_{l+1}/\ldots/p_{l'} \), where \( p_i =^{\ast} \), for \( l < i \leq l' \).

We consider that the wildcard operator does not match any specific element tag in the computation of the distance.

**Definition 1 (Distance Measure)** The distance \( d \) between two path expressions \( q \) and \( q' \) with length \( l \) and \( l' \) respectively, is defined as:

\[
d(q, q') = \begin{cases} 
  d(\text{ap}^*(q, l'), q') = \sum_{i=1}^{l'} \text{diff}(q'_i, q_i)/l', & \text{if } l \leq l' \\
  d(q, \text{ap}^*(q', l)) = \sum_{i=1}^{l} \text{diff}(q'_i, q_i)/l, & \text{if } l > l'
\end{cases}
\]

where \( q_i \) and \( q'_i \) denote the element in the position \( i \) in \( q \) and \( q' \) respectively, and \( \text{diff}(q_i, q'_i) = 0 \) if \( q_i = q'_i \) and 1 otherwise.

To acquire the approximate results to a query, we rely on generalizing the original query, by applying a number of structural transformations to it. Our transformations are based on the ones presented in [2], but are simplified for linear path expressions. We use edge generalization as defined in [2]. Leaf deletion is in the case of linear path expressions equivalent to the truncation of the last element of a path expression. Finally, the subtree promotion can be accomplished by the replacement of an element with a wildcard operator, though it is not directly applicable in linear path expressions. However, though the set of transformations is applicable in our case the measure used for evaluating the quality of results in [2] is a variation of the tf\*idf measure typically used in IR that requires global knowledge of the data distribution so as to be evaluated
and therefore it is not suitable for our purposes. Instead of relying on that type of measure we propose using the edit distance between two path expressions for ranking our results, that can be easily computed from the original query and the result fragment.

**Definition 2 (Structural Transformations)** We define a set of mapping functions $T : P \rightarrow P'$, where $P$ is the set of simple path expressions, that map a simple path expression $p = \mathcal{P}_1/p_2/.../p_n$ into a new path expression $T(p)$ by applying a structural transformation on $p$. In particular, we define the following three functions:

- **truncation of the last element of $p$:**
  
  \[ \text{Trunc}(p) = \mathcal{P}_1/p_2/.../p_{n-1} \]

- **replacement of an element tag at position $i$ of $p$ with the wildcard operator:**
  
  \[ \text{RepTag}(p, i) = \mathcal{P}_1/p_2/.../p_{i-1}/*p_{i+1}/.../p_n. \]

- **replacement of a ‘/’ axis in position $i$ with the ‘//’ axis:** \[ \text{RepAxis}(p, i) = \mathcal{P}_1/p_2/.../p_{i-1}///p_i/.../p_n. \]

We associate a cost for each structural transformation, which is evaluated by calculating the distance between the original path expression $p$ and the resulting relaxed path expression $p'$, which is attained after applying the transformation on $p$.

**Property 1 (Transformation Cost)** The cost for the structural transformations is defined as:

- \[ \text{cost}_{\text{Trunc}} := d(\text{Trunc}(p), p) = 1 + 0 = 1. \]

- \[ \text{cost}_{\text{RepTag}} := d(\text{RepTag}(p, i), p) = 0 + 1 = 1. \]

- \[ \text{cost}_{\text{RepAxis}} := d(\text{RepAxis}(p, i), p) = d(\mathcal{P}_1/p_2/.../p_n, \mathcal{P}_1/p_2/.../p_{i-1}/*.../*/p_i/.../p_n) = x. \]

The cost of the last transformation, is evaluated if we consider that the ‘//’ is equivalent to inserting a specific number of simple steps in the path expression with the wildcard operator. The number of the inserted steps $x$ is equal to the number of mismatching labels and therefore the distance between the initial and the resulting path expression.

Let us note the difference between the distance between two path expressions and the cost of a transformation. The distance measure is symmetric while the cost though evaluated also by calculating the distance between two path expressions is not defined for any pair of path expressions. For example, it holds that $d(a/ * /b, a/c/b) = d(a/c/b, a/ * /b) = 1$, but we can define the cost for transforming ‘$a/c/b$’ to ‘$a/ * /b$’ which is 1 since the transformation RepTag can perform this transformation, while the transition from ‘$a/ * /b$’ to ‘$a/c/b$’ cannot be obtained through any of the transformations we defined.
We denote as $T^t(p)$ the path expression that is generated after applying a sequence of $t$ transformations on $p$. That is, $T^t(p)$, denotes the composition of the $t$ transformation functions in the sequence, i.e., $T^t(p) = T^t(T^{t-1}(\ldots(T^1(p))\ldots))$.

It is straightforward to notice that:

$$d(q, T^t(q)) = d(q, T^{t-1}(q)) + \text{cost}(T^t),$$

where $T^t$ denotes the $t$-th transformation and depends only on the type of the transformation and not on the query it is applied upon.

After defining the cost for a sequence of transformations it becomes clear that the distance between the original query $q$ and the query $T^t(q)$ produced after a sequence of transformations is not necessarily equal to their cost. This is because the cost is an accumulating measure that depends on the intermediate steps (i.e. the particular transformations in a sequence) while the distance measure only takes into account the first and last path expression. Thus, if redundant transformations are applied, that is, we do not reach $T^t(q)$ through taking the minimum number of required steps, the cost of the transformation sequence is larger than the distance of the original and result path expressions.

We are interested in two important properties of the set of structural transformations.

**Property 2 (Monotonicity)** The set of available transformation functions is monotonic with respect to the distance measure $d$. That is, $d(q, T^i(q)) \leq d(q, T^{i+1}(q))$.

**Proof.** This is derived from eq. 1, since the cost function of any of the available transformations is a positive integer.

The second property of the set of structural transformations guarantees that if a an XML fragment $f$ matches the original query $q$ then it also matches any query that is derived after the application of any possible combination of transformations on $q$. Consequently, any fragment that matched any query that has undergone $i$ transformations, also matches the query produced with the same set of $i$ - 1 transformations. In particular:

**Property 3 (Guarantee)** if $f \in \text{result}(T^i(q), D) \Rightarrow f \in \text{result}(T^{i-1}(q), D)$.

**Proof.** This is a straightforward result of the type of structural transformations that only result in more general queries than the original. For example, any fragment that matches a path expression $//q_1/q_2/\ldots/q_n$, also matches $\text{Trunc}(q) = //q_1/q_2/\ldots/q_{n-1}$. The same observation is obvious for the $\text{RepTag}(q, i)$ and $\text{RepAxis}(q, i)$ functions.

