

Efficient Top- k Search across Heterogeneous XML Data Sources

Jianxin Li[†], Chengfei Liu[†], Jeffrey Xu Yu[‡], and Rui Zhou[†]

[†]Swinburne University of Technology, Melbourne, Australia,
{jili, cliu, rzhou}@ict.swin.edu.au

[‡]Chinese University of Hong Kong, China
yu@se.cuhk.edu.hk

Abstract. An important issue arising from XML query relaxation is how to efficiently search the top- k best answers from a large number of XML data sources, while minimizing the searching cost, i.e., finding the k matches with the highest computed scores by only traversing part of the documents. This paper resolves this issue by proposing a bound-threshold based scheduling strategy. It can answer a top- k XML query as early as possible by dynamically scheduling the query over XML documents. In this work, the total amount of documents that need to be visited can be greatly reduced by skipping those documents that will not produce the desired results with the bound-threshold strategy. Furthermore, most of the candidates in each visited document can also be pruned based on the intermediate results. Most importantly, the partial results can be output immediately during the query execution, rather than waiting for the end of all results to be determined. Our experimental results show that our query scheduling and processing strategies are both practical and efficient.

1 Introduction

Over decades, processing top- k query has been extensively studied in different research areas, such as relational databases [1–3], multimedia databases [4–9] and keyword search [10, 11]. Recently, Efficiently computing top- k answers to XML queries is gaining importance due to the increasing number of XML data sources and the heterogeneous nature of XML data. Top- k queries on approximate answers are appropriate on structurally heterogeneous data. On the one hand, it is difficult for users to formulate their queries exactly and search the exact answers. On the other hand, an XML query may have a large number of answers, and returning all answers to the user may not be desirable. The top- k approach can limit the cardinality of answers by returning k answers with the highest scores.

The problem of finding top- k answers within a large XML repository has been studied in [12], where an adaptive strategy is proposed for filtering out some unqualified candidates. However, its searching overhead is expensive due to frequent adaptivity among possible candidates and dynamic sort of partial

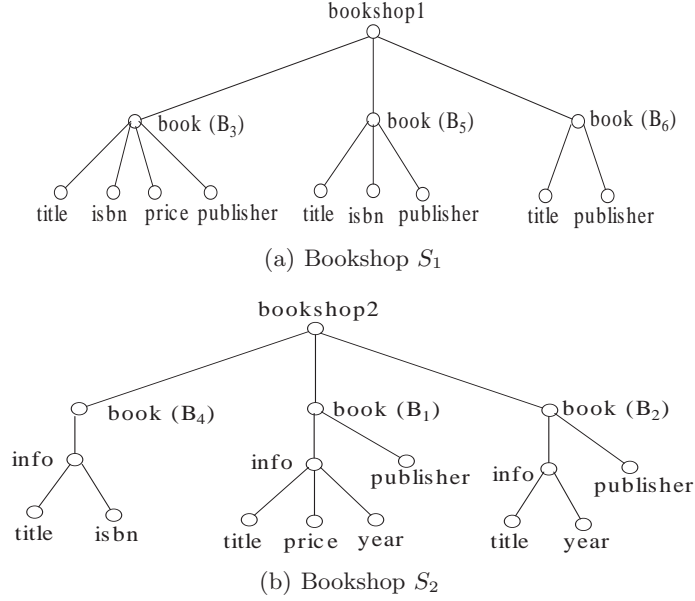


Fig. 1. Bookshop Example

matches. Furthermore, this work only considers query evaluation in a single XML document. For many applications, it is more meaningful to find top- k results from multiple heterogeneous XML data sources. In this paper, we target this problem by proposing a bound-threshold based scheduling strategy (in short *BT* strategy) with the help of schema information of each XML data source. We are not required to evaluate a top- k query over all data sources. Instead, we schedule the query to the most relevant ones by leveraging the schema information, which can produce top k results as early as possible and output each result immediately after it is generated.

Example 1. Consider two bookshop XML data sources in Figure 1 that maintain the partial or full information of each book: *title*, *isbn*, *price*, *publisher* and *year*. To search for two books (top- $k=2$) that contain “XML” in their titles and also include other specific information: expected price, published time and publisher, we can represent it as a tree pattern query q in Figure 2(a) where nodes are labeled by element tags, leaf nodes are labeled by tags and values, and edges are XPath axes (e.g. *pc* for parent-child, *ad* for ancestor-descendant). The root of the tree (shown in a solid circle) represents the distinguished node.

A naive solution to the above top-2 query is to retrieve the two most relevant books from each source and then select the more relevant ones by comparing their scores. However, this approach is not desirable for a large number of data sources due to amount of unnecessary processing cost. To solve this problem, we deploy XML schema information because a schema embodies to some extent the maximal structural information in the corresponding data sources that conform

