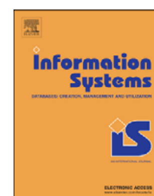
Information Systems 35 (2010) 688–707

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys



Adaptive relaxation for querying heterogeneous XML data sources

Chengfei Liu [a,*], Jianxin Li [a], Jeffrey Xu Yu [b], Rui Zhou [a]

[a] *Swinburne University of Technology, Australia*
[b] *Chinese University of Hong Kong, China*

ARTICLE INFO

Article history:
Received 17 March 2009
Received in revised form
10 February 2010
Accepted 17 February 2010
Recommended by: L. Wong

Keywords:
XML relaxation
Top-*k* query
XML schema

ABSTRACT

Searching XML data with a structured XML query can improve the precision of results compared with a keyword search. However, the structural heterogeneity of the large number of XML data sources makes it difficult to answer the structured query exactly. As such, query relaxation is necessary. Previous work on XML query relaxation poses the problem of unnecessary computation of a big number of unqualified relaxed queries. To address this issue, we propose an adaptive relaxation approach which relaxes a query against different data sources differently based on their conformed schemas. In this paper, we present a set of techniques that supports this approach, which includes schema-aware relaxation rules for relaxing a query adaptively, a weighted model for ranking relaxed queries, and algorithms for adaptive relaxation of a query and top-*k* query processing. We discuss results from a comprehensive set of experiments that show the effectiveness and the efficiency of our approach.

© 2010 Elsevier B.V. All rights reserved.

1. Introduction

As XML becomes the standard for representing web data, people are now publishing a large volume of data on the internet using XML for various purposes. For example, universities publish their course and research data for attracting students; travel and real estate agents publish their flight and property data for advertisement; stock brokers and car dealers publish stock and car information for online business; public service providers publish data such as tourist attractions and publications for providing information. As such, there is an increasing need to search and query XML data. Compared with a keyword search, a structured XML query allows a user to formulate the search requests more precisely. However, the structural heterogeneity of the potentially large number of XML data sources makes it difficult to answer a structured query exactly. The loosely coupled nature of the data sources also makes it inapplicable for deploying the traditional federated database approach for integrating the XML data sources by defining a global schema. It would be ideal that a query could be smartly relaxed then be answered according to the data sources against which the query is issued.

Amer-Yahia et al. [1,2] proposed a framework FleXPath for relaxing XML tree pattern queries (TPQs). Given a TPQ *q*, the closure of the structural and value-based predicates in *q* is first inferred and then is used to generate relaxed queries. The set of generated queries, including the one that includes the root of *q*, contains all possible relaxed queries. However, the relaxation process is basically blind and wild and the number of relaxed queries could be big. For a large number of heterogeneous XML data sources, many of the generated relaxed queries could be unqualified and will result in unnecessary cost of either computing or testing them.

As an example, we may issue a query against XML data sources maintained in all Australian universities for searching those departments that have a group running project with a name containing "xml" and having publications with a title containing "query relaxation". As the

* Corresponding author. Tel.: +61 3 92145035.
E-mail addresses: cliu@swin.edu.au (C. Liu), jianxinli@swin.edu.au (J. Li), yu@se.cuhk.edu.hk (J.X. Yu), rzhou@swin.edu.au (R. Zhou).

0306-4379/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2010.02.002

number of universities is large and their data source structures may vary, users normally formulate their queries against a *domain* schema according to the common understanding of a university. Here a domain schema bears similarity to a global schema. However, unlike a federated database, such a domain schema and its mapping from data source schemas may not be physically defined. So we cannot borrow the global-to-local query rewriting techniques in the context of data integration [4,5]. Fig. 1 shows the query q represented as a TPQ and it reflects the user's structural and value-based search requirements. Solely based on the query itself, FleXPath may need to consider 2^5 options, each could be a relaxed query that may be executed or tested against the university data sources. Some generated relaxed queries may be either too blind for some data sources thus return zero answers or too wild thus return answers that are far from what a user is expected. For example, the partial structures of the data source s_1 and s_2 for two universities are shown in schema d_1 in Fig. 2 and schema d_2 in Fig. 3, respectively. Obviously, the query itself will not return any result, and many relaxed queries will be generated by FleXPath from 2^5 options and then be evaluated or tested for both data sources. For example, among the relaxed queries, some of the useless relaxed queries for s_2 are listed in Fig. 4. Actually, q_2 and q_3 in Fig. 4 are also useless for s_1.

To deal with this problem, we propose an *adaptive query relaxation* (*AQR*) approach, which relaxes a query adaptively to each XML data source according to its conformed schema. Hence each relaxed query will be guaranteed to agree with the structural constraints imposed by the conformed schema of the data source, and as a result, has higher probability of generating answers compared with FleXPath. For example, for schema d_1 in Fig. 2 and schema d_2 in Fig. 3, the relaxed queries generated by AQR are shown in Figs. 5 and 6, respectively.

AQR avoids blind relaxation. Each generated relaxed query for an XML data source is specific to the data source.

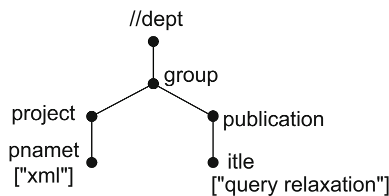


Fig. 1. A query q.

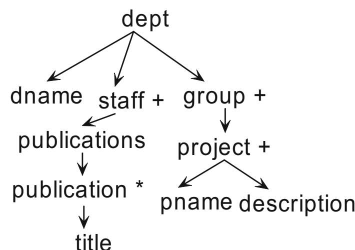


Fig. 2. Schema d_1 for s_1.

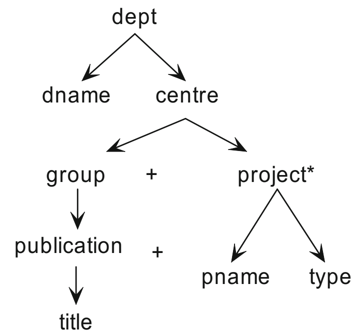


Fig. 3. Schema d_2 for s_2.

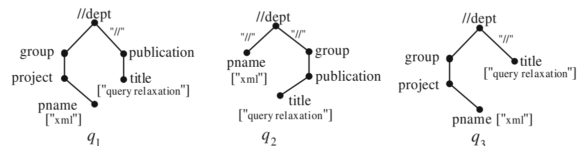


Fig. 4. Relaxed queries of FleXPath.

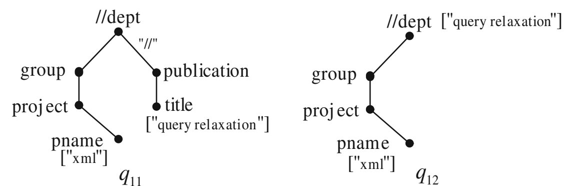


Fig. 5. Relaxed queries of AQR for s_1.

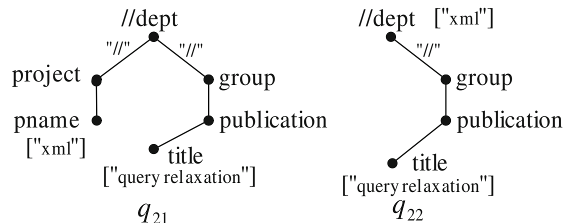


Fig. 6. Relaxed queries of AQR for s_2.

In other words, a relaxed query that does not satisfy the structural constraints imposed by the conformed schema will not be generated. This is similar to semantic query optimisation where a query that contradicts with an integrity constraint defined in the underlying schema may not need to be evaluated. For example, for data source s_2, query q_1 in Fig. 4 is useless and will not return any result because the edge between *group* and *project* in q_1 does not match d_2.

AQR also avoids wild relaxation. No unnecessary relaxation is needed because of the requirement that a data source has to conform with its schema. For example, the *-node *project* in d_2 implies the co-existence of *project* and *pname*. As such, for s_2, query q_2 in Fig. 4 is too wild compared with query q_{21} in Fig. 6. In other words, after q_{21}

is generated and evaluated, the time spent on generating and computing q_2 is unnecessary because no new result will be returned.

As a large number of data sources may be evaluated and the relaxed queries generated for these data sources may be different, it is desirable to first execute the relaxed query that is closest to the original query such that the most relevant results can be returned first. In other words, the most relevant results are returned from the closest relaxed query in our system. This is especially important to evaluate a top-k query [3,6]. For example, query q_{11} in Fig. 5 is the closest query to the original query q in all the generated relaxed queries in Figs. 5 and 6. In AQR, a top-k query is processed by incrementally evaluating the relaxed queries in their ranking order. Instead of ranking returned results, we rank the relaxed queries. To compare how much a relaxed query is close to the original query, we propose a penalty based ranking model to measure the difference between a relaxed query and the original query in AQR. For example, if the penalty for relaxing "/" to "//" is 0.1, we can compute the penalties of q_{11} and q_{12} in Fig. 5 as 2 and 5, and the penalties of q_{21} and q_{22} in Fig. 6 as 2.3 and 5.3. So q_{11} is the least penalized query or the closest query to the original query q. The details for computing the penalties can be found in Section 4.

To improve the accuracy and relevancy of the results, we allow a user to specify weights on edges of a query and thus incorporate a weight into the ranking model. If the relationship between two nodes is less important than others, a smaller weight may be specified compared with the maximum weight 1. Our ranking model is based on the weight set on the original query and the penalty derived for a relaxed query. The ranking score of a relaxed query is calculated as the difference between the query weight for the original query and the penalty of the relaxed query. For example, when the weights for all edges are set to 1, the weight of the original query is 11. We know that the penalty for q_{11}, q_{12}, q_{21} and q_{22} are 2, 5, 2.3 and 5.3, so the scores for them are 9, 6, 8.7 and 5.7, respectively, i.e., the ranking list is $[q_{11}, q_{21}, q_{12}, q_{22}]$. If a user thinks that the relationship for a project under a group is less important and likes to decrease the weight of the edge between group and project in q to 0.5 while keeping other edges as 1, the ranking list will be changed to $[q_{21}, q_{11}, q_{22}, q_{12}]$. The details for computing these ranking scores can also be found in Section 4.

In case the schema is not available for a data source, its structural information can be generated dynamically with data summarization tools [7,8]. In this paper, without loss of generality, we take DTD as the schema of XML data.

In summary, we claim the following contributions in this paper:

- We propose and formalize the adaptive XML query relaxation problem w.r.t. different DTDs and devise a set of schema-aware relaxation rules.
- We develop a weight modification and penalty evaluation model to assess to what extent the original query is relaxed.
- We design a set of algorithms to describe how the rules and penalty model are leveraged in the process of relaxing queries.
- We provide a scheduling strategy to incrementally evaluate relaxed queries across multiple data sources.
- We run extensive experiments on XMark Benchmark to justify the efficiency and validity of our adaptive relaxation approach.

The rest of the paper is organized as follows. We give an overview of our AQR in Section 2. Section 3 discusses the relaxation rules in detail. Section 4 provides our weight modification and penalty evaluation models. The detailed descriptions of our adaptive relaxation algorithm are provided in Section 5. Section 6 proposes strategies to schedule the relaxed queries during query evaluation. We present the results of extensive experiments in Section 7. A brief survey of related work and the conclusions of this work are given in Sections 8 and 9, respectively.