This property ensures that by relaxing our query we do not lose any results that may exist. Furthermore, it holds that $|\text{result}(T^i(q), D)| \geq |\text{result}(T^{i-1}(q), D)|$.

### 3.3 Index-based Relaxation

The set of structural transformations is the main tool for defining a relaxation algorithm that gradually generalizes a query so as to attain the number of
results the user requests. However, the issue that remains to be addressed is the
termination condition for the relaxation algorithm, that is, how much we need
to generalize the query to attain the required results. To solve this, we propose
deriving the termination condition by using a path index over the distributed
XML document collection.

The index we consider represents a structural summary of the XML doc-
uments in the system. Its functionality is to provide for any given path ex-
pression its frequency in the document collection, i.e. the number of match-
ing XML fragments. Thus, the PI supports one basic operation defined as
\textit{EstRes}(PI, q) \approx |\textit{result}(q, D)|. For a query \textit{q}, if \textit{EstRes}(PI, q) < \textit{k}, then the
system should try to find among the index entries approximate results for \textit{q} by
applying the relaxation algorithm.

The relaxation algorithm takes a user query as input and gradually applies
a combination of the available transformations to it so as to attain the user-
specified number of results. The order and the number of transformations that
the algorithm applies is driven by two factors, the quality of the results that the
relaxed query will attain with respect to the original query and the information
derived from the index.

Based on the distance measure \textit{d}, we introduce a greedy relaxation algorithm,
that tries to apply at each step the transformation that will result in a query \textit{q}'
with the smallest possible distance from the original query. Thus, the algorithm
tries to find a relaxed query that will result in the lowest loss of quality of
results possible. The clustered index is also consulted so as to consider only
possible transformations that actually have a match in the nodes data. Thus,
the algorithm is able to discard irrelevant relaxations that will not provide any
results for the query and continue only with promising candidates. Furthermore,
for transformed queries that result in the same loss of quality, the index is used
so as to select the one that is estimated to provide the larger number of results.
Finally, the relaxation process terminates when \textit{EstRes}(PI, q') \geq \textit{k}.

At each iteration, the Relax algorithm (Alg. 1) applies all possible one-step
transformations to each of the queries that belong to its \textit{qlist} (lines 3-12). Next,
the query with the minimum distance value (\textit{qMIN}) from all queries that were
generated in this iteration is selected (line 13). If more than one query has the
same value, then the one with the largest value of \textit{EstRes} is chosen (line 14)
and its results are accumulated with the results retrieved so far (line 15). Since
RepTag and Trunc have the same cost, the index is used to choose between the
two transformations. If the new number of results is less than the target value
\textit{k}, the algorithm proceeds to the next iteration after appending \textit{qMIN} to the
input query list (lines 16-20). If the target number is achieved the algorithm
ends.

If no index is maintained, then at each iteration of the algorithm, the node
applying the relaxation must send the query to all system nodes so it can be
evaluated against their local data. The results from each node are returned to
the server and if their number is smaller than \textit{k}, the algorithm proceeds on its
next iteration and so on. Thus, the use of the index is able to considerably
decrease the communication cost required for evaluating a query. In addition,
Algorithm 1 Relax($PI, K, qlist, NUM, k$)

$PI$: Path Index, $K$: Number of requested results
$qlist$: Input query list with pairs of the form ($expr, dist(expr, q)$), initially set to ($q, 0$)
$NUM$: iteration number of the algorithm, initially set to 1
$k$: Number of results retrieved so far, initially set to 0

1: $qMIN = NULL$ //query for the next iteration
2: $qw, dw, q1, qcan$ //Lists of the form of $qlist$ for storing the intermediate results

3: for $j = 1$ to $NUM$ do
4:   $q1.exp = Trunc(qlist[j])$
5:   $q1.dist = qlist[j].dist + 1$
6: for $i = 0$ to $n1$ do
7:   $qw[i].exp = RepTag(qlist[j], i)$
8:   $qw[i].dist = qlist[j].dist + 1$
9:   $dw[i].exp = RepAxis(qlist[j], i)$
10:  $qw[i].dist = qlist[j].dist + x$
11: end for
12: end for
13: $qcan$: Find $exprs$ with minimum $dist$ value in $qw, dw, q1$
14: $qMIN$: Find in $qcan$ the query with the largest $EstRes$
15: $k = k + EstRes(PI, qMIN)$
16: if $k < K$ then
17:   $NUM = NUM + 1$
18: end if
19: $qlist[NUM] = qMIN$
20: Go to 2
21: end if
if no index is used, the relaxation algorithm has no way of discerning between
relaxed queries with the same cost.

To improve the complexity of the relaxation algorithm we use a dynamic
programming technique. Based on eq. 1, the intermediate distances that are
calculated at each iteration are stored in auxiliary tables (qlist, qw, dw), so that
the algorithm does not have to recompute them at each iteration.

When there are transformed queries that have no results with regard to
the index but have a smaller distance with the original query than that of
the best relaxed query that matches the index, the algorithm still applies an
additional transformation step on these queries and so on, until each candidate
query in our algorithm has either a match in the clustered index or no more
possible transformations can be applied to it or its distance is larger than that
of a query with results. If all one step transformations do not produce any
results with respect to the clustered index we proceed by applying a second
step of relaxation to the already relaxed queries. The algorithm ensures that
the selected transformation is the one that will result in the smallest loss of
quality.

**Property 4 (Correctness)** The relaxation algorithm chooses at each iteration
the transformation that will result in the smallest loss of quality in the retrieved
results, and thus it leads to a global optimum.

**Proof.** Let us consider that the input query list of the algorithm consists of \( j \)
queries \( q_1, \ldots, q_i \) in the \( n \)-th iteration \( (j \leq n) \) of the algorithm. In lines 3-12
the algorithm considers every possible one-step transformation on each of the
queries and computes their corresponding distances with the original query \( q \). It
then selects (line 14-15) the transformed query with the lowest distance, i.e. it
chooses to apply the transformation that will result in the least loss of quality
from all the one-step transformations. Since every possible query is considered,
it suffices to show that there exists no two-step (or more steps) transformation
of the same query that results in a smaller loss in quality from any one-step
transformation to prove that the algorithm selects the best transformation. Since
the transformation function is monotonous, this assumption holds.

From the correctness property it is clear that our algorithm does not allow
redundant transformations to take place and always selects the minimal number
of steps to reach a relaxed form of a query. Thus, our algorithm ensures that
the cost of the transformations that resulted in a query \( q' \) is equal to its distance
from the original query \( q \).