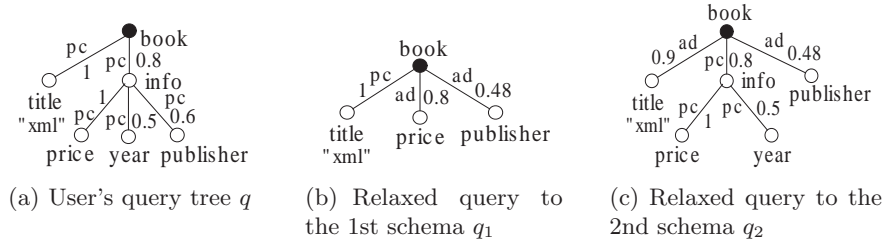


Fig. 2. Query Tree and Relaxed Forms to different schemas

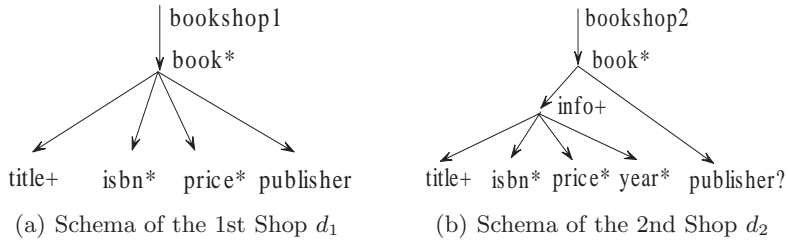


Fig. 3. Bookshop Schema Example

to the schema. For example, in Example 1, the two bookshop sources conform to schema d_1 in Figure 3(a) and d_2 in Figure 3(b), respectively. Apparently, we can see that d_2 is more similar to the query q than d_1 in Figure 3. Consequently, we may expect to get more relevant results from S_2 than S_1 at the first instance. To achieve this, query relaxation can be used [13, 14]. The top-2 query q against d_1 and d_2 can be relaxed into q_1 and q_2 , which are shown in Figure 2(b) and Figure 2(c), respectively. In other words, q_2 is more similar to q than q_1 , therefore, we may schedule to evaluate q_2 on S_2 first. If we are able to get enough qualified books from S_2 , we do not need to evaluate q_1 on S_1 at all. By a qualified book, we mean that the book contains more required information than any book in all other sources. In the example, two approximate results B_1 and B_2 in S_2 are qualified because both of them contain more information than any book in S_1 with regards to the original query q . As such, these two books may be returned as the results for the top-2 query.

However, not every approximate result returned from q_2 is qualified. For example, if we are evaluating a top-3 query, B_4 in S_2 may not be qualified because it contains less information than B_3 in S_1 . This is because that a result (XML fragment) conforming to a schema may not necessarily contain all structural information of the schema. For a schema represented in DTD, we are allowed to specify disjunctive semantics (denoted as “|”) and optional semantics (denoted as “*” or “?”). In other words, XML documents conforming to the same schema may vary in their structures. If we fail to find any or enough qualified results in one data source, we may try the next most relevant data source, say S_1 in the example to work out the results or rest of the results, say B_3 in the example for the top-3 query.

From the motivating example, it is not hard to find that for a large number of data sources, the processing time can be reduced significantly by adaptively scheduling user’s query on the most relevant data source at the time and progressively evaluating it according to schema information. Bearing this in mind, we design our upper/lower bounds and threshold for the BT-based strategy. This paper makes the following contributions:

- It proposes a BT-based scheduling strategy for efficiently searching top k results where we can skip most of the XML data sources according to schema information and also prune most candidates in each visited data source.
- It guarantees that results generated can be output immediately without waiting for the end of query evaluation.
- It provides stress tests and large-scale performance experiments that demonstrate the scalability and significant benefits of the proposed scheduling strategies.

The rest of the paper is organized as follows. In Section 2, we first introduce preliminary knowledge for query relaxation and then formalize top-k problem over a large number of XML data sources. Section 3 presents an overview of the BT-based strategy based on schema information and any scoring function that is used to measure the relevance of XML fragments with regards to a relaxed query. The detailed discussions of the BT-based scheduling strategy and its corresponding algorithms are given in Section 4. The experimental results are reported in Section 5. Finally, we discuss the related work and conclude the study in Section 6 and Section 7, respectively.

2 Preliminary and Problem Statement

XML Query Relaxation: In this paper, we represent XML queries as Tree Patterns [15] and allow users to add a weight on each edge to show their preferences on different path steps. We relax an XML query to different data sources based on the structural information provided in their corresponding DTDs while compute the changing weight of edges in query.

To guarantee that a relaxed query can be evaluated on a source, we need to collect information such as whether a node in the query appears in a DTD, the relationships between two query nodes, and the cardinality of a query node in a DTD, etc. Given a query q and a DTD d , the target of query relaxation is to find the relaxed query q' that is best suited to be evaluated on sources that conform to d by calling a set of relaxation operations: *deleting* a node, *generalizing* a pc-edge into a ad-edge and *promoting* a node [13, 14]. The minimal requirement for the relaxed query q' is that the root node r is kept in q' . For example, the query q in Figure 2(a) does not match exactly with the schema d_1 of the 1st shop in Figure 3(a). Therefore, in order to provide considerate and reliable service for users, relaxing the query against DTDs for the conformed documents is strongly in demand. According to the structural information in the schema d_1 , we firstly delete the nodes *info* and *year*. And then the nodes *price* and *publisher*

are promoted under the distinguished node *book* where they are connected with ancestor-descendant (ad) edges. The relaxed query is shown in Figure 2(b).

Problem Statement: Consider a weighted top-k query q and a large number of data sources $\{S_1, S_2, \dots, S_n\}$ that conform to different DTDs $\{d_1, d_2, \dots, d_n\}$ respectively. Let $\{q_1, q_2, \dots, q_n\}$ be the set of weighted relaxed queries of q with regards to the set of DTDs. Our aim in this paper is to efficiently search top k results by scheduling the evaluation of $\{q_1, q_2, \dots, q_n\}$ over the set of data sources.