2. Overview

The goal of query relaxation is to relax the query constraints such that approximate answers can be returned if the original query returns no answer or not enough answers. This is especially useful when we query a big number of heterogeneous XML data sources using a single structured XML query. Given a TPQ q, FleXPath generates relaxed queries by enumerating all possible combinations starting from q itself to the root of q, thus resulting in large number of relaxed queries. Among these queries, some of them do not even match the structure of any XML data source so return no result; some of them may return result from some data sources, but return no results from others. Keep this in mind, our AQR approach does not enumerate the possible combinations for generating relaxed queries. Instead, we use structural information of data sources such as DTDs for customizing the generation of relaxed queries for different data sources. To achieve this, we design a set of adaptive relaxation rules to guide the generation of relaxed queries for different data sources based on their conformed DTDs. To evaluate a top-k query incrementally, we choose the closest or least penalized relaxed query first for execution. As such, we devise a comprehensive ranking model based on penalties. To improve the accuracy of the returned answers, we also allow users to specify parameters such as edge weights and coefficient. We see this as an important alternative to users because users are able to express customised requirements on their queries in terms of their preferences. We incorporate this support into the ranking model and also extend the definition of a tree pattern query.

Definition 1 (*Weighted tree pattern query, WTPQ*). A weighted tree pattern query q is defined as a tree $T(V, E, r, w)$, where V is a finite set of nodes. Each $v \in V$ is uniquely identified and may have search requirement of a *term t* denoted as $\tau(t)$. *tagname*(v) specifies the tag name of v. E consists of two kinds of edges called *pc*-edges and *ad*-edges, corresponding to the child and descendant axes of XPath. r is the root node that is a distinguished node in V corresponding to the *answer node*. For any $e(v_1, v_2) \in E$,

$w(v_1, v_2) \in (0, 1]$ marks a weight for e with 1 as the default value. For any $v \in V$, $ch(v)$ gives the set of child nodes of v, $p(v)$ specifies the parent node of v. pc, ad specify parent–child and ancestor–descendant relationships between a pair of nodes (v_1, v_2), respectively. $pc(v_1, v_2)$ if $e(v_1, v_2) \in E$ and e is an pc-edge. $ad(v_1, v_2)$ if v_1 is an ancestor node of v_2, or $e(v_1, v_2) \in E$ and e is an ad-edge.

For example, Figs. 1 and 7 represent the same tree pattern query with different edge weights. The former holds the default weight 1.0 for each edge while the latter reduces the weights of *dept/group* and *group/publication* to 0.8 and 0.5, respectively.

In AQR, a DTD is defined as a directed acyclic graph (DAG) with a single root (for the document element). We will extend the definition to allow recursive DTD definition when we discuss the recursive relationship relaxation.

Definition 2 (*DTD graph*). A DTD d is defined as a directed acyclic graph $G(V_d, A_d, r_d)$, where V_d is a finite set of nodes. Each node $v \in V_d$ specifies an element or an attribute in XML documents that conform to d and is uniquely identified. $tagname(v)$ specifies the tag name of v. $ch(v)$ gives the set of child nodes of v. $p(v)$ yields the set of parent nodes of v. $opt(v)$ specifies if v is optional under $p(v)$. This corresponds to the cardinality requirement of a node. It will be set to *true* for "*" and "?", and to *false* for "+" and "1". $bar(v)$ corresponds to "|" in DTD and specifies if v takes only one of the child nodes at a time. A_d is a finite set of arcs. $pc(v_1, v_2)$ if $e(v_1, v_2) \in A_d$. $ad(v_1, v_2)$ if a path exists from v_1 to v_2. pc, ad specify parent–child and ancestor–descendant relationships between a pair of nodes (v_1, v_2), respectively. r_d specifies the root node, $r_d \in V_d$ and $p(r_d) = \phi$.

For example, Fig. 2 shows a DTD graph d_1, where *dept* is the root node, *dept/staff* is a parent–child relationship, and "+" on node *staff* means that one department contains at least one staff.

Now, we formulate our adaptive query relaxation problem as follows: Given a WTPQ $q = T(V, E, r, w)$ and a set of DTDs $d_1, d_2, \ldots, d_n$, we would like to find a set of relaxed queries $Q = Q_1 \cup Q_2 \cup \cdots \cup Q_n$, where Q_i is the set of queries conforming to d_i. To determine Q_i, we firstly relax q into a relaxed query q_i' that preserves maximum query requirements of q w.r.t. d_i. Based on query requirements, q_i' may be further relaxed into a set of queries Q_i according to the cardinality information, such as "*" in d_i.

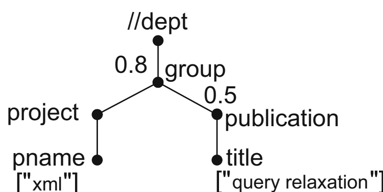


Fig. 7. A query q with weight modification.

3. Adaptive relaxation rules

As discussed above, AQR relaxes a query for an XML data source based on its conformed DTD. Basically, AQR avoids blind relaxation by filtering out those query nodes that do not appear in the DTD and adjusting the node relationships if they do not match the DTD; AQR also avoids wild relaxation by preserving the query requirements which are definitely satisfied by the DTD. Before we introduce the set of adaptive relaxation rules for these purposes, we need the following definitions.

Definition 3 (*Corresponding node*). Let a WTPQ $q = T(V, E, r, w)$ and a DTD $d = G(V_d, A_d, r_d)$, $v' \in V_d$ is called the *corresponding node* of a node $v \in V$ if $tagname(v') = tagname(v)$. A node in V may not always have a corresponding node.

Definition 4 (*Consistent corresponding node*). Let a WTPQ $q = T(V, E, r, w)$, a DTD $d = G(V_d, A_d, r_d)$, and $r', v' \in V_d$ are the corresponding nodes of $r, v \in V$, respectively, v' is called the *consistent corresponding node* of a node v if either $pc(r', v')$ or $ad(r', v')$ holds. The corresponding node r' of the root node r is always consistent.

Consider the query and the DTD in Fig. 8. According to Definitions 3 and 4, six nodes *dept*, *group*, *project*, *pname*, *publication* and *title* in the query have their corresponding nodes in the DTD, among them four nodes *dept*, *group*, *project*, *pname* are the consistent corresponding nodes. The query node *year* has no corresponding node in the DTD.

For a WTPQ $q = T(V, E, r, w)$ and a DTD d, the minimal requirement for the relaxed query q' of q w.r.t. d is that the root node r of q keeps in q', i.e., r has a corresponding node r' in d. For any other node $v \in V$, we need to check if it has a consistent corresponding node in d.

3.1. Ontology relaxation

Before we discuss ontology relaxation, we assume that a tagname uniquely represents a concept and different tagnames representing same concept can be renamed as a single tagname, e.g., a worker can be renamed as an employee. With this assumption, a tagname can uniquely identify a node in a DTD. This means that a corresponding node defined in Definition 3 is unique if exists.

For each $v \in V$, if its corresponding node $v' \in V_d$ does not exist, we can search, say from WordNet, to see if there exists a superclass (hypernym) of $tagname(v)$ that matches the *tagname* of a node $v'_{super} \in V_d$ and v'_{super} is under the

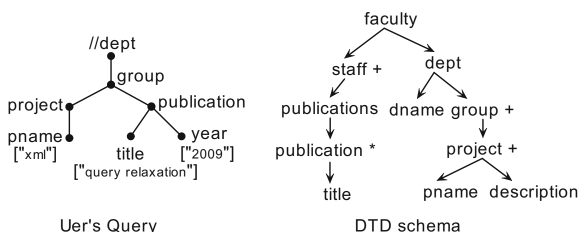


Fig. 8. Illustration of corresponding node and consistent corresponding node.

corresponding node of r. If so, the tagname of v is renamed as that of v'_{super} in the relaxed query q' for d. We call v'_{super} a *relaxed consistent corresponding node* of v. A fixed penalty applies for this kind of relaxation.

A query q can be relaxed to q' for d if either the corresponding node or the relaxed corresponding node of the root node r exists in V_d.

3.2. Node relaxation

When a node $v \in V$ cannot find a consistent corresponding node or a relaxed consistent corresponding node in V_d, v needs to be deleted in the relaxed query q' for d. The following two situations are treated differently.

Case 1: If the node is a *leaf* node, i.e., $ch(v) = \phi$, we can directly delete the node v and the edge $e(p(v), v)$ in the relaxed query q'.

Case 2: If the node is an *internal* node, i.e., $ch(v) \neq \phi$ and $v \neq r$, we first delete the node v and the edge $e(p(v), v)$, then for each $v_i \in ch(v)$, we replace the edge $e(v, v_i)$ with an *ad*-edge $e'(p(v), v_i)$ in the relaxed query q' regardless of whether e is a *pc*-edge or an *ad*-edge.

Since a node (an element or attribute) that does not exist in a DTD will not appear in the corresponding data sources that conform to the DTD, the adoption of the *node relaxation* rule can delete the node in advance, rather than leave it in the query which will be evaluated later on these data sources with no result. Consequently, unqualified relaxed queries will not be generated in AQR. That will reduce the number of relaxed queries generated and improve the performance of relaxed query evaluation. The deletion of a node in the query may affect the original query requirements. To mark this change in the relaxed query, a structural penalty applies depending on the importance of the relationships between the node and other nodes. This will be discussed in detail in Section 4.

3.3. Term relaxation

Assume a node v contains search requirements for terms $t_1, \ldots, t_m$, i.e., $v[\tau(t_1)$ *and* $\ldots \tau(t_m)]$, and its parent node $v_p[\tau(t_{p1})$ *and* $\ldots$ *and* $\tau(t_{pn})]$. If v is deleted, its terms will be merged to its parent resulting $v_p[\tau(t_{p1})$ *and* $\ldots$ *and* $\tau(t_{pn})$ *and* $\tau(t_1)$ *and* $\ldots$ *and* $\tau(t_m)]$, where t_i $(1 \le i \le m)$ does not match any of t_{pj} $(1 \le j \le n)$.

While query requirements on XML structure are important for finding accurate information, search requirements on terms are fundamental to most queries from users. When a node that contains a term search request is deleted, we can apply the above term relaxation rule to keep the term search request by promoting it to its parent using operator "*and*", rather than deleting it together with the node. The semantics of this term promotion widens the search scope from a child node to its parent node when the child node does not appear in the DTD.

Since penalty has been applied to a node relaxation, no extra penalty applies to a term relaxation.

3.4. Inconsistent edge relaxation

The inconsistent appearance of nodes between a query q and a DTD d is handled by a node relaxation rule. Now we consider each edge $e(v_1, v_2) \in E$ of q. In q, a user specifies either $pc(v_1, v_2)$ (*pc*-edge) or $ad(v_1, v_2)$ (*ad*-edge). In the relaxed query q' of q, we try to keep this relationship as close as possible. However, even though there is no such close relationship between v'_1 and v'_2 in d where v'_1 and v'_2 are consistent corresponding nodes of v_1 and v_2, it is better to keep v_1 and v_2 in q' than just delete them, which can keep the maximum query requirements of the original query. So we have the following definition.