4 The Relaxation Index

The relaxation algorithm relies on the use of a path index that supports the
EstRes operation, i.e. provides a selectivity estimation for any path expression
\( p \in D \). Several data structures for efficiently summarizing and indexing
XML data provide this type of statistics (\([1],[14],[15]\)). We describe a simple
table structure that fits our requirements to demonstrate the feasibility of our approach, although any of the other structures can be used for our purposes.

We consider firstly, a centralized path index CPI that maintains information for the entire collection of distributed XML documents.

**Definition 3 (Centralized Index CPI)** Given a distributed collection of documents D, the centralized CPI index is defined as a table T that contains an entry t for each distinct path p that appears in the documents in D. Each entry is composed of three fields. The path expression p, its total number of appearances in all the documents in D (frequency) and a set of the peer identifiers that store documents that contain p (peer_ids).

To evaluate a query using the CPI the query is forwarded to the centralized index server and evaluated using the Relax algorithm. However, using a centralized index server is not suitable for a distributed environment where the server would easily become a bottleneck deteriorating system performance. Furthermore, keeping this server updated will incur large communication costs in the system. Thus, the need for a distributed index arises. However, building and then partitioning the path table structure we described incurs a large communication cost. Furthermore, the structure itself requires a large space overhead and more importantly when evaluating each query, it requires to scan each entry of the table thus incurring also large processing costs. To this end, we introduce the use of a new indexing structure, the Multi-level Bloom Histogram, that is suitable for our purposes. i.e., supports the basic EstRes operation, while requires much less space and processing cost. Additionally, we describe a distributed procedure for building the distributed index of the system using this new structure as its building block.

### 4.1 Multi-Level Bloom Histogram

The Multi-level Bloom Histogram is based on the combination of two indexing structures used for summarizing XML documents, the Multi-level Bloom Filter and the Bloom Histogram.

Bloom filters [4] are compact data structures for probabilistic representation of a set that support membership queries (“Is element a in set A?”). Consider a set A = {a₁, a₂, ..., aₙ} of n elements. The idea is to allocate a vector v of m bits, initially all set to 0, and then choose k independent hash functions, h₁, h₂, ..., hₖ, each with range 1 to m. For each element a ∈ A, the bits at positions h₁(a), h₂(a), ..., hₖ(a) in v are set to 1. A particular bit may be set to 1 many times. Given a query for b, the bits at positions h₁(b), h₂(b), ..., hₖ(b) are checked. If any of them is 0, then certainly b ∉ A. Otherwise, we conjecture that b is in the set although there is a certain probability that we are wrong. This is a false positive. It has been shown [4] that the probability of a false positive is equal to \(1 - e^{-kn/m} \)^k. To support updates of the set A we maintain for each location i in the bit vector a counter c(i) of the number of times that
the bit is set to 1 (the number of elements that hashed to $i$ under any of the hash functions).

However, Bloom filters are not appropriate for summarizing XML data, since they cannot represent the hierarchical structure of the data. In [7], Bloom filters are extended to Multi-level Bloom Filters, which are appropriate for summarizing hierarchically structured data and support efficient processing of path queries, with low false positive ratios. Two structures are presented, the **Breadth** and **Depth** Bloom filters, that differ in the way they partition the XML tree into different levels of the filter. Depth Bloom filters are shown to exhibit better performance, i.e. lower false positive ratios, while being competitive in space requirements with the Breadth ones. Thus, in our work we deploy the Depth Bloom filter structure, though the extension we present can be also applied for the Breadth Bloom Filter.

In particular, the **Depth Bloom Filter** (DBF) for an XML tree $T$ with $j$ levels is a set of simple Bloom filters $\{DBF_0, DBF_1, DBF_2, \ldots, DBF_{j-1}\}$, $i \leq j$. There is one Bloom filter, denoted $DBF_i$, for each path of the tree with length $i$, (i.e., a path of $i + 1$ nodes), where we insert all paths of length $i$. Note that all the paths are inserted as a whole and each element of the path is not hashed separately.

Though Multi-level Bloom Filters support XPath queries, they do not provide any selectivity estimations. Another structure, based on simple Bloom filters, the Bloom Histograms [17], are designed for supporting selectivity estimations for XML path expressions. The Bloom Histogram uses a histogram to summarize the frequency of the path expressions that belong to a document, while using a Bloom filter to summarize the values, i.e. the path expressions that fall into each bucket of the histogram. A histogram $H$ for $D$ is therefore a two-column table $H(paths, v)$ with a fixed number of tuples, called buckets, $b$, where $paths$ represents a set of paths in $D$ and $v$ is a representative value for the frequency values of all path,$p$ in $paths$. Given a path $p$, we can find from $H$ a tuple $H_i$, with $p \in H_i$, $paths$ and return $H_i.v$ as an estimation of the frequency of $p$ in $D$. Bloom filters are used to represent the set of paths in each bucket so that, for a given path, the bucket containing the frequency of the path can be quickly located.

Combining the two structures we mentioned before, we introduce a new structure the **Multi-level Bloom Histogram**. Instead of using a simple Bloom filter for each level of the Multi-Level Bloom filter we use a Bloom Histogram, which is constructed with input only the path expressions that would be assigned to this level of a regular Multi-Level Bloom filter. Thus, a Depth Bloom Histogram (DBH) for an XML tree $T$ with $j$ levels is a set of Bloom Histograms $\{DBH_0, DBH_1, DBH_2, \ldots, DBH_{j-1}\}$, $i \leq j$. Each of the $DBH_i$ is a two column table $DBF_i(BF, v)$ with $b$ tuples (DBF$^b_i$), where $b$ is the number of the histogram’s buckets. The first column of a DBH$^i$ is a simple Bloom filter representing all the paths of length $i$ that fall into a bucket, and the second column is the representative value of frequency for the given bucket. Figure 2 depicts a DBH with 3 levels and 2 buckets for the path expressions of the path count table of the figure.
4.1.1 Bloom Multi-Level Histogram

Instead of using a Bloom Histogram for each of the levels in the Depth Bloom filter as is in the case of the Multi-level Bloom Histogram, we can take an alternate approach and use a multi-level Bloom filter for each bucket of a Bloom Histogram resulting in a new structure we call the **Bloom Multi-Level Histogram**. In this case, we first split the paths of the input XML document into buckets as we would if we used a simple Bloom Histogram, but instead of constructing a single Bloom filter summarizing the paths into each bucket we split the paths according to their length and build the corresponding Depth Bloom Filter for them. Thus a Bloom Depth Histogram (BDH) with \( b \) buckets for an XML tree with \( j \) levels, is a set of \( b \) Depth Bloom Filters \( DBF_1, DBF_2 \ldots DBF_b \), where each \( DBF_i \) each associated with a bucket \( i \). Thus each Depth Bloom Filter \( DBF_i \) summarizes the paths with frequency that falls into bucket \( i \).