3 Overview

As stated in the problem statement, a set of relaxed queries $\{q_1, q_2, \dots, q_n\}$ can be generated from the original query q based on the conforming DTDs $\{d_1, d_2, \dots, d_n\}$, respectively. To start with, we require to rank the similarity between each q_i and q . During the query evaluation on a data source, we also check if a returned result is qualified or not. In this regard, we need a scoring function.

In a tree pattern query q , a user may specify, on an edge $e(v_1, v_2)$, how close v_1 and v_2 are associated with each other. To compute the weight of a query q , a naive function is to combine the weights of all edges in the query together where the edges are assumed to be independent from each other [14]. However, according to common understandings about XML queries, we find (1) the more path steps there are between two nodes, the less related the two nodes may be; (2) nodes lying on different paths are not related with each other, i.e., nodes are only related with their ancestors and descendants. Keeping these two features in mind, we introduce the concept of extended edge weight between a pair of nodes with *ad* relationship, $ad(v_i, v_j)$. The extended edge weight can be derived by multiplying weights along the path from v_i to v_j . Extended edge weights can be computed on the fly when required, or be calculated out beforehand, and then maintained dynamically.

Definition 1. Extended Edge Weight: Let a WTPQ $q = (V, E, r, w)$, for two nodes $v_i, v_j \in V$ ($i \neq j$), the extended edge weight between v_i and v_j , denoted as $w_e(v_i, v_j)$, is defined as follows: if $ad(v_i, v_j)$ or $ad(v_j, v_i)$ holds, let w_1, w_2, \dots, w_n be weights on edges along the path from v_i to v_j , $w_e(v_i, v_j) = \prod_{t=1}^n w_t$; otherwise, $w_e(v_i, v_j) = 0$. Note that extended edge weight is a symmetric function on v_i and v_j , i.e., $w_e(v_i, v_j) = w_e(v_j, v_i)$.

Based on Definition 1, we can score the weight of a tree pattern query q by summing all extended edge weights of q , i.e., $score(q) = \sum_{\forall v_i, v_j \in V, ap(v_i, v_j)} w_e(v_i, v_j)$ where $ap(v_i, v_j)$ means either $ad(v_i, v_j)$ or $pc(v_i, v_j)$. Similarly, we can measure the similarity of a potential result rooted at any node v in source S with q by summing the weights of those extended edges that match q . We denote this as $score(v, q)$. For q_1 and q_2 in Figure 2, we have $score(q_1) = w_e(book, title) + w_e(book, price) + w_e(book, publisher) = 1 + 0.8 + 0.48 = 2.28$ and $score(q_2) = w_e(book, title) + w_e(book, info) + w_e(book, publisher) + w_e(book, price) + w_e(book, year) + w_e(info, price) + w_e(info, year) = 0.9 + 0.8 + 0.48 + 0.8 \times 1 + 0.8 \times 0.5 + 1 + 0.5 = 4.88$. For the potential results, we have $score(B_1, q_2)$

= $score(q_2)$ because B_1 covers all edges of q_2 ; $score(B_2, q_2) = score(q_2) - w_e(book, price) - w_e(info, price) = 3.08$; Similarly, we have $score(B_3, q_1) = score(q_1) = 2.28$ and $score(B_4, q_2) = 1.7$, and scores for B_5 and B_6 are less than that of B_4 .

Come back to our problem, we are now able to compute the scores of the relaxed queries $\{q_1, q_2, \dots, q_n\}$ as $score(q_1), score(q_2), \dots, score(q_n)$ respectively. The BT-based strategy we propose is based on the concepts of upper/lower bounds [16] and threshold. We initialize the upper bound $U(i)$ and lower bound $L(i)$ of each source S_i as $score(q_i)$ and zero, respectively. To start our adaptive scheduling, we first choose the data source to be evaluated as S_{k_1} if $U(k_1) = \max\{U(i) | 1 \leq i \leq n\}$ (i.e., the highest upper bound) and the threshold $\sigma = U(k_2) = \max\{U(i) | 1 \leq i \leq n \wedge i \neq k_1\}$ (i.e., the next highest upper bound). Then we start to evaluate q_{k_1} on S_{k_1} by probing an edge $e(v_1, v_2)$ of q_{k_1} at a time. If $e(v_1, v_2)$ cannot be found in S_{k_1} , $U(k_1)$ will be decreased; otherwise, $L(k_1)$ will be increased. The probing continues for next edge of q_{k_1} until either $L(k_1) \geq \sigma$ or $U(k_1) < \sigma$. If $L(k_1) \geq \sigma$, all the candidates may become possible results depending on the value of k required in top- k . If the number of candidates equals to k , all the candidates can be returned as qualified results and the process stops; if the number of candidates is less than k , all the candidates can also be returned as qualified results (with the adjustment of k) but the probing process continues on S_{k_1} ; otherwise, more probing is required to refine the qualified results. If $U(k_1) < \sigma$, we will continue the process. The next data source to be evaluated will be S_{k_2} and the threshold will be chosen based on the updated list of the upper bounds. The process stops until k results are returned.