Definition 5 (*eSibling*). Given a pair of nodes (v_1, v_2), if neither $pc(v_1, v_2)$ or $ad(v_1, v_2)$, nor $pc(v_2, v_1)$ or $ad(v_2, v_1)$ holds, v_1 and v_2 are said to satisfy an extended sibling relationship and is denoted as $eSibling(v_1, v_2)$.

For example, nodes *group* and *publication* in Fig. 2 hold the eSibling relationship.

Note, an *eSibling* relationship permits that v_1 and v_2 appear at different levels of a tree or DTD DAG. Unlike *pc* or *ad* relationships, we have $eSibling(v_1, v_2) = eSibling(v_2, v_1)$.

Now we consider an edge $e(v_1, v_2) \in E$ of q, and assume that the consistent corresponding pair of nodes (v'_1, v'_2) of (v_1, v_2) exist in DTD d. If e is a *pc*-edge and $pc(v'_1, v'_2)$ also holds, or e is an *ad*-edge and either $pc(v'_1, v'_2)$ or $ad(v'_1, v'_2)$ holds in d, we can keep e as it is in q'; otherwise, we need to handle the following two situations.

Case 1: Relaxing a *pc-edge* into an *ad*-edge. If e is a *pc*-edge and only $ad(v'_1, v'_2)$ (not $pc(v'_1, v'_2)$) holds in d, e needs to be relaxed from a *pc*-edge to an *ad*-edge in q'.

Case 2: Relaxing a *pc*-edge or an *ad*-edge into an *eSibling* relationship. In d, if neither $pc(v'_1, v'_2)$ nor $ad(v'_1, v'_2)$ holds (i.e., $eSibling(v'_1, v'_2)$ or $pc(v'_2, v'_1)$ or $ad(v'_2, v'_1)$ holds), e will not appear in q', instead, a new edge will be created to make v_1 and v_2 to have $eSibling(v_1, v_2)$.

For Case 2, we need to know how to create the new edge, i.e., to connect v_2 under a *common ancestor* of v_1 and v_2 based on the positions of v'_1 and v'_2 in d. Again, we try to keep the relationship as close as possible in q' with respect to d so a *nearest* common ancestor is required. The following definition serves for this purpose.

Definition 6 (*Nearest common ancestor, NCA*). Let a WTPQ $q = T(V, E, r, w)$, a DTD $d = G(V_d, A_d, r_d)$, and $v_1, v_2 \in V$ are a pair of nodes in q. $NCA(v_1, v_2, q, d)$ is defined as a node in V, denoted as v_{nca}, which satisfies all the following conditions: (1) $pc(v_{nca}, v_1)$ or $ad(v_{nca}, v_1)$, and $pc(v_{nca}, v_2)$ or $ad(v_{nca}, v_2)$ hold. (2) v_1, v_2 and v_{nca} all have their consistent corresponding nodes v'_1, v'_2 and v'_{nca} in d. (3) $pc(v'_{nca}, v'_1)$ or $ad(v'_{nca}, v'_1)$, and $pc(v'_{nca}, v'_2)$ or $ad(v'_{nca}, v'_2)$ hold. (4) $\neg\exists v_x \in V$ such that v_x satisfies the above conditions, and $pc(v'_{nca}, v'_x)$ or $ad(v'_{nca}, v'_x)$ holds.

For example, given the query q in Fig. 1, DTD d_1 in Fig. 2 and DTD d_2 in Fig. 3, $NCA(group, project, q, d_1) = group$ while $NCA(group, project, q, d_2) = dept$.

Theorem 1 (*Existence of NCA*). *Let a WTPQ* $q = T(V, E, r, w)$, *a DTD* $d = G(V_d, A_d, r_d)$, *and* $e(v_1, v_2) \in E$ *in q*. $NCA(v_1, v_2, q, d)$

exists if and only if $\exists v_1', v_2', r' \in V_d$ such that they are the consistent corresponding nodes of $v_1, v_2, r \in V$, respectively.

Proof of " ← ": Since v_1' and v_2' are the consistent corresponding nodes of v_1 and v_2, respectively, by Definition 4, we know that both v_1' and v_2' are under the consistent corresponding node r' of r. In other words, r' is a common ancestor of v_1' and v_2'. This guarantees that the NCA v_{nca} exists with the root node r serves as a guard.

Proof of " → ": Since v_{nca} exists, by Definition 6, we know that the consistent corresponding nodes v_1' and v_2' exist for v_1 and v_2, respectively, with the consistent corresponding node r' of r as the pre-condition.

The algorithm for finding an NCA v_{nca} of (v_1, v_2) will be introduced in Section 5. Now we can create a new *ad-edge* $e_n(v_{nca}, v_2)$ in q' to replace $e(v_1, v_2)$. Furthermore, to guarantee relaxation precision, we also move the child nodes of v_2 adaptively. For each node $v_i \in ch(v_2)$, if its consistent corresponding node v_i' exists, and either $pc(v_2', v_i')$ or $ad(v_2', v_i')$ holds in d, no change is required to the edge $e_i(v_2, v_i)$; otherwise, $e_i(v_2, v_i)$ will be deleted and a new *ad-edge* $e_i(v_1, v_i)$ will be created.

We take the WTPQ q and DTD d shown in Fig. 9 as an example. q needs to be relaxed using the inconsistent edge relaxation rule (Case 2) when v_2 is considered because the inconsistent relationship for (v_1, v_2) in q and d. Before moving node v_2 to put under node v—the NCA of v_1 and v_2, we first check the relationships between v_2 and v_2's child nodes v_{21} and v_{22}. Based on $pc(v_1', v_{21}')$ and $pc(v_2', v_{22}')$, v_{21} will be connected to v_1 and v_{22} will follow v_2, resulting the relaxed query q' shown in Fig. 9.

Different penalties apply to Cases 1 and 2 of the inconsistent edge relaxation rule. Detailed penalty computation will be discussed in Section 4.

3.5. Recursive relationship relaxation

In Definition 2, a DTD is defined as a DAG. When a DTD includes some recursively defined elements, cycles may appear in the graph. Fortunately, cycles can be efficiently detected [9]. For the previous rules work properly, we can treat a cycle as a node for simplicity. However, when a user's query involves nodes in a cycle, proper processing is required. When a query path including nodes that appear in some cycles, we find that two patterns can be used for different treatments. One is *non-repetitive* while the other is *repetitive*. A node in a cycle only appears once in a non-repetitive pattern while more than one in a repetitive one. The difference between these two patterns is that users expect recursively defined elements in source DTDs in the

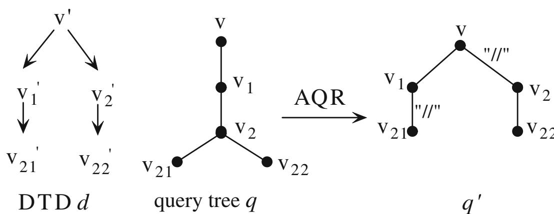


Fig. 9. Inconsistent relaxation.

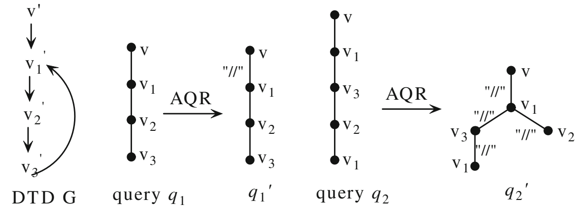


Fig. 10. Recursive relaxation.

latter while not in the former. In either case, we call the first node that appear in both a cycle and the query as a *recursion start node* and the last node that connects to a recursion start node a *recursion entry node*. For example, v_1 and v are the recursion start and entry nodes, respectively, in both queries q_1 and q_2 shown in Fig. 10.

Case 1: Consider a non-repetitive WTPQ q where the nodes in a cycle appear in the same sequence as that in the recursively defined DTD d. Let v_0 and v_1 be the recursion entry node and start node, respectively. If the edge $e(v_0, v_1)$ is a *pc-edge*, based on the assumption that the user is not aware of a recursively defined source DTD, $e(v_0, v_1)$ is relaxed to an *ad-edge* in the relaxed query q'. For example, the non-repetitive query q_1 is relaxed as q_1' in Fig. 10.

Case 2: Consider a repetitive WTPQ q where the nodes in a cycle appear in the same sequence as that in the DTD d but d does not include the corresponding cycle. We apply the node relaxation rule to retain only a single occurrence for those nodes that appear multiple times. If there are different term search conditions on each occurrence of the node, they are merged as the search conditions of the single occurrence of the node. This relaxation allows q to be evaluated on the sources that conform to d.

Case 3: Consider a repetitive WTPQ q where the nodes in a cycle appear in the same sequence as that in the recursively defined DTD d. Let v_0 and v_1 be the recursion entry node and the recursion start node, respectively. Assume $v_1,\ldots,v_n,\ldots,v_1,\ldots,v_k$ $(k \le n)$ the repetitive pattern, we relax the edge $e(v_0, v_1)$ the same as Case 1 and shorten the repetitive pattern to $v_1,\ldots,v_n, v_1,\ldots,v_k$ $(k \le n)$.

Case 4: Consider either a non-repetitive or repetitive WTPQ q where the nodes in a cycle appear in a different sequence from that in the recursively defined DTD d. We can delete the inconsistent nodes using a node relaxation rule. Similarly we can adjust the inconsistent edges using the inconsistent edge relaxation rule with the difference that the recursion start node takes the role of the NCA node. For example, the repetitive query q_2 with the inconsistent node sequence compared with the DTD cycle is relaxed as q_2' in Fig. 10.

Penalty allocations of the above relaxation rules are similar to that of non-recursive relaxation rules.

4. Weight and penalty

In order to improve the precision of a user specified query, we allow users to assign weights to edges in the query to show their preferences for different paths. Surely, the default weight 1 will be taken if users do not have preferences. The weight information will serve as a

foundation of associating each relaxed query with a reasonable penalty. Obviously, a less modified query with a low penalty is supposed to capture the user's original query aim more accurately.

4.1. Weight model

Weights are assigned on edges in users' queries, as defined in Definition 1. With a weight specified on an edge $e(v_1,v_2)$, users can specify how close v_1 and v_2 are associated with each other. However, relaxing a query may call for structural changes in the query tree, weighted relationships between only adjacent nodes are inadequate to rectify edge weights and determine the penalty after a relaxation step, especially in node relaxation and inconsistent edge relaxation. For instance, when deleting a node v_1, new edge weight on $e(p(v_1),v_2)$ between v_1's parent $p(v_1)$ and v_1's child $v_2 \in ch(v_1)$ should be determined; similarly when moving a node v_2 under a new node $v_{nca}=NCA(v_1,v_2,q,d)$, the relationship between v_{nca} and v_2 needs to be found out to determine the new edge weight for $e(v_{nca},v_2)$. To this end, 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. If $eSibling(v_i,v_j)$ holds, the extended edge weight for (v_i, v_j) is 0.