The comparison of the two structures is not obvious. The first structure splits the paths primarily according to their length, while the second uses their frequencies as the primary criterion. Therefore, with data with a lot of paths with the same number of appearances the second structure is expected to perform worse than the first since the paths are not evenly distributed among the filters. However, it is clear that the performance of the two structures depends largely on the characteristics of the document collection.

4.2 Depth Bloom Histogram Construction

To construct a Depth Bloom Histogram of \( l \) levels with \( b \) buckets each, for a document \( d \), first we extract all possible distinct subpaths from \( d \) up to length \( l \), while recording the number of appearances of each subpath \( p \) in the XML document \((freq_p)\). Next, we split the subpaths into groups according to their length and sort them in no-increasing order, according to their \( freq_p \) values. For each of these groups we construct the corresponding Bloom Histogram, following the algorithm presented in [17] in order to select the bucket boundaries of the histogram and determine their values. The DBH consists of the set of Bloom Histograms constructed for each of the groups each corresponding to the level according to the group paths lengths.
Algorithm 2 ConstructDBH(DBH, d)

DBH: Depth Bloom Histogram, with d levels with b buckets each,
d: XML document

1: EstSet = NULL //Set of selectivity estimations for each level of the DBH
2: EstSet = NULL //Set of average selectivity estimations
3: Extract from d all possible subpaths p with length 0 to l
4: freqᵢ := number of occurrences for each subpath p in d
5: Split the set of subpaths p into groups, groupᵢ, according to their length i
6: for all groupᵢ do
7:   Build BH with b buckets
8: end for
9: Sort the BHs according to their paths length
10: RETURN DBH

4.3 Query Evaluation and Selectivity Estimation

To evaluate a query against a Depth Bloom Histogram we combine the query processing algorithms used for Depth Bloom Filters and Bloom Histograms ([7], [17]). First, the query q with length l is split to all possible subpaths of length 0 (single elements) to length l − 1. Each subpath is then hashed and checked against the Bloom Histogram corresponding to its length. In particular, the subpath is searched in all b Bloom Filters that comprise the Bloom Histogram of this level. If we have a match (match(subpath, DBF) = true) in more than one filters, then the average of the values in the corresponding buckets of the filters is returned as the query’s selectivity estimation. If any of the subpaths of q is not matched in the corresponding Bloom Histogram then q does not have an exact match in the indexed data. After all subpaths are evaluated, a set of selectivity estimations one for each of the subpaths of q is retrieved. For the query to have an exact 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.

When q contains either the wildcard or the "/" operator query evaluation again follows the algorithm described in [7]. In particular, the query is split at the position of this operator and the two resulting subqueries are evaluated separately by using the procedure we described above. To give an estimation on the total number of results for the whole query again we select the minimum selectivity estimations returned for the two subqueries. This procedure is also extended when more than one such operators are present by splitting the query to more than two subqueries by using again the position of the operators to select the splitting position.
Algorithm 3 EvaluateDBH(DBH, q, l)

DBH: Depth Bloom Histogram, with d levels with b buckets each,
q: query with length l
1: \text{EstSet}_i := \text{NULL} //Set of selectivity estimations for each level of the DBH
2: \text{EstSet} := \text{NULL} //Set of average selectivity estimations
3: Split q to all possible subpaths p with length 0 to l
4: for all \( p \) with length \( 0 \leq i \leq d \) do
5:     for \( j=1 \) to \( j=b \) do
6:         if match(\( p, DBH_i^j(BF) \))=true then
7:             \( DBH_i^j(v) \) to \text{EstSet}_i
8:         end if
9:     end for
10: end for
11: \textbf{return} Minimum of \text{EstSet}

5 Index Construction

To avoid the disadvantages of the centralized index, we consider building a distributed index over the data collection. In particular, our system is based on a hybrid architecture with a number of nodes having increased responsibilities, called superpeers. The superpeers are usually nodes that have increased capabilities (storage, processing, etc) and good stability properties. Each superpeer in the system is responsible for the management of a number of simple nodes, that lack the ability to process complicated queries due to limited resources. We assume that the superpeers are fully connected with each other forming the backbone of the overlay network, although other topologies are also feasible. Simple nodes are organized into groups supervised by a superpeer, through following a star topology, with each simple node connected to a superpeer (Fig. 3). The superpeers are responsible for processing the queries that are addressed to their group of nodes and also route the queries originated from their own group to the other groups in the system.

The superpeers are also responsible for maintaining the distributed index. If we assume the use of the CPI, then the index is partitioned into random disjoint units, with each unit assigned to a different superpeer. In our case, the DBH is used as the indexing structure for space and processing efficiency. Thus, each superpeer maintains a DBH that summarizes the information of all documents that belong in its partition unit. This DBH, denoted as clDBH, is constructed by merging the DBHs that correspond to each of the documents in the unit. When a new node joins the system, its documents are randomly distributed to the partition units. The node constructs the DBHs for its documents which are forwarded to the superpeers and merged with their clDBH.

To merge two DBHs we merge the respective BHs according to their level. The basic idea for merging two BHs is to merge together buckets from the
two initial DBHs that have the closest possible values. In particular, let us assume two DBHs, DBH(1) and DBH(2) with l levels with b buckets each. The procedure is followed for each of the Bloom Histograms that correspond to the different levels i of the multi-level Bloom Histogram. We examine each bucket value of DBH_i(2) one by one. Each such value is compared to all bucket values of DBH_i(1). After every bucket is examined, we merge bucket j of DBH_i(2) with the bucket of DBH_i(1) for which \( x_k = \min_{i=1}^{b} x_i \). The merging of the corresponding bloom filters is accomplished by applying the bitwise OR between the two Bloom filters and the value of the new bucket is the average between the values of the two buckets.

5.1 Building the Clustered Index

To determine the units in which we partition the index, we can use more intelligent techniques than a random partition. Our approach is based on exploiting the similarity between the indexed data and thus assigns similar data to the same partition unit. In particular, we cluster the documents based on their structural information. The produced clusters are then used as the units of partition for the index, resulting in a clustered index for the system. Our motivation behind building a clustered index is that a query is likely to find more results faster if similar data are indexed together. In addition, the clustered index assists in finding the best way to relax a query since all similar data are indexed together.

When a node joins the system, it constructs the structural summary of each of its documents (either a DBH or a simple path count table) and sends them to a random superpeer. This superpeer forwards the summaries to all other superpeers in the system. All superpeers work in parallel by comparing the new summaries against the summary of the data in their index unit, to determine the structural similarity between their indexed data and the new document.