In our example, S_2 is chosen as the the data source to be evaluated first because $U(2) = score(q_2) > U(1) = score(q_1)$ at the beginning. If we have a top-2 query, B_1 and B_2 in S_2 will be returned as qualified results because both $score(B_1, q_2)$ and $score(B_2, q_2)$ are no less than the threshold $U(1) = score(q_1)$. If we have a top-3 query, we will first have B_1 and B_2 in S_2 as qualified results but the probing in S_2 continues until B_4 is met. At this time, $U(2)$ is decreased to $score(B_4, q_2) = 1.7$, which is less than the threshold $U(1) = score(q_1) = 2.28$. So the next source to be evaluated is switched to S_1 , and its threshold is $score(B_4, q_2) = 1.7$. Since $score(B_3, q_1) (=2.28)$ is greater than the new threshold, B_3 becomes the third qualified result.

If the number of the data sources is large, we can avoid to evaluate most of the data sources based on the BT scheduling strategy. In addition, the qualified results can be returned immediately without waiting for all results to be determined.

4 Scheduling Strategy for top- k Queries

In this section, the BT-based scheduling strategy for evaluating top- k queries will be discussed in detail. Specifically, in Section 4.1 we introduce data source and result determination properties that can be applied to schedule query evaluation over different data sources. Then, in Section 4.2 static/dynamic strategies are

proposed to evaluate the edges, which can reduce unnecessary computational cost. Finally, in Section 4.3 we design a set of algorithms for the BT-based scheduling.

4.1 Data Source and Result Determination Properties

From the overview of the BT-based scheduling strategy in Section 3, we can get the following two properties.

Property 1. Data Source Determination and Switching: At any time of query evaluation, we always evaluate the data source S_{k_1} that has the highest upper bound $U(k_1) = \max\{U(i) | 1 \leq i \leq n\}$. When an edge $e(v_1, v_2)$ in q_{k_1} is evaluated on the data source S_{k_1} , if it turns out that $e(v_1, v_2)$ cannot be successfully evaluated on the fragments rooted from all of the distinguished nodes of S_{k_1} , then the upper bound $U(k_1)$ will be decreased by $U(k_1) = U(k_1) - \text{score}(v_2, q_{k_1}) - w_e(v_1, v_2)$. Suppose that the threshold $\sigma = U(k_2)$, then we have:

- If the updated upper bound $U(k_1)$ is still larger than or equal to the threshold σ , then we need to continuously evaluate other edges in the query over the current data source S_{k_1} .
- If the updated upper bound $U(k_1)$ becomes lower than the threshold σ , then the current data source S_{k_1} needs to be suspended and query evaluation will be switched to the data source S_{k_2} .

Property 2. Result Determination: When an edge $e(v_1, v_2)$ in q_{k_1} is evaluated on the data source S_{k_1} , if it turns out that $e(v_1, v_2)$ can be successfully evaluated on the fragments rooted from some of the distinguished nodes of S_{k_1} , then the lower bound $L(k_1)$ will be increased by $L(k_1) = L(k_1) + w_e(v_1, v_2)$. Suppose that the threshold $\sigma = U(k_2)$ and the updated lower bound becomes larger than σ . Then we can affirm that some candidates generated so far in S_{k_1} must be qualified as top- k results. We divide the set of candidates in S_{k_1} into two groups G_1 that satisfies $e(v_1, v_2)$ and G_2 that does not, then the two groups will have different upper/lower bounds. Suppose that $\sigma \geq U(k_1)(G_2)$, then we have:

- If $|G_1| = k$, all the candidates in group G_1 can be returned as the qualified results and searching task would be terminated.
- If $|G_1| < k$, all the candidates in group G_1 can be returned as the qualified results and the k value will be decreased by $k = k - |G_1|$. Then the group G_2 should be evaluated if it is not suspended. If all the other groups in the data sources have been suspended, then we should switch to the next data source based on Property 1.
- If $|G_1| > k$, we will evaluate other edges in the query q_{k_1} on G_1 to find the top k results.

Consider the top-2 query in our example again. We first evaluate q_2 on S_2 because $U(2)$ is larger than $U(1)$ (Property 1). Then we will choose some edges in q_2 to be evaluated, such as $(book, title)$, and $(book, info)$. All the edges can be

found in the candidates of S_2 . After that, the lower bound of the data source will increase to 1.7 (i.e., $L(2) = 0.9 + 0.8$). Then suppose we continue to evaluate $(info, year)$, at this point, we have two groups. The group G_1 of B_1 and B_2 satisfies $(info, year)$ while the group G_2 of B_4 does not. The lower bound of G_1 is increased to 2.6 ($L(2)(G_1) = 0.9 + 0.8 + 0.8 \times 0.5 + 0.5$) while the upper bound of G_2 is decreased ($U(2)(G_2) = 4.48 - 0.9 = 3.58$). When $(info, price)$ is evaluated, the upper bound of G_2 is further dropped to 1.78 ($3.58 - 0.8 \times 1 - 1$). To this point, the 2 candidates in G_1 can be output as qualified results because $L(2)(G_1) > \sigma$ and $U(2)(G_2) < \sigma$. The process stops here.

4.2 Edge Selection and Unqualified Edge Reduction

According to the above properties, the relationships among $U(k_1)$, $L(k_1)$ and σ need to be checked during query evaluation. Obviously, changing the value of the three variables will produce different query evaluation sequences over the large number of data sources. But the changing is likely to be influenced to some extent by the next edge that will be evaluated. Therefore, the selection of next edge can also affect the performance of query evaluation. In this section, we first introduce three ways to determinate the next edge. Then we discuss how to filter unqualified edges during query evaluation.