Definition 7 (*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 $w_1,w_2,\ldots,w_n$ 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, $w_e(v_i,v_j)=0$. For convenience, define $w_e(v_i,v_i)=0$.

For example, the relaxed query q_{11} in Fig. 5 of the query q in Fig. 7 contains an *ad* edge *dept//publication*. Its extended edge weight can be computed as $0.8 \times 0.5 = 0.4$.

According to the above definition, 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. These two aspects reflect our common understandings about queries. With the concept of an extended edge weight, we can easily determine the weight between two nodes that will be connected by a new *ad*-edge. Extended edge weights can be computed on the fly, or be calculated out beforehand, and then maintained dynamically.

Definition 8 (*Query weight*). The whole weight of a WTPQ q, *query weight*, denoted as $w_q(q)$ is constructed by summing all extended edge weights:

$$w_q(q) = \sum_{\forall v_i,v_j \in V, ap(v_i,v_j)} w_e(v_i,v_j)$$

where $ap(v_i, v_j)$ means either $ad(v_i, v_j)$ or $pc(v_i, v_j)$.

Query weight contains every extended edge weight, because the importance of a node in the query tree is reflected by the relationships between the node with all other nodes, not only with its parent node. Furthermore, it is obvious that query weight should be decreased after each relaxation step, since any relaxation that increases the value will be considered to add in additional relationships, and should not be carried out. This common sense is also implicitly reflected in our model.

For example, the query weight of the relaxed query q_{11} in Fig. 5 of the query q in Fig. 7 can be computed as $w_q(q_{11}) = w_e(\text{dept, group}) + w_e(\text{dept, project}) + \cdots + w_e(\text{group, project}) + \cdots + w_e(\text{project, pname}) + w_e(\text{publication, title}) = 8.2$.

4.2. Penalty evaluation

A penalty needs to be determined when a query is relaxed into a new form, accompanied with possible weight modification. There are mainly three basic operations when utilizing rules to relax user's queries: (a) deleting a node, (b) relaxing a *pc*-edge into an *ad*-edge, and (c) moving a node. For example, (a) is used in node relaxation while all the three operations may be used in the recursive relationship relaxation.

Deleting a node: Let the deleted node be v, its parent node and one of the child nodes (if exists) be $p(v)$ and $v_c \in ch(v)$. In this case, the weight of the new *ad*-edge established between $p(v)$ and v_c will be assigned with their extended edge weight $w_e(p(v),v_c)$, i.e. $w(p(v),v_c) = w_e(p(v),v_c) = w(p(v),v) \cdot w(v,v_c)$. Moreover, as the relationships between v and all the other nodes in the query disappear after this operation, the penalty is calculated as

$$f_1 = \sum_{\forall v_i \in V, v \neq v_i} w_e(v,v_i) \qquad (1)$$

After weight modification on the new edge, extended edge weight between any pair of nodes in the relaxed query keeps unchanged. And deleting a node at the conjunction of two branches may lead to heavier penalty than deleting a node on a single path. This also accords with common sense.

Relaxing a pc-edge into an ad-edge: Let the inconsistent *pc*-edge be $e(v_1,v_2)$. $pc(v_1,v_2)$ needs to be relaxed into $ad(v_1,v_2)$. Edge weight $w(v_1,v_2)$ will be reduced to $\lambda \cdot w(v_1,v_2)$, where $\lambda \in [0,1]$ is a coefficient specified by users. λ shows, to what extent, a *pc* relationship in the query can be taken place by an *ad* relationship. Now considering v_2's ancestor v_i, v_1's descendant v_j, relationship between v_i and v_j is sort of weakened, as $e(v_1,v_2)$ along the path between them is relaxed; while such variation should not affect the other paths. And the penalty of this operation is

$$f_2 = (1-\lambda) \cdot \sum \{w_e(v_i,v_j) | (\forall i \forall j, ad(v_i,v_2) \wedge ad(v_1,v_j))\} \qquad (2)$$

After weight modification on the edge, the extended edge weight between v_i and v_j, satisfying $ad(v_i,v_2) \wedge ad(v_1,v_j)$, will be changed.

Moving a node: Let the inconsistent edge and their NCA be $e(v_1,v_2)$ and v_{nca}, and let the path from v_{nca} to v_1 be $v_{nca}, v_{i_n}, \ldots, v_{i_1}, v_1$. The new established *ad*-edge weight $w(v_{nca},v_2)$ will be assigned with $w_e(v_{nca},v_2)$. And v_2 will lose relationship with the nodes between v_{nca} and v_2. The penalty of moving node v_2 exclusively (without

considering descendants of v_2) is

$$f_{v_2} = w_e(v_1, v_2) + \sum_{t=1}^{n} w_e(v_{i_t}, v_2) \quad (3)$$

However, as is discussed in Section 3.4 Case 2, for any subtree rooted at $v_{2c} \in ch(v_2)$, it may remain under v_1 or follow v_2 after inconsistent edge relaxation. Hence further weight modification and penalty evaluation about the subtree are triggered according to different cases. Let the subtree rooted at v_{2c} be q_s, the node set of the subtree q_s be V_s, the penalty related with subtree q_s be f_s, and penalties of all subtrees be $\sum f_s$, we discuss further adjustments as follows:

Case 1: Subtree q_s rooted at v_{2c} will remain under v_1: New edge weight established between v_1 and v_{2c} is $w(v_1, v_{2c}) = w_e(v_1, v_{2c})$. Further penalty is the deletion of relationships between v_2 and nodes in V_s, for v_2 is also separated away from q_s:

$$f_s = \sum_{\forall v_j \in V_s} w_e(v_2, v_j) \quad (4)$$

Case 2: Subtree q_s rooted at v_{2c} will follow v_2: No weight modification on edges are required for nodes in V_s. As to penalty, different from the above case, relationships between nodes in V_s and nodes along path v_{i_n} to v_1 are removed, while relationships between nodes in V_s and v_2 are kept:

$$f_s = \sum_{\forall v_j \in V_s} w_e(v_1, v_j) + \sum_{\forall v_j \in V_s} \sum_{t=1}^{n} w_e(v_{i_t}, v_j) \quad (5)$$

After edge weight modification, extended edge weights between the nodes separated into two branches are also needed to be rectified to 0. Total penalty of moving a node is the sum of two parts:

$$f_3 = f_{v_2} + \sum f_s \quad (6)$$

Two reasonable properties exist for moving a node:

Property 1. *Moving a node to an upper level will cause more penalty than to a lower level.*

Proof. Let v_2 be moved to v_{i_m} or v_{i_n} and the path from v_{i_m} to v_2 be $v_{i_m}, \ldots, v_{i_{n+1}}, v_{i_n}, \ldots, v_{i_1}, v_1, v_2$ $(m > n \geq 1)$. Suppose v_2 has a single child v_{2c} (multiple children case can be proved similarly), let $\Delta f = f_3(v_2 \to v_{i_m}) - f_3(v_2 \to v_{i_n})$, both in cases 1 and 2, based on Eqs. (3)–(6), we have $\Delta f > 0$:

$$\Delta f = \begin{cases} \sum_{t=n}^{m-1} w_e(v_{i_t}, v_2) & \textit{Case } 1 \\ \sum_{t=n}^{m-1} w_e(v_{i_t}, v_2) + \sum_{\forall v_j \in V_s} \sum_{t=n}^{m-1} w_e(v_{i_t}, v_j) & \textit{Case } 2 \end{cases} . \quad \square$$

Property 2. *Deleting a node will be more penalized than moving a node without considering child distribution.*

Proof. Let the path from v_{nca} to v_2 be $v_{nca}, v_{i_n}, \ldots, v_{i_1}, v_1, v_2$. When v_2 is moved under v_{nca}, the difference between deleting v_2 and moving v_2 based on Eqs. (1) and (3) is

$$\Delta f = \sum_{\forall v_m, ad(v_m, v_{nca})} w_e(v_m, v_2) + w_e(v_{nca}, v_2) > 0 \quad \square$$

In Section 4.1, we have computed $w_q(q) = 11$ for the weight of the original query q, where the weight of each edge is set as 1. When q is relaxed to q_{11}, node *publication* is moved under node *dept* based on the inconsistent edge relaxation rule. During the procedure of relaxation, we compute the penalty with Eq. (6) in which f_s is calculated with Eq. (5). In this case, we have $f_{publication} = w_e(group, publication) = 1$ and $f_s = w_e(group, title) = 1$ since the edges *group*/*publication* and *group*/*title* are lost. Therefore, the weight of q_{11} can be computed as $w_q(q_{11}) = 11 - 2 = 9$. Similarly, we have $w_q(q_{12}) = 6$. When q is relaxed to q_{21}, the *pc*-edge between node *dept* and node *group* is relaxed to an *ad*-edge. Assume the coefficient λ is set to 0.9, the penalty of this relaxation can be computed as 0.3 with Eq. (2). Similar to q_{11}, the penalty caused by moving node *project* under node *dept* is also 2. So $w_q(q_{21}) = 11 - 0.3 - 2 = 8.7$. Similarly we have $w_q(q_{22}) = 5.7$. Now we can sort these four relaxed queries and get the ranking list [q_{11}, q_{21}, q_{12}, q_{22}]. However, if a smaller weight 0.5 is specified on the edge between *group* and *project* while others remain the weight 1, we can recompute the weights of these four relaxed queries and have $w_q(q_{11}) = 7$, $w_q(q_{12}) = 4$, $w_q(q_{21}) = 7.7$, $w_q(q_{22}) = 5.7$. Consequently, the ranking list is changed to [q_{21}, q_{11}, q_{22}, q_{12}].

4.3. Penalty fitness discussion

Evaluating penalty is subject to different weight models and penalty plans. It is difficult to reach a consensus for finding out a perfect model. In the former section, we have shown that our penalty deduced is sound and suits real cases by justifying the possession of some basic qualities that a good penalty strategy should provide. We will now give two other features with respect to weight and penalty rendered on an overall basis.

Free order of processing eSibling nodes: Query relaxation is done in a top-down manner. After processing node v, the order of processing nodes in $ch(v)$ should not make any difference. More generally speaking, for any two nodes of *eSibling* relationship, processing order should also be free. Suppose v_i, v_j be a pair of *eSibling* nodes, basic operations on v_i will affect v_j's extended edge weights. Then from penalty evaluation of basic operations about v_i, affected edge weights may be $w(p(v_i), v_i)$, $w(p(v_i), v_c)$, $w(v_{nca}, v_i)$, where $v_c \in ch(v_i)$ and v_{nca} is the NCA which v_i will be moved under. Given another node v_k, let $w_e(v_j, v_k)$ be affected by one of the edge weights above, then from the definition of the extended edge weight, there should exist an edge $e \in S = \{(p(v_i), v_i), (p(v_i), v_c), (v_{nca}, v_i)\}$ on the path from v_j to v_k. No matter which edge $e \in S$ is on the path, we will have $ad(v_i, v_j)$ or $ad(v_j, v_i)$. This contradicts to the supposition. Thus operations on v_i cannot affect v_j's extended edge weights, and processing *eSibling* nodes is of free order.