If we use the path count table as the building block of the distributed index then the number of appearances of the path expressions of a document in the path count table of a cluster can be used as a measure to determine their structural similarity. This procedure is rather expensive since each path expression
that appears in the new summary needs to be looked-up in the cluster’s path
count table to retrieve the number of its appearances.

The use of the DBHs enables us to deploy a much more efficient construction
procedure. In this case, we derive the structural similarity between a new doc-
ument and an index partition, from the similarity between the corresponding
DBH and cDBH. To obtain this similarity score we define a similarity measure
between two simple Bloom filters and define a new algorithm that applies this
measure to evaluate the similarity between two DBHs.

**Definition 4 (Bloom Filter Similarity)** The similarity between two Bloom
filters BF1 and BF2 with m bits is defined as the number of positions in the
filter that both filters have set to 1 divided by m.

The algorithm EvaluateSim (alg. 4) that measures the similarity between
two DBHs, DBH1 and cDBH with l levels with b buckets proceeds as follows:
For each level of the two filters i, the Bloom filters that correspond to the
different buckets of the level are merged together to produce the Bloom filter
for the level cBFi. The similarity of the b Bloom filters corresponding to the
different buckets of the i-th level of DBH1 with cBFi is evaluated by using the
similarity measure we defined. Next, the weighted sum of the different similarity
scores of the b filters is calculated by using as weight their corresponding bucket
value. After this procedure is repeated for all l levels, the total similarity score
is evaluated by adding up all the level scores.

After receiving the similarity scores of all other superpeers for a specific
DBH the superpeer chooses the one with the largest similarity score as the
superpeer that is to index the new document. This superpeer is informed and
the corresponding DBH is merged with its cDBH.

**Algorithm 4 EvaluateSim(DBH, cDBH)**

\[ DBH: \] Depth Bloom Histogram, with d levels with b buckets each,
\[ cDBH: \] Cluster Depth Bloom Histogram, with d levels with b buckets each,

\begin{verbatim}
1: sim = 0
2: for i=0 to i=l do
3:   Merge all BHs of level i to cBFi
4:   sim_i := 0
5:   for j=0 to j=b do
6:     sim_i = sim_i + DBH^j_i(v) * sim(DBH^j_i,cBFi)
7:   end for
8: sim = sim + sim_i
9: end for
10: RETURN sim
\end{verbatim}
5.1.1 Discussion

There is an issue regarding the bootstrapping of the system. This is not the focus of our work, however we briefly discuss some techniques that can be used for the index initialization.

When no information about the data distribution is available we can use a simple heuristic based on a similarity threshold to assign the first documents to the clustered index. In particular, when a new document enters the system it is compared against the other indexed documents and if its similarity with the one selected as the most similar is largest than the predefined threshold it joins its cluster. Otherwise, it is assigned to a new cluster. This way we try to diffuse the dependency the formed clusters will have on the order upon which the documents entered the system. The threshold ensures that dissimilar documents are not appointed to the same cluster due to a lack of a better candidate. Furthermore, this way the number of the clusters is determined somewhat dynamically and it is derived from the data distribution.

The problem of finding an appropriate value for the threshold is a difficult one. If the value is very large we will result in a very large number of small clusters in our index. On the other hand, if the threshold is too small then it won’t be able to detect the structural differences between the documents. Knowledge about the data in the system and their schema can assist in finding an appropriate value for the threshold. Without that knowledge the system can dynamically tune the threshold in accordance to the system’s behavior. For instance, if there are a lot of clusters created that are sparsely populated the system can choose to decrease the threshold value. We note here that the use of the threshold is only required on the bootstrapping of the system until a number of well-formed clusters is created. However, we can extend its use so as to allow the creation of new clusters when very dissimilar documents enter the system, like a fine-tuning mechanism. In this case, the value of the threshold is set to a large value.

Another approach that avoids the use of a threshold is to consider as our main input parameter the number of clusters in the index $C$. In this case, when an appropriate sample of initial documents $D_i n$ has been inserted into the system we apply the well known $K$-means clustering algorithm to initialize the clustered index. A random superpeer is selected from the set of superpeers and gathers all documents to apply the $K$-means algorithm on their DBHs. As with any application of $K$-means, this approach is very sensitive to the initial sample of documents that belong to $D_i n$ and a different sample would result in a completely different result. In particular, if we consider the case where all documents in our initialization set belong to the same semantic category, $K$-means will still partition them into $C$ ($K = C$) clusters. To deal with this problem we enforce two additional constraints on the basic $k$-means. In particular, after finding the $C$ centroids in our system we check the pairwise distance of all of them and if this is lower than a value $\epsilon$ then we merge the two clusters into one, finding a new centroid for the new cluster. The now empty cluster can be filled with the new documents entering the system. To determine
which documents need to be placed into the empty clusters, we check the follow
constraint, if the distance of a document with its assigned cluster’s centroid is
larger than a value $\delta$ then we assign this document to a new cluster.

5.2 Query Routing and Results Retrieval

In a centralized index scenario, each query is forwarded to the index server which
applies the relaxation algorithm against the centralized index and evaluates
the list with the top-$k$ results ranked in increasing distance from the original
query. With no index present, the query is sent to each node and the results are
returned to the query origin which has to produce the final list of top-$k$ results
by merging the individual node lists. Furthermore, if all nodes cannot perform
the relaxation themselves, then a node responsible for them is forced to send
each transformed query produced at each step of the relaxation algorithm to all
nodes until it retrieves the satisfying number of results.

Our goal is to use a distributed index among a number of index servers
(superpeers) so as to both reduce the communication cost and the process-
ing required from each node or the responsible for the relaxation node in the
distributed scenario, and also reduce the load that a single server has in the
centralized scenario.

When a distributed index is deployed, the query needs to be processed
against each partition of the index and the partial results from different servers
maintaining the partition units need to be combined to derive the final result.
The process that is followed to retrieve the top-$k$ results proceeds in phases.
When a query is issued by a node, the node propagates it to the superpeer
it is attached to. In the first phase (Local Query Evaluation), the superpeer
forwards the query to all other superpeers in the system. In parallel, each su-
perpeer checks the query against its part of the index and applies the relaxation
algorithm to it for as long as it takes to attain the specified number of results
($k$). The results are sorted locally at each superpeer by using the distance mea-
sure between the matching relaxed query and the original query as a measure
of their quality.