Intuitively, there are three processing strategies for determining next edge: *random* i.e., the next edge can be evaluated at random; *min_weight* i.e., the edge with the minimal weight can be evaluated first and *max_weight* i.e., the edge with the maximal weight can be evaluated first. For the first two strategies, the possibility that some data sources would be visited frequently is likely to be increased to some extent, which may lead to unnecessary costs. For the third one, at every time the edge with the maximal weight is selected to be evaluated, so that it has the higher possibility to increase the score of $L(k_1)$ if the edge can be found, otherwise, the score of $U(k_1)$ would be decreased at most. Both of the trends are likely to locate the data sources as early as possible that can return the answers. Therefore, the last one would yield a better performance.

Besides next edge selection, the determination of selection range is another important factor to improve the performance of query evaluation. A simple method is to consider all edges together at the beginning and rank them based on the weights of their corresponding subtrees, denoted as *static* style. Although it makes next edge selection very easy in real application, some edges that should be filtered out based on the intermediate feedbacks have to be still evaluated. Therefore, an optimized approach is proposed to incrementally expand the selection range, denoted as *dynamic* style. The reason that the dynamic approach can do better than the static one depends on the disjunctive and optional semantics in DTD. For example, if an edge (e.g. x/y) in a query is specified as optional in a DTD and does not exist in a data source conforming to the DTD, then all the edges coming from the element y are not required to be evaluated because they cannot exist in the current fragments. Therefore, if we expand the selection range in a dynamic style, some edges can be filtered beforehand based on the intermediate results.

Algorithm 1 BT-based Scheduling Strategy

input: a set of weighted relaxed queries $\{q_1, q_2, \dots, q_n\}$ rooted at $\{r_1, r_2, \dots, r_n\}$ and a set of data sources $\{S_1, S_2, \dots, S_n\}$

output: top k results

- 1: call for the function *computingScore()* in Algorithm 4 to compute query weight as upper bound for each data source and denote the two highest upper bound as $U(k_1)$ and $\sigma = U(k_2)$ where $U(k_1) \geq U(k_2)$, $L(k_1) = 0$;
 - 2: // $\{S_{k_1}$ will be first evaluated}
 - 3: put all candidates in S_{k_1} into group G ;
 - 4: **if** $ch(r_{k_1}) \neq \phi$ **then**
 - 5: list $l = \text{sortAllChildNodes}(ch(r_{k_1}))$;
 - 6: *ScheEval*($l, q_{k_1}, G, U(k_1)(G), L(k_1)(G), \sigma$) in Algorithm 2;
-

4.3 BT-based Scheduling Strategy

We use Algorithm 1 to initialize query evaluation over the data source S_{k_1} . Algorithm 4 is used to compute the weight of each subtree in the query q_{k_1} and $score(q_{k_1})$ is taken as the initial value of the upper bound $U(k_1)$. Based on the BT scheduling strategy, we always evaluate the query q_{k_1} on the data source S_{k_1} with the highest upper bound $U(k_1)$ at any point. Then all the candidates in the data source S_{k_1} can be clustered initially into one group G by using index or other technologies. After that, we will evaluate the edges in the query q_{k_1} in a similar *breadth-first search* (BFS). To this end, three functions are deployed during query evaluation: *sortAllChildNodes()* sorts a list of nodes based on the weight of the subtrees rooted at these nodes where any traditional sorting algorithm can be applied (e.g., Insert Sort in [17]); *mergesort()* merges two sorted lists like Merge Sort in [17], which can improve the sorting efficiency because the previous list has been sorted before; *getFirstNode()* gets the first node from the sorted list l . At last, we will call for Function *ScheEval()* to probe a data source. Based on the evaluated results, we determine how to proceed at next step. The detailed procedure is described in Algorithm 2.

In Algorithm 2, we first get a node v with the function *getFirstNode()* and evaluate the edge $e(v'parent, v)$ over the group of candidate nodes G . There are three possibilities. **(1)** Line 4 - 12: If no candidates in G satisfy the evaluated edge e , then the upper bound $U(k_1)(G)$ for the group will get a penalty $score(v)$, i.e., subtracting the score of the subtree rooted at v from the current upper bound. After that, we will compare the updated $U(k_1)(G)$ with the threshold σ . If $U(k_1)(G)$ is lower than σ , the current group will be suspended. And then previous groups or next data source will be evaluated depending on the conditions $\sigma = U(k_1)(G_x)$ or $\sigma = U(k_2)$, respectively. **(2)** Line 13 - 18: If all candidates in G satisfy the evaluated edge e , then the lower bound $L(k_1)(G)$ for the group will be increased by summing the extended weight $w_e(v'parent, v)$ of the edge. If $L(k_1)(G)$ is higher than σ , it means that the current group contains part or all results that can be determined by Function *determineCandidates()* in Algorithm 3. Otherwise, we open the child nodes of the node v to expand the current range of edges because the group of candidates can not be determined based on the current edge e so far. **(3)** Line 19 - 30: Most of the time, only part candidates