Equal penalty for step-by-step computation and batch computation: Penalty can be computed at each step and accumulated together, or evaluated by comparing *query weight* between the final relaxed query and the original user specified query. Our model conforms to the fact well, as penalty is regarded as losing relationships and is implicitly defined by the decrease of query weight, in

which all relationships between nodes in user's query are contained. An interesting question is in which way, penalty can be computed efficiently. Unluckily, the answer is not definite. Step-by-step computation suits the situation where few relaxation occurs, while batch overwhelms the former in case of more relaxation steps. In practice, it is difficult to determine how much relaxation will be done on the original query in advance. We reserve the problem as future work.

5. Adaptive relaxation process

Given a WTPQ q and a certain DTD d_i, we can relax q and generate a set of relaxed queries Q_i using the relaxation rules discussed in Section 3. To get Q_i, we can first relax q and generate a relaxed query q_i that preserves maximum query requirements of q w.r.t. d_i. In other words, q_i receives the minimal penalty w.r.t. d_i. Based on q_i, we can then generate other relaxed queries in Q_i according to the cardinality and disjunctive information provided in d_i. In this section, we introduce the relaxation algorithm for generating q_i from q w.r.t. d_i. The generation of the rest of Q_i will be discussed in Section 6, together with the introduction of top-k query evaluation.

5.1. The relaxation algorithm

Our main algorithm relaxes a WTPQ q with a DTD d in a top-down manner, i.e. relaxation operations carried out on a certain node v are always handled before the operations of $p(v)$. This guarantees that edge adjustments brought in by later processed nodes will not violate the established relationships between earlier processed nodes. The implied subtree constituted by the relaxed nodes always conforms to DTD d during the whole relaxation process, which reflects the correctness of our relaxation algorithm. The query tree is scanned in a way similar to breadth-first traversal. For each node in a query tree, ontology relaxation, node relaxation, recursive relationship relaxation and inconsistent edge relaxation are checked in order. Here recursive relationship relaxation may trigger other relaxation rules as well.

Algorithm 1 (*Adaptive relaxation*).

input: $q=T(V, E, r, w)$, $w_e(q)$ and $d=G(V_d, E_d, r_d)$
output: relaxed query q' and $w_e(q')$
1: globalQueue.addElement(r); f=0;
2: **while** globalQueue is not empty **do**
3: v = globalQueue.pop();
4: v' = FindConsistentDTDNode(v, q, d);
5: **if** $v' \neq NIL$ **then**
6: **if** $v' \in RecursiveTable$ **then**
7: Taking v_p=$p(v)$, v, q, v', and d as input to call Algorithm 3;
8: **else**
9: **if** existAncestor(v) **then**
10: globalQueue.addElement(each $v_c \in ch(v)$);
11: processing repeated node with Case 2 in Section 3.5 while computing penalty f_1 with Eq. (1) and $f = f + f_x$;
12: **else if** $pc(p(v), v) \wedge ad(p(v)', v')$ **then**
13: relaxing $pc(p(v), v)$ with Case 1 in Section 3.4 while computing penalty f_2 with Eq. (2) and $f = f + f_2$;
14: globalQueue.addElement(each $v_c \in ch(v)$);
15: **else if** $(pc(v', p(v)') \vee ad(v', p(v)') \vee eSibling(v', p(v)')$ **then**
16: Taking $v_p = p(v)$, v, v'_p=$p(v)'$, v', q and d as input to call Algorithm 4;
17: **return** q' = q and $w_e(q')$ = $w_e(q)$ $-f$;

Algorithm 1 gives the whole relaxation process. The queue *globalQueue* is used to hold nodes waiting to be processed. At the beginning of each loop, globalQueue.-pop() serves a node to be relaxed. Firstly, we try to find its consistent corresponding node v' in DTD d using function *FindConsistentDTDNode*() in Algorithm 2. Node relaxation will apply when neither a consistent corresponding node is found in d, nor a superclass node is sought out. If v' can be found and is in a recursive cycle in d, Algorithm 3 is called to relax recursive relationships. Otherwise, there are three other relaxation cases: (1) A repetitive pattern appears in the query tree with recursion start node v, but no recursive cycle including v' exists in d. Based on Case 2 in Section 3.5, we delete the repetitive node in the query if the node has already appeared once on the path from v to r (detected by function existAncestor(v)). (2) If node v is not repetitive, and there is an inconsistent edge between $p(v)$ and v on condition that $pc(p(v), v)$ holds in q but $ad(p(v)', v')$ in d, we will relax the pc-edge into ad-edge based on Case 1 in Section 3.4; (3) If node v is not repetitive and the inconsistent edge between v and $p(v)$ satisfies $pc(v', p(v)')$ or $ad(v', p(v)')$ or $eSibling(v',p(v)')$, we call Algorithm 4 to promote node v under the NCA of $p(v)$ and v. In this way, we continuously deal with the nodes in *globalQueue* until the queue is empty. Finally, the relaxed query q' and its query weight $w_e(q')$ can be obtained.

Algorithm 2 (*Find consistent corresponding node FindConsistentDTDNode()*).

input: current node v, DTD d, query q' and globalQueue
output: the consistent corresponding node of v
1: **if** $checkConsistentNode(v, q', d)$ **then**
2: **return** getConsistentDTDNode(v, q', d);
3: **else**
4: Get the superclass node v_{super} of v;
5: **if** $checkConsistentNode(v_{super}, q', d)$ **then**
6: Relaxing v with ontology relaxation rule in Section 3.1 and recording the corresponding penalty into f;
7: **return** getConsistentDTDNode(v_{super}, q', d);
8: **else**
9: globalQueue.addElement(each $v_c \in ch(v)$);
10: Relaxing v with node relaxation rule in Section 3.2 and recording the corresponding penalty into f;
11: **return** NIL;

Algorithm 2, containing ontology relaxation and node relaxation, is actually a function. It firstly check if the consistent corresponding node v' of v exists in d by calling *checkConsistentNode*(). If does, the node v' will be obtained by function *getConsistentDTDNode*(). Otherwise, it will continue to check if the superclass node v_{super} of v exists in d. After making sure v_{super} can be found in d, node v will be relaxed with ontology relaxation rule in Section 3.1 and return the consistent corresponding node v'_{super} of v_{super}. If no match either, node relaxation rule in Section 3.2 will be revoked and NIL will be returned.

5.2. Relaxing recursive relationship

Algorithm 3 (*Relaxing recursive relationship*).

```
input: entry node v_p, start node v, current query q', consistent
   corresponding nodes v' and DTD d
output: relaxed query q'
1:  if pc(v_p, v) then
2:     relaxing pc edge with Case 1 in Section 3.5 while computing
       penalty f_2 with Eq. (2) and f = f + f_2;
3:  qPatternBase = getPatternBase(v_p, v, q');
4:  qPatternTail = getPatternTail(v_p, v, q');
5:  while nextPattern(qPatternBase) ≠ qPatternTail do
6:     deleteNextPattern(qPatternBase) and compute penalty and
       add it into f;
7:  Connect qPatternBase to qPatternTail with ad edge;
8:  DTDPattern = getDTDPattern(v', d);
9:  if ¬ exactMatch(qPatternBase, DTDPattern) then
10:    Call Case 4 in Section 3.5 and compute penalty and add it into
       f;
11: return q';
```

Algorithm 3 handles recursive relationship relaxation. Let v_p and v be the recursion entry node and recursion start node in current query tree q', respectively. v' is the consistent corresponding node of v in DTD d. We firstly check the relationship of node pair (v_p, v). If $pc(v_p, v)$ holds in the query, we will generalize the *pc*-edge between v_p and v into an *ad*-edge based on Case 1 in Section 3.5. Then we determine the first repetitive pattern *qPatternBase* and the last part *qPatternTail* in query q' by calling functions *getPatternBase*() and *getPatternTail*(), respectively. After that, we delete the repetitive pattern between *qPatternBase* and *qPatternTail* based on Case 3 in Section 3.5 if the pattern is repeated for many times. Moreover, we create an *ad*-edge to connect *qPatternBase* with *qPatternTail*. Finally, we use function *getDTDPattern*() to obtain the current recursive pattern *DTDPattern* starting from node v' in DTD d. If *qPatternBase* and *DTDPattern* do not match exactly, we follow Case 4 in Section 3.5 to relax *qPatternBase* against *DTDPattern* using other relaxation rules. During the above process, the corresponding penalties will be computed and recorded.

5.3. Determining NCA and moving nodes

Algorithm 4 describes the procedure to determine the NCA v_{nca} for node pair (v_1, v_2) in query q and DTD d, and promote node v_2 as a descendant under v_{nca}. We begin to search v_{nca} from the parent node of v_1 using the function *checkCommonAncestor*(v', v'_1, v'_2, d) to check if the node v' satisfies *pc* or *ad* relationship with v'_1 and v'_2 in d. Fortunately, lots of work can serve to efficiently judge the *pc* or *ad* relationship between any two nodes in a DTD, in this paper, we use an auxiliary structure called reachability matrix in [10,11] to solve the problem. After v_{nca} is found, we further distribute the child nodes of v_2 according to different relationships between the child nodes to v_1 and to v_2. For each child node $v_{2c} \in ch(v_2)$, after checking the existence of its consistent corresponding node in DTD d, we promote the subtree rooted at v_{2c} under v_1, if neither $pc(v'_2, v'_{2c})$ nor $ad(v'_2, v'_{2c})$ holds in d; otherwise we move the subtree together with v_2.

Algorithm 4 (*Determine NCA and move nodes*).

```
input: a pair of nodes v_1, v_2 and their consistent corresponding nodes
   v'_1, v'_2, query q, source DTD d and globalQueue
output: relaxed query q'
1:  v = p(v_1);
2:  v' = getConsistentDTDNode(v, q, d);
3:  while ¬ checkCommonAncestor(v', v'_1, v'_2, d) do
4:        v = p(v);
5:        v' = getConsistentDTDNode(v,q, d);
6:  for all v_2c ∈ ch(v_2) do
7:        v'_2c = FindConsistentDTDNode(v_2c, q', d);
8:        if v'_2c ≠ NIL then
9:              if ¬pc(v'_2, v'_2c) ∧ ¬ad(v'_2, v'_2c) then
10:                   Connect subtree rooted at v_2c to v_1;
11:             globalQueue.addElement(v_2c);
12: delete e(v_1, v_2), move the subtree rooted at v_2 to NCA v and
    compute penalty and add it into f;
13: return q';
```

6. Top-*k* query evaluation

Given a WTPQ q and a set of heterogeneous data sources $s_1, s_2, \ldots, s_n$ conforming to a set of DTDs $d_1, d_2, \ldots, d_n$, respectively, we can use the relaxation algorithm introduced in Section 5 to generate a set of relaxed queries $q_1, q_2, \ldots, q_n$ together with their query weights $w_e(q_1), w_e(q_2), \ldots, w_e(q_n)$. We denote these query weights as $score(q_1), score(q_2), \ldots, score(q_n)$ in this section. Each $q_i (1 \le i \le n)$ is the relaxed query that preserves the maximum query requirements of q w.r.t. d_i and serves as a base for further relaxation for s_i. With q_i, $score(q_i)$ and d_i given for each source s_i, we are able to evaluate the top-*k* queries. To do this, we may generate all relaxed queries for each data source. That is, for each q_i, we generate the set of relaxed queries $Q_i = \{q_{i1}, q_{i2}, \ldots, q_{im_i}\}$ based on d_i. And then we rank the generated queries in $Q_1, \ldots, Q_n$ and evaluate them in an order based on their scores until we get k results. The generation of Q_i from q_i can be achieved by considering the optional semantics "*" and "?" and the disjunctive semantics "|" defined in the DTD d_i. If q includes quite a few nodes with their correspondent nodes having disjunctive and optional semantics in d_i, the size of Q_i may be increased. In this case, it is time-consuming to generate and evaluate all possible relaxed queries.