By following a procedure commonly used in distributed top-$k$ algorithms for
minimizing the communication cost, in a second phase (Elimination Phase),
each superpeer sends its maximum distance score (the score of the $k$-th result)
to the other superpeers. Upon receiving these scores, each superpeer eliminates
from its $k$ result list, each result with a distance higher than the lowest of all the
maximum scores it has received. In the third phase (Final Result Construction),
each superpeer forwards the remaining results in its partial list to the superpeer
from which it has initially received the query. The superpeers only forward a
list with the respective index entries that correspond to their result sets and not
the actual data. After the initial superpeer gathers all the results, it merges the
different partial lists and locally ranks them so as to construct the final result
list, which is then send to the issuing node. On its turn, the node contacts the
nodes it finds in the ranked list to retrieve the actual documents.

The elimination phase of the routing process can consist of more than one
Algorithm 5 QueryEvaluation

Input:
q: Input query
N: number of clustered index units
k: number of requested results

1: q is forwarded to all N superpeers
2: **Phase 1**: Local Query Evaluation
3: for all superpeers i, 1 ≤ i ≤ N do
4:  Apply the Relax algorithm locally
5:  Construct its local result list
6:     Partial; := (expr, dist)
7: end for
8: **Phase 2**: Elimination Phase
9: for each superpeer i do
10:    Send the k-th distance score to all other superpeers, dist_i^k
11:    After receiving all (N − 1) distance scores dist_k
12:    remove from Partial; all expr_j with dist_j > max(dist_k)
13: end for
14: **Phase 3**: Final Result Construction
15: for each superpeer i do
16:    Send Partial; to initiator superpeer
17: end for
18: Initiator merges and sorts the Partial list it receives
19: Returns final top-K results
rounds so as to further prune the result list ([5]) and thus further reduce the size of the data that has to be transmitted over the network. However, there is a tradeoff with the query evaluation time. We define the pruning degree to measure the number of results that are eliminated in each round of the elimination phase of the query evaluation algorithm.

Let us consider $N$ superpeers, each maintaining a unit of the distributed index $U_i$, $1 \leq i \leq N$. We denote each iteration (round) of the elimination phase of the query processing algorithm as $round_i$.

**Definition 5 (Pruning Degree)** For each unit of the distributed index $U_i$, we define the pruning degree ($PD_j(U_i)$) at each round $i$ of the elimination phase of the query evaluation algorithm as the number of eliminated results ($Eliminated_i$) in the list $PL_i$ of the partial results of the superpeer $i$ responsible for unit $U_i$, divided by $k$.

$$PD_j(U_i) = \frac{Eliminated_i}{k}$$  \hspace{1cm} (2)

We consider the average pruning degree for each round $i$ of the elimination phase of the routing algorithm as:

$$PD_j = \frac{\sum_{i=1}^{N} Eliminated_i}{N}$$

$$PD_j = \frac{\sum_{i=1}^{N} Eliminated_i}{N \times k}$$ \hspace{1cm} (3)

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

The intuition behind the use of a clustered index is that since the data is grouped based on their similarity, the pruning degree of the elimination phase will be larger than in the case of a random partition where the data is uniformly scattered among the servers. In the clustered index, the servers responsible for the clusters that are relevant to the query will provide lower distance scores for their results and thus the other servers will achieve a higher pruning degree.

Let us first consider that the distributed index is used for indexing a collection of XML documents $D$ in which the documents can be classified to $M$ different categories $C_1, C_2, ..., C_M$ that have very low structural similarity, with $|C_1| = |C_2| = ... = |C_M| = x$. We consider a distributed index with $M$ units, i.e. $N = M$. Furthermore, let us assume that if any query $q$ matches a document $d' \in C_i$ then for all documents $d \notin C_i$, results($q, d$) = 0. By relaxing $q$ we can attain results from other categories as well. However, since the documents in different categories have low structural similarity, it holds that $d(q, d' \in C_i) < d(q, d \notin C_i)$. We want $k$ results for the query. Let us further assume that there exist $Ex$ exact matches to $q$ with $Ex > k$.

We consider two extreme scenarios for the distribution of the documents to the partition units of the index:

- **Uniform Distribution.** In this case, approximately $x/M$ documents of each category falls into each partition unit $U_i$. The exact matches to $q$,
are also uniformly distributed among the $M$ partition units. If we assume for simplicity that all other results to $q$ are of a distance $v$, then each partition unit return a $k - th$ result with distance $v$ of the original query. Thus, no pruning is achieved. Each superpeer has a pruning degree of 0 and the average pruning degree is 0 as well.

Even if the distance scores were different due to the uniform distribution assumption and the structural similarity between the documents of each category, the $k - th$ distance score for all the partition units would be close, and Eliminated $<< k$.

- **Clustered Distribution.** We use the classification taxonomy to split the documents to units corresponding to the category they belong to. In this case, the documents of category $C_1$ are allocated to unit $U_1$, of category $C_2$ to $U_2$ and so on. If query $q$ is then posed to the system, all its results are located in one unit, unit $U_1$. Thus, the distance score of the $k - th$ result that superpeer $t$ sends to the other superpeers is 0. Since there are no exact matches in the other clusters, the distance score of their first result in their partial result list is greater than 0. Thus, all superpeers except $t$ have a pruning degree of 1. The average pruning degree is $(N - 1)/N$.

It is obvious from these two extreme scenarios in which we compare the pruning degree for an optimal clustering that follows the classification of the documents to a uniform random partition that using a clustered index can improve the pruning degree in the query retrieval algorithm.

### 5.3 Update Propagation

To support updates in the system, we are based on the procedures proposed for supporting updates for the Bloom Histogram and the Multi-Level Bloom filter structures.

To support updates for the Bloom Histogram in [17] they consider an auxiliary structure that maintains a more analytical summary for the data in the $BH$ and first applies the update in this structure and then reconstructs a new Bloom Histogram based on the updated auxiliary structure. The auxiliary structure we use for this purpose is the full path count table because it offers larger accuracy than the alternative sketch structure also proposed in [17]. We deploy this procedure for applying the updates locally at each node for the documents it owns.

The next step, is to propagate the update to the indexing server responsible for the corresponding document. We treat the update as a two phase procedure. For a document $D_i$ with a corresponding $DBH_i$ that is updated to $D'_i$ and $DBH'_i$ respectively, the owner node sends to the index server responsible for the document an update message containing both $DBH_i$ and $DBH'_i$. When the index server receives this messages it needs to remove $DBH$ from its $dDBH$ and insert the new $DBH'_i$ instead. Based again on the corresponding algorithms for supporting updates for Depth Bloom filters, each bloom filter is a vector of
counters indicating the number of paths that set each position instead of a simple bit vector. Thus, the $cDBH$ can be updated by decreasing the counters in each of its simple filters. In particular, the deletion is performed by subtracting the value of a counter of the $DBH$ of the corresponding value of the counter in the same position in the $cDBH$. The values in the buckets remain fixed. After $DBH$ has been removed the new $DBH'$ can be easily inserted by applying the merging procedure that is used for the insertion of new documents in the system.