Algorithm 2 ScheEval(a list l , query q , group G , $U(k_1)(G)$, $L(k_1)(G)$, σ)

```
1: while  $l \neq \phi$  do
2:   node  $v = \text{getFirstNode}(l)$  and delete the node  $v$  from the list  $l$ ;
3:   evaluate the edge  $e(v'parent, v)$  in query  $q$  over the candidates  $G$ ;
4:   if No candidates in  $G$  satisfy the edge  $e$  then
5:      $U(k_1)(G) = U(k_1)(G) - \text{score}(v, q) - w_e(e)$ ;
6:     if  $U(k_1)(G) < \sigma$  then
7:       suspend the current group  $G$ ;
8:       if  $\sigma == U(k_1)(G_x)$  then
9:         Switching to probe the group  $G_x$  in the current data source  $S_{k_1}$ ;
10:        ScheEval( $l, q, G_x, U(k_1)(G_x), L(k_1)(G_x)$ );
11:       else
12:         Switching to the next data source  $S_{k_2}$  due to  $\sigma = U(k_2)$ ;
13:       else if All candidates in  $G$  satisfy the edge  $e$  then
14:          $L(k_1)(G) = L(k_1)(G) + w_e(v'parent, v)$ ;
15:         if  $L(k_1)(G) \geq \sigma$  then
16:           determineCandidates();
17:         else
18:           list  $l' = \text{sortAllChildNodes}(ch(v))$  and list  $l = \text{mergeSort}(l, l')$ ;
19:         else
20:           //{Partial candidates in  $G$  satisfy the edge  $e$ }
21:           divideGroup( $e, q, G$ ) into two groups  $G_1$  that satisfies the edge and  $G_2$  that
           doesn't and putActiveGroup( $G_2$ );
22:            $U(k_1)(G_2) = U(k_1)(G_2) - \text{score}(v, q) - w_e(v'parent, v)$ ;
23:            $L(k_1)(G_1) = L(k_1)(G_1) + w_e(v'parent, v)$ ;
24:           if  $U(k_1)(G_2) > \sigma$  then
25:              $\sigma = U(k_1)(G_2)$ ;
26:             //{The group  $G_2$  in  $k_1$  data source would be evaluated at next step.}
27:           if  $L(k_1)(G_1) \geq \sigma$  then
28:             determineCandidates();
29:           else
30:             list  $l' = \text{sortAllChildNodes}(ch(v))$  and list  $l = \text{mergeSort}(l, l')$ ;
```

in G satisfy the edge e , e.g., a subgroup G_1 of candidates satisfy while another subgroup G_2 of candidates do not. We use the function $divideGroup(e, q, G)$ to divide the group G of candidates into G_1 and G_2 . Then we compute the upper bound, lower bound for each group. For G_1 , its upper bound $U(k_1)(G_1)$ does not change, but its lower bound $L(k_1)(G_1)$ will increase. For G_2 , its upper bound $U(k_1)(G_2)$ will decrease, however its lower bound $L(k_1)(G_2)$ keeps unchanged. Obviously, we have $U(k_1)(G_1) > U(k_1)(G_2)$. Therefore, we prefer searching in group G_1 to G_2 while cache group G_2 with Function $putActiveGroup()$. If $\sigma < U(k_1)(G_2)$, we should take $U(k_1)(G_2)$ as the new threshold for the current group G_1 . And if $L(k_1)(G_1)$ is greater than or equal to the updated threshold σ , we will call for Function $determineCandidates()$ in Algorithm 3. Otherwise, a new edge need to be evaluated on the current group of candidates.

Algorithm 3 can be designed to determine the correct ones from the group if we find that a group of candidates in a data source would contain the correct answers for top- k query. There are three ways to process the candidates in G .

Algorithm 3 Function: determineCandidates()

```
1: if  $|G| = k$  then
2:   return k results while Stop searching;
3: else if  $|G| < k$  then
4:   return  $\lambda$  results and  $k = k - |G|$ ;
5:   if  $\sigma == U(k_1)(G_x)$  then
6:     Switching to probe the group  $G_x$  in the current data source  $S_{k_1}$ ;
7:     ScheEval( $l, q, G_x, U(k_1)(G_x), L(k_1)(G_x)$ );
8:   else
9:     Switching to the next data source  $S_{k_2}$  due to  $\sigma = U(k_2)$ ;
10: else
11:   list  $l' = \text{sortAllChildNodes}(ch(v))$  and list  $l = \text{mergeSort}(l, l')$ ;
12:   ScheEval( $l, q, G, U(k_1)(G), L(k_1)(G)$ );
```

(1) If $|G| = k$, the group of candidates are correct answers for top- k query and searching is terminated; (2) If $|G| < k$, the group of candidates are part of the correct answers and the value of k will be decreased by $k = k - |G|$. At next step, we would probe the previous groups G_x in the current data source S_{k_1} if we have $\sigma = U(k_1)(G_x)$ or switch to the next data source S_{k_2} if we have $\sigma = U(k_2)$ (3) Otherwise, we will expand the edges and continuously evaluate them over the group G for determining the k best ones.