To avoid generating and evaluating all relaxed queries, we propose an effective scheduling strategy to dynamically schedule the evaluation of relaxed queries based on the concepts of upper/lower bounds and threshold for searching the top k answers. We call it the BT-based scheduling strategy.

6.1. Overview of BT Strategy

The idea of BT-based scheduling strategy is described as follows. 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_{k1} if

$U(k_1)=\max\{U(i)|1\le i\le n\}$ (i.e., the highest upper bound) and the threshold $\sigma=U(k_2)=\max\{U(i)|1\le i\le n\wedge i\ne k_1\}$ (i.e., the next highest upper bound). Then we start to evaluate q_{k1} on s_{k1} by probing an edge $e(v_1, v_2)$ of q_{k1} at a time. If $e(v_1, v_2)$ cannot be found in s_{k1}, $U(k_1)$ will be decreased; otherwise, $L(k_1)$ will be increased. The probing continues for next edge of q_{k1} until either $L(k_1)\ge\sigma$ or $U(k_1)<\sigma$. If $L(k_1)\ge\sigma$, 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_{k1}; 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_{k2} and the threshold will be chosen based on the updated list of the upper bounds. The process stops until k results are returned.

Let us illustrate the procedure of overview with another example. There are two bookshop XML data sources in Figs. 11 and 12 that maintain the partial or full information of each book: *title*, *isbn*, *price*, *publisher* and *year*. The two bookshop sources conform to schema d_1 in Fig. 13 and schema d_2 in Fig. 14, respectively. To search for the books 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 Fig. 15. The root of the tree (shown in a solid circle) represents the distinguished node.

Based on Definition 8, 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 in Fig. 16 and q_2 in Fig. 17, 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.

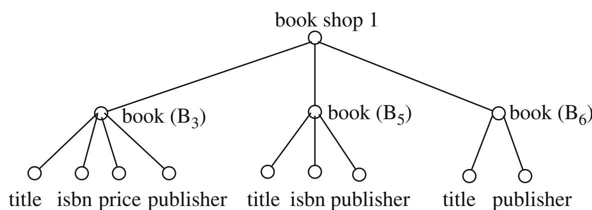


Fig. 11. Example bookshop S_1.

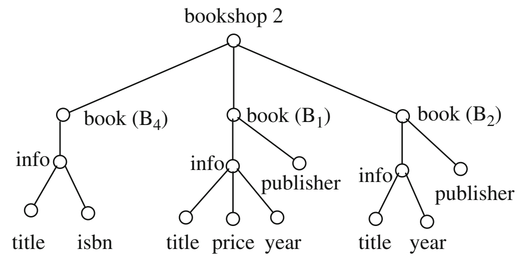


Fig. 12. Example bookshop S_2.

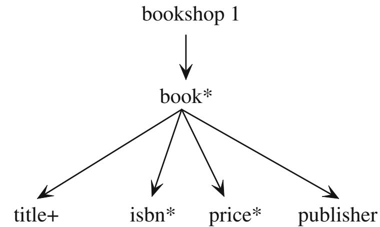


Fig. 13. Schema of the 1st shop d_1.

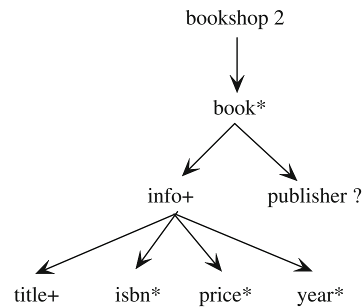


Fig. 14. Schema of the 2nd shop d_2.

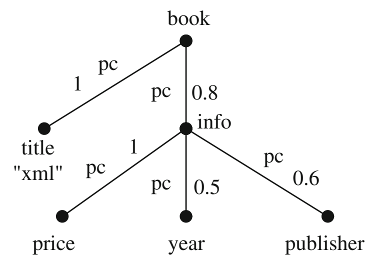


Fig. 15. User's query tree q.

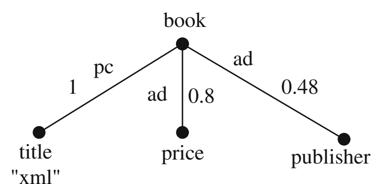


Fig. 16. Relaxed query to the 1st schema q_1.

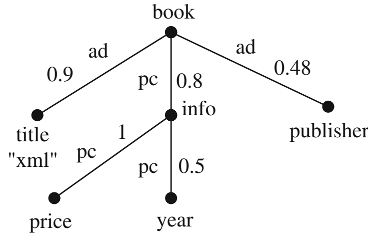


Fig. 17. Relaxed query to the 2nd schema q_2.

In this example, S_2 is chosen as 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.

6.2. Properties

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

Property 3 (*Data source determination and switching*). *At any time of query evaluation, we always evaluate the data source s_{k1} that has the highest upper bound $U(k_1) = \max\{U(i)|1\le i\le n\}$. When an edge $e(v_1, v_2)$ in q_{k1} is evaluated on the data source s_{k1}, 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_{k1}, then the upper bound $U(k_1)$ will be decreased by $U(k_1) = U(k_1) - score(v_2, q_{k1}) - w_e(v_1, v_2)$. 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_{k1}.*
- *If the updated upper bound $U(k_1)$ becomes lower than the threshold σ, then the current data source s_{k1} needs to be suspended and query evaluation will be switched to the data source s_{k2}.*

Property 4 (*Result determination*). *When an edge $e(v_1, v_2)$ in q_{k1} is evaluated on the data source s_{k1}, 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_{k1}, 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_{k1} must be qualified as top-k results. We divide the set of candidates in s_{k1} 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 $\sigma\ge U(k_1)(G_2)$, 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 3.*
- *If $|G_1|>k$, we will evaluate other edges in the query q_{k1} on G_1 to find the top k results.*

Property 3 controls the scheduling of relaxed query evaluation over multiple data sources while Property 4 controls the output of top k results. The BT-based strategy incrementally evaluates the next least penalized relaxed query all the time thus guarantees the monotonic feature of generating the k most relevant 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 3). 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 ($U(2)(G_1) = 4.48 - 0.9 = 3.58$). 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 two 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.

6.3. BT-based scheduling algorithms

Algorithm 5 (*BT-based scheduling strategy*).

input: a set of weighted relaxed queries $\{q_1, q_2, \ldots, q_n\}$ rooted at $\{r_1, r_2, \ldots, r_n\}$ and a set of data sources $\{S_1, S_2, \ldots, S_n\}$
output: top k results
1: call for the function *computingScore*() in Algorithm 8 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 $U(k_1)\ge U(k_2)$, $L(k_1) = 0$;
2: //S_{k1} will be first evaluated
3: put all candidates in S_{k1} into group G;
4: **if** $ch(r_{k1})\ne\phi$ **then**
5: list l = sortAllChildNodes($ch(r_{k1})$);
6: ScheEval(l, q_{k1}, G, $U(k_1)(G)$, $L(k_1)(G)$, σ) in Algorithm 6;

We use Algorithm 5 to initialize query evaluation over the data source S_{k1}. Algorithm 8 is used to compute the weight of each subtree in the query q_{k1} and $score(q_{k1})$ 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_{k1} on the data source S_{k1} with the highest upper bound $U(k_1)$ at any point. Then all the candidates in the data source S_{k1} 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_{k1} 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 [11]); *mergesort*() merges two sorted lists like Merge Sort in [11], 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 6.

Algorithm 6 (*ScheEval(a list l, query q, group G, $U(k_1)(G)$, $L(k_1)(G)$, σ)*).

```
1:  while l ≠ φ do
2:    node v = 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(k1)(G) = U(k1)(G) − score(v, q) − we(e);
6:      if U(k1)(G) < σ then
7:        suspend the current group G;
8:        if σ = = U(k1)(Gx) then
9:          Switching to probe the group Gx in the current data
            source Sk1;
10:         ScheEval(l, q, Gx, U(k1)(Gx), L(k1)(Gx));
11:       else
12:         Switching to the next data source Sk2 due to σ = U(k2);
13:   else if All candidates in G satisfy the edge e then
14:     L(k1)(G) = L(k1)(G) + we(v' parent, v);
15:     if L(k1)(G) ≥ σ then
16:       determineCandidates();
17:     else
18:       list l' = sortAllChildNodes(ch(v)) and list l = mergeSort(l,
          l');
19:   else
20:     //{Partial candidates in G satisfy the edge e}
21:     divideGroup(e, q, G) into two groups G1 that satisfies the
        edge and G2 that does not and putActiveGroup(G2);
22:     U(k1)(G2) = U(k1)(G2) − score(v, q) − we(v'parent, v);
23:     L(k1)(G1) = L(k1)(G1) + we(v'parent, v);
24:     if U(k1)(G2) > σ then
25:       σ = U(k1)(G2);
26:       //{The group G2 in k1 data source would be evaluated at
          next step.}
27:     if L(k1)(G1) ≥ σ then
28:       determineCandidates();
29:     else
30:       list l' = sortAllChildNodes(ch(v)) and list l = mergeSort(l, l');
```

In Algorithm 6, 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) Lines 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) Lines 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 7. Otherwise, we open the child nodes of the node v to expand the current range of edges because the group of candidates cannot be determined based on the current edge e so far. (3) Lines 19–30: Most of the time, only part candidates 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*() inAlgorithm 7. Otherwise, a new edge need to be evaluated on the current group of candidates.