**Algorithm 6 UpdatedDBH($DBH, DBH', cDBH$)**

$DBH, DBH'$: Depth Bloom Histograms, with $d$ levels with $b$ buckets each and $RFs$ of size $m$

$cDBH$: Cluster Depth Bloom Histogram, with $d$ levels with $b$ buckets each and $RFs$ of size $m$

1: for $i=0$ to $i=d$ do
2:  for $j=0$ to $j=b$ do
3:   for $k=0$ to $k=m$ do
4:    $cDBH'_i[j][k] = cDBH'_i[j][k] - DBH_i[j][k]$ 
5:   end for 
6: end for
7: end for
8: Merge $DBH'$ into $cDBH$
9: RETURN $cDBH$

### 5.4 Motivation

The use of the clustered index enables the system to support an incremental query retrieval procedure exploiting the locality of the peers within a cluster. Instead of waiting for a query to be processed against all clusters before the results are returned to the originating peer, the system can provide the user with results incrementally as each index server finishes processing the query against its portion of the index, starting with the index server responsible for the peer that issued the query. We have to note that this alternative retrieval algorithm does not return to the user the top-$k$ results with respect to quality across the entire document collection distributed in the system. Whereas, each index server only evaluates the locally top-$k$ results and sends them to the user. This algorithm is able to reduce response times as the user will receive the first results a lot faster than having to wait for all index servers to cooperate so as to determine the final top-$k$ list. If the user is not satisfied by the results returned by the first cluster that responded it can go through the rest of the results as they arrive.
6 Experimental Evaluation

For the performance evaluation of our approach, we conduct two sets of experiments. The first set aims at evaluating the performance of the new indexing structure, while the second set concentrates on the distributed system performance with respect to top-k queries and the communication and processing cost they entail. We use real data for our experiments, that belong to predefined categories. In particular, we use the data found at the Niagara Project ([13]). We use documents belonging to 8 different categories. From the 8 different categories, there are two pairs of categories that share structural similarities, i.e., the Sigmod Record data with the bibliographical data and the actors data with the movies data, while there is little to none overlap between the rest of the document categories. For the query generation we use the zipf distribution to select paths belonging to the documents in the system and we apply randomly a structural transformation to them. The transformation we apply is either the deletion of the last element of the path, or the substitution of an element in a random position in the path with a random foo element, or the insertion of a subpath of random length in a random position in the path expression. For all the experiments we limited the number of levels in the Depth Bloom Histogram to 3 and the number of buckets per level to 4. For the Bloom filters of the structure we use 4 hash functions.

6.1 Multi-Level Bloom Histogram vs Bloom Histogram

In the first set of our experiments, we evaluate the performance of the multi-level bloom histogram indexing structure. We compare the false positive ratio and the estimation error of the new structure (MLBH) to the Bloom Histogram structure (BH) with respect to the size they occupy.

The estimation error is defined as the average absolute error given by the following equation:

$$\text{error} = \frac{1}{n} \sum_{i=1}^{n} |X_i - V_i|,$$

where the $n$ is the number of queries submitted to the DBF, $X_i$ is the number
Table 1: Input parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of documents</td>
<td>250</td>
</tr>
<tr>
<td># of clusters</td>
<td>8</td>
</tr>
<tr>
<td># of queries</td>
<td>1000</td>
</tr>
<tr>
<td>k</td>
<td>20</td>
</tr>
<tr>
<td># of document categories</td>
<td>8</td>
</tr>
<tr>
<td>Size of document</td>
<td>2KB-50KB</td>
</tr>
<tr>
<td>MLBH size</td>
<td>5 - 160KB</td>
</tr>
<tr>
<td>MLBH depth</td>
<td>3</td>
</tr>
<tr>
<td>MLBH hash functions</td>
<td>4</td>
</tr>
</tbody>
</table>

of actual appearances of the query $q_i$ in our data collection and $V_i$ is the corresponding estimation returned by the query evaluation algorithm $(\text{EvaluateDBH}(DBH, q))$.

In this experiment, we use exact match queries and do not apply the transformations we described to the paths generated by the zipf distribution.

Figure 4 shows that the new structure outperforms the Bloom Histogram both with regards to the false positive ratio and the estimation error. This is due to the query evaluation algorithm of the Depth Bloom Histogram that performs multiple checks for each query at different levels of the filter, whereas the Bloom Histogram only performs one.

6.2 Distributed System Performance

In the second set of experiments, we evaluate the performance of the system by using as our performance measure the average pruning degree of the system. We simulate a network of nodes in which each node is assigned a set of XML documents. We predefine a fixed number of nodes as the indexing servers of the system and we use four different approaches for constructing the distributed index. The two first approaches use the path count table ($PCT$) indexing structure as the building block of the distributed index, while the latter two use the multi-level Bloom histogram ($MLBH$). Furthermore, using both indexing structures we construct both a random index, where the documents are assigned to indexing servers uniform at random ($\text{randPCT}$ and $\text{randMLBH}$) and a clustered index where we use the construction processes we described for each of the two indexing structures ($\text{clustPCT}$ and $\text{clustMLBH}$). The Relax algorithm is used for query evaluation.

**Influence of the number of clusters.** In the first experiment we measure the pruning degree of the four approaches with respect to the number of clusters, i.e. superpeers in the system. We fixed $k$ to 20 and the number of indexed documents to 250. We vary the number of clusters from 4 which is half of the actual categories the documents belong to, to 24. Thus, for 4 index servers,
each server is assigned documents belonging to two different categories, while for a server number larger than 8 the documents of the same category are split among different servers.

Figure 5(left) shows that as we assign multiple categories to the same superpeers the pruning degree of the system decreases for both the clustered index approaches, while it remains almost constant for the random ones. This is due to the fact that if the documents from one category are distributed among different superpeers then the relaxation algorithm applied to all of them produces results with similar quality, thus the elimination phase of the results retrieval algorithm is not efficient. If we increase the number of clusters further then the pruning degree will decrease further almost reaching the performance of the uniform distribution scenario. Using as many superpeers as the number of document categories or even less, alleviates this problem. However, we have to consider that the load that the superpeers have depends on their number. Therefore, there is a tradeoff between the communication cost and the load-balance of the system.

Scaling. In this experiment we measure the scaling capability of our approach by increasing the number of documents that are indexed in the system. The number of the index servers is fixed to 8 and k=20.