Algorithm 4 ComputingScore()

input: a weighted query rooted at r

output: a query that every subtree is marked with scores

```
1: push(the root  $r$ , a stack  $S$ );
2: while the stack is not empty  $S \neq \phi$  do
3:    $v = \text{getStackTop}(S)$ ;
4:    $existEdgeScore = \text{getEdgeScore}(v)$ ;
5:   if  $ch(v) \neq \phi$  then
6:     for all  $v_c \in ch(v)$  do
7:        $newEdgeScore = \text{getEdgeScore}(v_c)$ ;
8:        $currentEdgeScore = existEdgeScore \times newEdgeScore$ ;
9:        $\text{updateEdgeScore}(v_c, currentEdgeScore)$ ;
10:    push  $v_c$  into the stack  $S$ ;
11:    ComputingScore( $v_c$ );
12:   else
13:     pop(a node, a stack  $S$ );
14:      $v_x = \text{getStackTop}(S)$ ;
15:      $xEdgeScore = \text{getEdgeScore}(v_x)$ ;
16:      $\text{updateEdgeScore}(v_x, xEdgeScore + existEdgeScore)$ ;
```

Algorithm 4 is used to mark the weight for each subtree in *depth-first search* style. For each internal node v (i.e., $ch(v) \neq \phi$), we should push it into the stack S while update its score by computing the extended edge weight between the node v and its ancestor. For each leaf node or internal node that its child nodes have been processed, we will pop the node from the stack S while update its parent's score by propagating its score to its parent. Two important functions

getEdgeScore() and *updateEdgeScore()* are used to retrieve and update the score of each node, respectively.

5 Experiments

The presented algorithms for the BT strategy are implemented in a Java prototype using JDK 1.4. B+-tree indexes are used to access the nodes in each data source. Wutka DTDparser ¹ is used to analyze the source DTDs and extract their structural information. We run our experiments on an Intel P4 3GHz PC with 512M memory.

Table 1. Designed Queries

q_1 :	<code>//item [./description /parlist]</code>
q_2 :	<code>//item [./description /parlist /mailbox /mail [./text]]</code>
q_3 :	<code>//item [./mailbox /mail /text [./keyword and ./xxx] and ./name and ./xxx]</code>

Dataset and Queries: We use XMark XML data generator ² to generate a number of data sets, of varying sizes and other data characteristics, such as the fanout (MaxRepeats) and the maximum depth, using the *auction.dtd* and changed versions by deleting some nodes. We also use the XMach-1 ³ and XMark ⁴ benchmarks, and some real XML data. The results obtained are very similar in all cases, and in the interest of space we present results only for the largest auction data set that we generated. We evaluate the presented algorithms using the set of queries shown in Table 1 where the symbol “xxx” is added as noise node that do not appear in the DTD. In our query set, we consider the structural difference between the query and the DTD, such as the edge “parlist/mailbox” does not exist in the DTD. It will be adjusted by calling for previous query relaxation. We also take into account two semantics in DTD, such as the edge “description/parlist” satisfies disjunctive semantics and the nodes “mail” and “text” satisfy optional semantics.

Test Results: In our experiments, we also implement the Naive strategy that first retrieves top k matches from each data source and then selects the k most relevant ones from the intermediate results. Our test results show that the BT scheduling strategy is faster than the Naive strategy to search top- k matches over multiple data sources. Especially, when the number of data sources or the value of top- k are large, more benefits would be gained.

Figure 4 shows that dynamic sort-based BT scheduling strategy can improve the performance more than static sort-based BT scheduling strategy, in terms of evaluation of unqualified edges for some documents where the three queries are evaluated over 5, 10, 15 and 20 number of XML documents respectively and

¹ Wutka DTD parser. <http://www.wutka.com/dtdparser.html>.

² Xmark XML data generator. <http://monetdb.cwi.nl/xml/index.html>.

³ XMach-1. <http://dbs.unileipzig.de/en/projekte/XML/XmlBenchmarking.html>.

⁴ The XML benchmark project. <http://www.xml-benchmark.org>.