Algorithm 7 (*Function: determineCandidates()*).

```
1:  if |G| = k then
2:    return k results while Stop searching;
3:  else if |G| < k then
4:    return λ results and k = k − |G|;
5:    if σ = = U(k1)(Gx) then
6:      Switching to probe the group Gx in the current data source
        Sk1;
7:      ScheEval(l, q, Gx, U(k1)(Gx), L(k1)(Gx));
8:    else
9:      Switching to the next data source Sk2 due to σ = U(k2);
10: else
11:   list l' = sortAllChildNodes(ch(v)) and list l = mergeSort(l, l');
12:   ScheEval(l, q, G, U(k1)(G), L(k1)(G));
```

Algorithm 7 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. (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_{k1} if we have $\sigma = U(k_1)(G_x)$ or switch to the next data source S_{k2} 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 8 (*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≠φ do
3:       v = getStackTop(S);
4:       existEdgeScore = getEdgeScore(v);
5:       if ch(v)≠φ then
6:           for all v_c ∈ ch(v) do
7:               newEdgeScore = getEdgeScore(v_c);
8:               currentEdgeScore = existEdgeScore × newEdgeScore;
9:               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 = getStackTop(S);
15:      xEdgeScore = getEdgeScore(v_x);
16:      updateEdgeScore(v_x, xEdgeScore + existEdgeScore);
```

Algorithm 8 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.

7. Experiments

We ran the experiments on an Intel P4 3 GHz PC with 512 M memory. Wutka DTDparser [12] was used to analyze the source DTDs and extract their structural information. All relaxed queries were evaluated as XPath patterns in Oracle Berkeley DB XML [13].

Dataset and queries: We used XMark XML data generator [14] to create a set of XML documents with different size from 5 to 40 MB, which conform to *auction.dtd* [15]. These XML documents can be used to test the efficiency of AQR. In order to compare the effectiveness between AQR and FleXPath, we selected the 10 MB document as a base to generate other three 10 MB documents that have different structures. The generated three documents conform to the DTDs d_1 in Fig. 18, d_2 in Fig. 19 and d_3 in Fig. 20, respectively. To keep readers clear, the three documents are named as xmark10MBd1.xml, xmark10MBd2.xml and xmark10MBd3.xml, respectively.

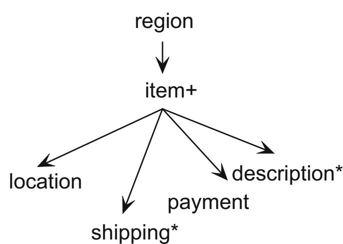


Fig. 18. Schema d_1.

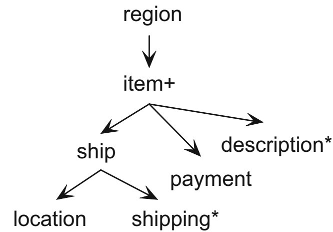


Fig. 19. Schema d_2.

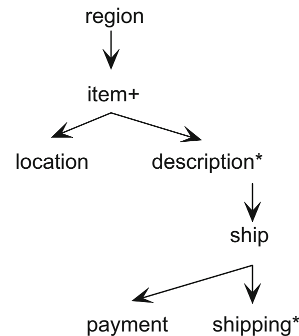


Fig. 20. Schema d_3.

With the structures of the three DTDs in mind, we first designed a query q with some keywords as follows.

- q: //item[./description [./payment.contains('Creditcard') and ./ship [./location.contains('United States') and ./shipping.contains('international')]]]

This query implies that the users are interested in the items: (1) the returned items can be paid with credit card; (2) the returned items can be shipped world wide, i.e., international; and (3) the returned items maybe export to or import from United States. Due to the different structures between the query and the three DTDs, the query q will be relaxed differently for each document in AQR. By analyzing the relaxed queries and corresponding answers, we can show the benefits of AQR in an intuitive way.

To study the efficiency of AQR, we designed a set of queries with complex structures based on *auction.dtd* as follows.

- q_1: //item [./description /parlist and ./mailbox /mail]
- q_2: //item [./description /parlist /mailbox /mail [./text and ./from and ./to]]
- q_3: //item [./description /parlist /listitem and ./mailbox /mail /text[./keyword and ./emph] and ./name and ./payment]
- q_4: //item [./description /xxx /yyy and ./mailbox /mail /text[./keyword and ./emph] and ./name and ./payment]
- q_5: //item[./description [./xxx /yyy and ./mailbox /mail /text [./keyword and ./emph]] and ./name and ./payment]
- q_6://item[./description [./xxx /yyy and ./mailbox /mail /text [./keyword /keyword /keyword and ./emph /xxx]] and ./name and ./payment]

In these queries, we focused on the structural requirements by considering the structural difference between the queries and the DTD, such as the edge "parlist/mailbox" does not exist in the DTD, the edge "description/parlist" satisfies disjunctive semantics and the nodes "mail" and "text" satisfy optional semantics. We further added "xxx" and "yyy" into some of the queries as noise nodes that do not appear in the DTD. We would like to guarantee certain level of computational scale so keywords are not included.

7.1. Effectiveness of relaxation

To demonstrate the effectiveness of AQR, we compared it with FleXPath to show the advantage of leveraging DTD in relaxing users' queries. For the query q designed for testing effectiveness, the maximum number of results is the same as that of the answers when we evaluate the simple query "//item['*United States*', '*Creditcard*', '*international*']", which returns those *item* nodes containing all the three keywords. To check if a relaxed query will generate further results, we checked these documents and found that the maximum number of results are 55, 55, 46 for *xmark10MBd1*, *xmark10MBd2* and *xmark10MBd3*, respectively. Fig. 21 shows the number of relaxed queries to be generated with the increase of k value when we evaluate q as a top-k query over the three documents.

When k = 39, FleXPath has to generate 6, 5, $4\sim7$ relaxed queries and evaluate all of them over the three XML documents, respectively. However, only one relaxed query needs to be generated and evaluated for each document with AQR. When k = 46, FleXPath needs to generate 7 relaxed queries for *xmark10MBd1*, $6\sim8$ relaxed queries for *xmark10MBd2* and $4\sim7$ relaxed queries for *xmark10MBd3*. However, AQR only generates and evaluates 2, 2 and 1 relaxed queries for the three XML documents to obtain the same number of results. Interestingly, we find that the numbers of relaxed queries generated over *xmark10MBd3* for k=39 and 46 are the same. This is because the relaxed queries generated for k=39 can also be used for k=46, i.e., to return top 46 results. Compared with k = 39 and 46, FleXPath generates a lot more new relaxed queries for k=55, most of them either return no new result or output the same results repeatedly. In contrast, AQR only generates 1 new relaxed query for each document, but produces the same set of results as FleXPath. When we process *xmark10MBd3* for k=55, both approaches return no new result, yet generate new relaxed queries. However, AQR only generates 1 new query while FleXPath generates $8\sim16$ new queries.

When k=60, no new result can be found from all the three documents by both approaches because the current value of k (i.e., 60) is larger than the maximum number of results (i.e., 55) for all three documents. In this case, AQR stops generating new queries while FleXPath continues generating $2\sim17$ and $7\sim23$ new queries for *xmark10MBd1* and *xmark10MBd2*, respectively.

From the experiments, we find that FleXPath generates far more relaxed queries compared with AQR. The reason behind this finding is that to meet a large k, FleXPath has to relax a user's query and evaluate it until the root node of the query if necessary. However, AQR is able to stop unnecessary query relaxations early as possible for a particular data source with the guideline of its conformed DTD.

7.2. Impact of edge weights

In this section, we show the impact of the user assigned edge weights on the ranking of generated relaxed queries and the returned search results. Consider the query q, and two schemas d_2 in Fig. 19 and d_3 in Fig. 20. By AQR, two sets of relaxed queries of q generated for d_2 and d_3 are listed in Figs. 22 and 23, respectively. In these two sets, q_{21} and q_{31} are the closest relaxed queries of q w.r.t. d_2 and d_3, respectively.

Firstly, assume all the edges in q hold the default weight 1 as shown in Fig. 24 (we hide the term information of q in the figure). As the query weights of q_{21} and q_{31} are 7 and 8.8 (we assume $\lambda = 0.9$ that means the edge weight of description//payment will be reduced from 1 to 0.9 due to edge relaxation pc-to-ad), q_{31} is evaluated first. After that we compare the query weight of q_{32} which is 5.8 with that of q_{21} and continue the evaluation of q_{21}. Following the similar procedure, we can get the sequence for evaluating all the relaxed queries depending on their query weights. In this case, the sequence is $q_{31}(\text{weight}=8.8) \to q_{21}(\text{weight}=7) \to q_{22}(\text{weight}=6) \to q_{32}(\text{weight}=5.8) \to q_{23}(\text{weight}=5) \to q_{24}(\text{weight}=4) \to q_{33}(\text{weight}=1)$.

Now we modify the edge weights of the query q in Fig. 24 and get two new queries q' and q'' as shown in

		k=39	k=46	k=55	k=60
xmark10MBd1	FleXPath	6 queries	7 queries	10~15 queries	17~27 queries but no new results
	AQR	1 queries	2 queries	3 queries	no new queries
xmark10MBd2	FleXPath	5 queries	6~8 queries	9~15 queries	22~32 queries but no new results
	AQR	1 queries	2 queries	3 queries	no new queries