As figure 5(right) illustrates the performance of the system scales gracefully when increasing the number of shared documents. The pruning degree only slightly decreases for all four approaches. Furthermore, it is evident how both the clustered index-based approaches clearly outperform the respective random ones.

Construction Cost We repeated the first two experiments to measure the construction cost of the clustered index when varying the number of clusters in the system (Fig. 6(left)) and the number of shared documents (Fig. 6(right)). We measure the construction cost of each approach as the total size of the messages that need to be exchanged between the nodes of the system to construct the distributed index. We compare only clustPCT and clustMLBH since we already showed how these two approaches outperform the respective random ones in terms of the pruning degree they achieve. The goal of this experiment is to show that although the clustPCT approach results in a higher pruning
degree than \textit{clustMLBH}, its construction cost is much more high. Figure 6 illustrates this result. While the cost of \textit{clustMLBH} remains in the order of a few MBs the cost of \textit{clustPCT} reaches the order of GBs.

\textbf{Influence of $k$} In this experiment, we examine the influence of $k$ with respect to the pruning degree of the system. We vary $k$ from 10 to 50, while keeping the rest of the parameters fixed to their default values.

As $k$ increases the pruning degree decreases for all approaches (Fig. 7). This is because as more results are required the results contained in the documents belonging to the category of the query do not suffice. Thus, the query needs to be further generalized and results that belong to other document categories, i.e. maintained at more than one superpeer, are also required.

To get a better understanding of the influence of $k$ on the performance of our system we have to examine its relationship with the selectivity of each query, i.e. the actual number of results available for each query. Thus for each query we also measure the actual results maintained in the system.

\textbf{Incremental Results Retrieval}

In this experiment we compare the threshold based result retrieval algorithm
to the incremental variation. In particular, we apply the same set of queries on the same system configuration and compare the results returned by the threshold-based algorithm with the results returned from the best (first) cluster with the incremental approach. We measure the percentage of results of the second approach that also belong in the result list returned by the first, i.e. the percentage of results returned by the best cluster that are actually part of the best $k$ results in the system.

6.3 Updates Propagation

This set of experiments aims at evaluating the performance of the DBH structure as the clustered index in case of updates.

Since an update consists of a delete of the old DBH and the insertion of a new one, the message required for propagating an update performed at the local data of a peer to the appropriate portion of the clustered index is twice the size of the DBH structure. Furthermore, there is some additional information required which denotes for each bucket of the old DBH filter with which bucket of the clDBH it has merged so that it can be removed when performing the update. Therefore, the total size of the message is $2 \times \text{size(DBH)} + (b \times l \times \text{sizeof(integer)})$, where $b$ is the number of buckets per level and $l$ is the number of levels of the filter. Note that we do not consider the case where the new DBH needs to be moved into another cluster into our experiments. However, this could be easily addressed by sending the new DBH for insertion to the corresponding index server after the appropriate cluster was determined. Determining the need for such a relocation is not in the scope of this paper and is left as future work.

Since the performance of the update propagation with respect to the communication cost and bandwidth consumption is straightforwardly determined, we concentrate our evaluation on determining how well the DBH’s estimation accuracy is maintained after a series of updates. To this end, we consider a centralized DBH structure and perform a number of deletions and insertions on it and measure the estimation error after the updates. We assume that the deleted document has no similarity to the new inserted document. Observe that this is a worse case scenario, since usually updates in a document are not that drastic to completely change its structure. We measured both the false positive ratio and the estimation error after performing an increasing number of updates in the structure. The original structured maintained 250 documents and we repeated our measurements varying the number of updates performed from 0 (to show the structure’s performance before any updates) to 100. Our results that are depicted in Fig. 8, show that the false positive ratio remains almost constant even after a considerable number of updates. The estimation error increases more as the number of updates also increases. This indicates the need for rebuilding the cluster index after a considerable number of updates has occurred so as to preserve its accuracy.

We repeated the same experiment using documents from a single cluster maintained into the structure. That is the DBH summarized only documents belonging to a single category. In this case, the deleted and newly inserted
Figure 8: (left) False Positive Ratio and (right) Estimation Error with an Increasing Number of Updates

Figure 9: (left) False Positive Ratio and (right) Estimation Error with an Increasing Number of Updates

document also belonged to the same cluster. This experiment was performed so as to examine how the behavior of the structure changes when the updates occurring are less drastic than in the first scenario. This is a more realistic scenario resembling the updates that are performed in a distributed setting at each clustered partition. Our results show that though both errors are decreased when the updates are not as drastic, still after a large number of them the need to rebuild the structure arises (Fig. 9). Therefore, we conclude that the measure of the change in the document does not influence the behavior of the structure drastically.

To determine the point in which a reconstruction is required we can use the following technique. We run a batch of updates in the system and we take two approaches. In the first approach, we apply the updates on our DBH using the update propagation procedure we proposed (DBH₁) while in the second approach we do not apply the updates on the DBH but reconstruct a new DBH taking as input the updates data (DBH₂). Next, we apply a batch of queries and compare the top-k results that both structures provide. When the two result list become too different we can determine that it is the appropriate time to reconstruct DBH₁.

Since the updates damage the selectivity estimations provided by the structure, i.e. they increase the estimation error, they may lead to unnecessary
processing by the Relax algorithm. In particular, the algorithm may choose to perform the wrong transformation based on a large estimation of frequency while the actual number of results that correspond to the real query are not that many and thus another transformation would be more preferable.

7 Conclusion and Future Work

In this paper, we address the problem of the efficient evaluation of XPath approximate queries over dynamic distributed collections of XML data. We focus on reducing the communication cost required for evaluating top-k queries in such settings, by introducing the use of a distributed clustered path index which enables the nodes responsible for query evaluation to prune the number of candidate results they need to consider. We present a relaxation technique that exploits the clustered index and an appropriate distance measure for ranking the results. Furthermore, we deploy a new efficient indexing structure, the Multi-Level Bloom Histogram, as the building block of our index. This compact structure enables us to provide an efficient construction procedure for the clustered index.

Since we are dealing with a dynamic environment we anticipate content updates in the documents. This may result to documents being maintained in clusters with which they no longer share structural similarities. Thus, such a document needs to be re-assigned to a new cluster which is more similar to its new content. Furthermore, the insertion or deletion of documents may arise the need for splitting or merging existing clusters. As a future direction, we plan to extend our approach so as to dynamically adjust the clustered index to suit such updates.

Acknowledgments

This paper is part of the 03-ED-591 research project, implemented within the framework of the "Reinforcement Programme of Human Research Manpower" (PENED) and co-financed by National and Community Funds (25% from the Greek Ministry of Development-General Secretariat of Research and Technology and 75% from E.U.-European Social Fund).

References