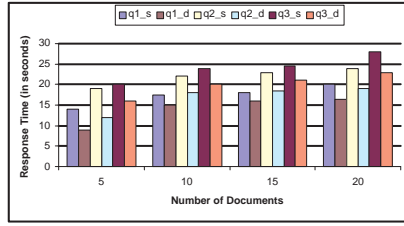


Fig. 4. Static Sort vs. Dynamic Sort

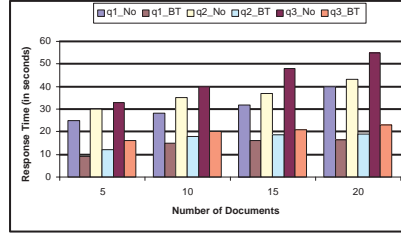


Fig. 5. No Schedule vs. BT Schedule

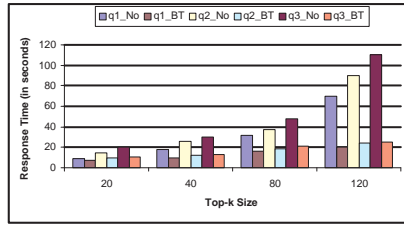


Fig. 6. Varying Top-k Size

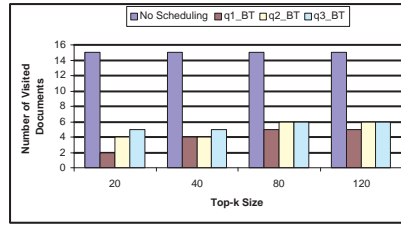


Fig. 7. Varying Top-k Size

top- k is set as 80. In the following paragraph, we mainly choose the experiments about dynamic sort-based BT scheduling strategy in different conditions. Figure 5 shows BT scheduling strategy outperforms Naive strategy greatly. Two appealing features can be obtained: one the one hand, the larger the number of XML documents to be searched, the more benefits the BT scheduling strategy can gain; on the other hand, the BT scheduling strategy has good scalability, i.e., the increasing trends will become slow after the number of XML documents is relatively large. For example, the trends evaluating the three queries over 10, 15, and 20 documents are much slower than the trend between 5 and 10 documents. This is because a larger number of documents have higher possibility to contain noise documents. Figure 6 and Figure 7 illustrate the performance when we vary the size of top- k value across 20, 40, 80 and 120 where all the three queries are evaluated over 15 documents. From Figure 6, the both strategies can gain similar time cost when the top- k value is small. But the gap between the BT scheduling strategy and the Naive strategy will become much larger when top- k is 120. In addition, Figure 7 shows the number of documents that need to be visited in order to answer the three queries. Obviously, for the Naive strategy, all the documents require to be checked. However, for the BT scheduling strategy, only part of the documents are visited during query evaluation. Furthermore, the same number of documents are traversed for q_2 and q_3 when top- k is 80 or 120. This is because some elements in query like *mailbox* are distributed in the same documents when we design our data sets.

6 Related Work

Top- k query processing has been extensively studied in the literature. In relational databases, existing work has focused on extending the evaluation of SQL queries for top- k processing. None of these works follows an adaptive query evaluation strategy. Carey and Kossmann [1] optimize top- k queries when the scoring is done through a traditional SQL order by clause, by limiting the cardinality of intermediate results. Other works [2, 3] use statistical information to map top- k queries into selection predicates which may require restarting query evaluation when the number of answers is less than k . Over multiple repositories in a mediator setting, Fagin et al. propose a family of algorithms [4–6], which can evaluate top- k queries that involve several independent subsystems, each producing scores that are combined using arbitrary monotonic aggregation functions. These algorithms are sequential in that they completely process one tuple before moving to the next tuple. The Upper [7], MPro [8] and TPUT [9] algorithms show that interleaving probes on tuples results in substantial savings in execution time. In addition, Upper [7] uses an adaptive per-tuple probe scheduling strategy, which results in additional savings in execution time when probing time dominates query execution time.

Recently in [10, 11], top- k keyword queries for XML have been studied via proposals extending the work of Fagin et al., [5, 18] to deal with a bag of single path queries. Adaptivity and approximation of XML queries are not addressed in their work. Marian et. al. in [12] explore an adaptive top- k query processing strategy in XML, which permits different query plans for different partial matches and maximizes the best scores. Based on the intermediate results, the irrelevant answers for the top- k query may be pruned as early as possible. But they do not discuss top- k query evaluation over a larger number of different data sources. Furthermore, the correct results can not be determined until all candidates are evaluated.

Different from previous work, we study top- k search over a large number of XML data sources and focus on issues such as exploring the BT-based scheduling strategy that not only skips many data sources without probing for top- k evaluation, but also prunes the unqualified distinguished nodes of each visited data source. Additionally, we also deploy an adaptive query relaxation strategy to filter out some unqualified edges in the query for some data sources based on schema information, which can further improve query evaluation efficiency.

7 Conclusions

The primary contribution of this paper lies in the BT-based scheduling strategy that we proposed. Based on the strategy, we are able to avoid the evaluation of big number of data sources, and prune unqualified results in the visited data sources. Besides, the strategy also satisfies monotonic feature for returning qualified results. The experimental results demonstrated the BT scheduling strategy can gain more benefits when the value of top- k and the number of data sources

are large. Additionally, the results also shown that the BT scheduling strategy can skip most of data sources during query evaluation. Therefore, it is appropriate and practical for the BT scheduling strategy to be applied to XML searching system.

Acknowledgments. This work was supported by grants from the Australian Research Council Discovery Project (DP0559202) and the Research Grant Council of the Hong Kong Special Administrative Region, China (CUHK418205).

References

1. Michael J. Carey and Donald Kossmann. On saying "enough already!" in sql. In *SIGMOD Conference*, pages 219–230, 1997.
2. Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.
3. Chung-Min Chen and Yibei Ling. A sampling-based estimator for top-k query. In *ICDE*, pages 617–627, 2002.
4. Ronald Fagin. Fuzzy queries in multimedia database systems. In *PODS*, pages 1–10. ACM Press, 1998.
5. Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, New York, NY, USA, 2001. ACM Press.
6. Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.
7. Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.
8. Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.
9. Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.
10. Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for topx search. In *VLDB*, pages 625–636, 2005.
11. Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD Conference*, pages 779–790, 2004.
12. Amélie Marian, Sihem Amer-Yahia, Nick Koudas, and Divesh Srivastava. Adaptive processing of top-k queries in xml. In *ICDE*, pages 162–173, 2005.
13. Torsten Schlieder. Schema-driven evaluation of approximate tree-pattern queries. In *EDBT*, pages 514–532, 2002.
14. Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. Tree pattern relaxation. In *EDBT*, pages 496–513, 2002.
15. Laks V. S. Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng (Jessica) Zhao. On testing satisfiability of tree pattern queries. In *VLDB*, pages 120–131, 2004.
16. Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–, 2002.
17. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
18. Ronald Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.