xmark10MBd3	FleXPath	4~7 queries	no new queries	15~20 queries with 1 new result	no new queries
	AQR	1 queries	no new queries	2 queries with 1 new result	no new queries

Fig. 21. Relaxed queries of q over different sources with the increase of k.

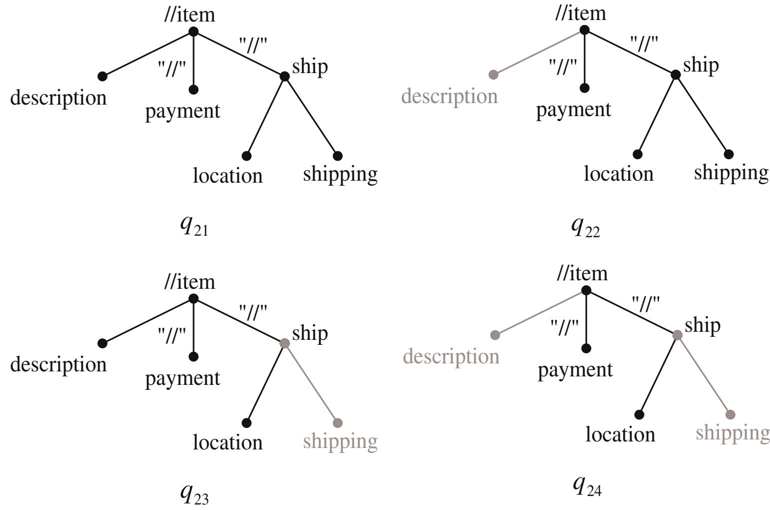


Fig. 22. Relaxed queries w.r.t. schema d_2.

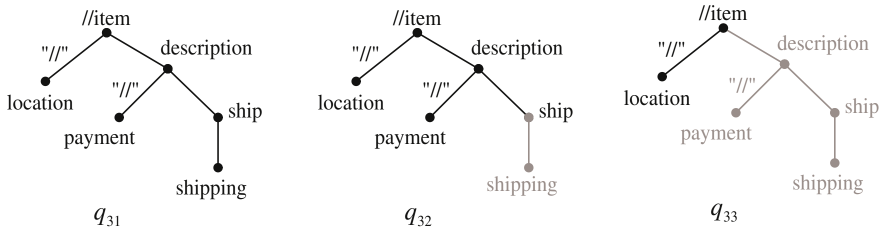


Fig. 23. Relaxed queries w.r.t. schema d_3.

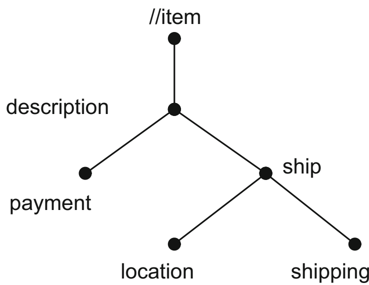


Fig. 24. Query q with the default edge weight 1.0.

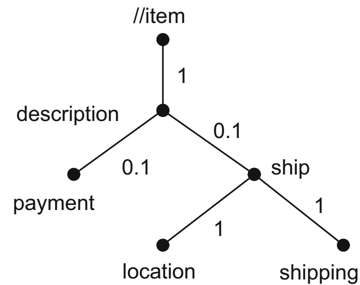


Fig. 26. Query q'' with w_e (*description, payment*) = w_e (*description, ship*) = 0.1.

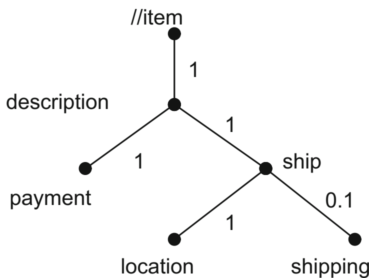


Fig. 25. Query q' with w_e (*ship, shipping*)=0.1.

Figs. 25 and 26, respectively. For q', the evaluation sequence is $q_{31}(\text{weight}=6.1)\rightarrow q_{32}(\text{weight}=5.8)\rightarrow q_{21}$ $(\text{weight}=5.2)\rightarrow q_{23}(\text{weight}=5)\rightarrow q_{22}(\text{weight}=4.2)\rightarrow q_{24}$ $(\text{weight}=4)\rightarrow q_{33}$ $(\text{weight}=1)$. For q'', the evaluation sequence is q_{21} $(\text{weight}=3.4)\rightarrow q_{31}(\text{weight}=2.68)\rightarrow$ $q_{22}(\text{weight}=2.4)\rightarrow q_{23}(\text{weight}=2.3)\rightarrow q_{32}(\text{weight}=1.48)$ $\rightarrow q_{24}$ $(\text{weight}=1.3)\rightarrow q_{33}(\text{weight}=0.1)$.

Fig. 27 shows that when k = 39, q and q' generate the same results from evaluating q_{31} while q'' generates another set of results from evaluating q_{21}. When k = 46, q and q' can get enough results without evaluating new relaxed query while q'' needs to evaluate new query q_{31}

AQR		k=39	k=46	k=47	k=85	k=86	k=92	k=93
q	xmark10MBd2 xmark10MBd3	q_{31}	no new queries	q_{21}	no new queries	q_{22}	no new queries	q_{32}
q'	xmark10MBd2 xmark10MBd3	q_{31}	no new queries	q_{32}	q_{21}	no new queries	q_{23}	no new queries
q''	xmark10MBd2 xmark10MBd3	q_{21}	q_{31}	no new queries	no new queries	q_{22}	no new queries	q_{23}

Fig. 27. Result comparison with the increase of k when the edge weights are assigned differently where q_{31} generates 46 results, q_{32} generates 1 result, q_{21} generates 39 results, q_{22} generates 7 results, q_{23} generates 9 results.

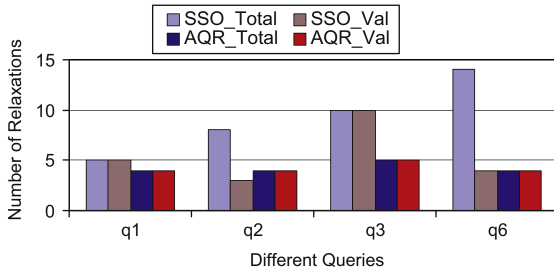


Fig. 28. Relaxed number vs. different queries.

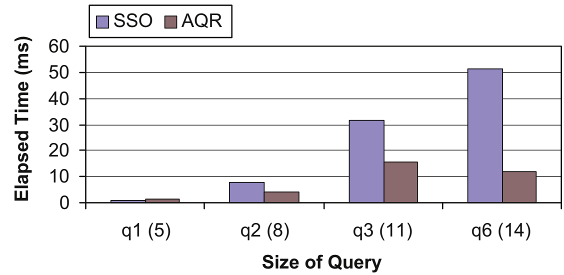


Fig. 29. Elapsed time vs. query size.

because only 39 results can be retrieved by q_{21}. When k=47, q and q' have to evaluate new relaxed queries q_{21} and q_{32}, respectively, to get new results because only 46 results can be retrieved by q_{31}. However q' can keep getting further results from evaluating q_{31} because q_{21} and q_{31} can generate up to 85 results. As shown in Fig. 27, when k is set to 85, 86, 92, and 93, respectively, q, q' and q' output different sets of search results from evaluating different relaxed queries. Therefore, we can say that the same query with different edge weights may result in different ranking lists for evaluating the same set of generated relaxed queries, and consequently returns different sets of results when given the required result number k.

7.3. *Efficiency of relaxation*

To demonstrate the efficiency of the AQR algorithm, we chose the SSO algorithm of FleXPath[2] for comparison. For each query q_i, the set of relaxed queries Q_i are generated by both AQR and SSO and ranked as $\{q_{i,1}, \ldots, q_{i,j-1}, q_{i,j}, \ldots\}$. We call $q_{i,j}$ as a *valuable relaxed query* if it generates at least one new result compared with the result set generated till $q_{i,j-1}$. The relaxed query set will be processed in the ranked order during query evaluation (see Section 7.4).

Varying query size: We selected four queries q_1, q_2, q_3 and q_6 consisting of different number of nodes, i.e. 5, 8, 11 and 14, respectively, and then relaxed them against *auction.dtd*. q_4 and q_5 were not used because they have the same number of nodes (i.e., 11) as q_3. Fig. 28 shows the number of relaxed queries and valuable relaxed queries. Most of the relaxed queries produced by AQR are valuable, while for SSO, only part of the relaxed queries can return query results, such as q_2 and q_6, which do not totally conform to *auction.dtd*. The valuable relaxed query sets for AQR and SSO may not be identical. The set of queries generated by AQR is contained in that generated by SSO.

Fig. 30. Elapsed time vs. query type.

This is because SSO also generates some queries that are too wild, such as "//item" and "//item/description", and the answers returned by these wild queries may not relevant and thus not significant to users. This shows that AQR can guarantee the quality of relaxed queries.

From Fig. 29, both algorithms can handle q_1 with the same time cost. This is because q_1 matches the DTD well and the number of nodes is also small. For q_2, q_3 and q_6, the elapsed time goes up in both algorithms as the number of query nodes increases. AQR is superior in efficiency, because by utilizing DTD, AQR avoids generating a large number of unqualified queries. In addition, although the size of q_6 is larger than that of q_3, AQR needs less time to relax q_6 than q_3. The reason is that the recursive nodes *keyword* can be efficiently relaxed based on our recursive relaxation rule.

Varying query types: To verify the effectiveness for processing different types of queries, we took q_3, q_4, q_5 because they have same number of nodes but different types, and q_6 with the recursive relationship and bigger number of nodes (i.e., 14). From the results in Fig. 30, we find the elapsed time of AQR increases slowly in relaxing the former three queries, and falls down when processing

q_6, this is because recursive relaxation reduces the cost of evaluating q_6. AQR guarantees that no noise nodes or edges exist in relaxed queries. As to SSO, it took almost same time to relax q_3 to q_5 due to the similar size of these queries. However, the elapsed time increases for processing q_6 because SSO has to spend more time for useless relaxation of noise nodes, and relaxation on recursion and inconsistent edges.

7.4. Evaluating top-*k* queries on XML documents

We assessed the performance of AQR with a DTD-validated method DTDVal in answering top-*k* queries. DTDVal leverages DTD to validate relaxed queries produced by SSO before actually evaluating them on documents. Both algorithms can answer top-*k* in an incremental way, by evaluating the least penalized relaxed query on documents in order until top *k* qualified results are retrieved.

Varying k: Due to different types of relaxation it covers, q_5 was selected and tested with *k* varying from 200 to 800 on the 5 MB document. Before our experiments, we checked that more than 800 answers exist in the document. Fig. 31 gives out the total number of relaxed queries and the valuable part in both algorithms. Similar to Fig. 28, most of the relaxed queries generated by AQR can return results, while without DTD information in relaxation process, DTDVal has to invalidate a considerable number of relaxed queries containing noise nodes. Fig. 32 verifies that AQR is better than DTDVal. The similar response time at k=400 and 600 implies that there exists a relaxed query which can return more than 200 results that spans from top 400 to 600.

Varying document size: Fig. 33 shows the case of running query q_5 with *k* set to 1000 and the document size varying from 5 to 40 MB. In the figure, we find that querying larger documents gives quick response time, for the reason that more answers are prone to be found in larger documents. The response time of querying the 20 MB document is almost the same as that of the 40 MB document for both algorithms, that is because they both contain large number of answers and need less relaxed queries.

For a top-*k* query to a larger number of XML data sources and each source contains small number of results, AQR would show greater advantage. FlexPath needs to evaluate all documents while we do not because we know the DTDs and the quality of the generated relaxed queries.

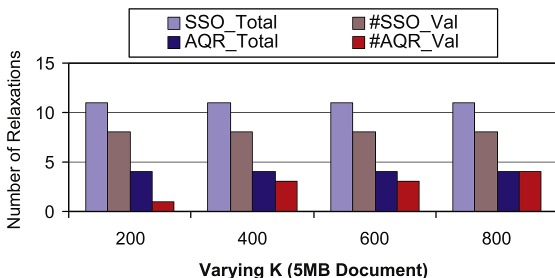


Fig. 31. Relaxed number vs. varying *K*.

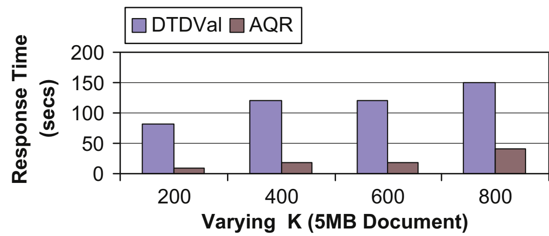


Fig. 32. Response time vs. varying *K*.

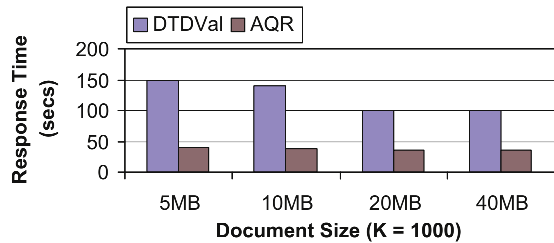


Fig. 33. Response time vs. document size.

8. Related work

Query relaxations on structure have been studied recently. Some approaches propose to relax queries that return no result. Delobel and Rousset [16] define three kinds of relaxations: unfolding a node (replicating a node by creating a separate path to one of its children), deleting a condition at a node, and propagating a condition to its parent node. Schlieder [17] considers relaxations on an XQL query: deleting nodes for making the context loose, inserting a node between inner nodes for specifying more specific context and renaming nodes for changing the search context. Koudas et al. [18] present a framework for relaxing join and selection conditions in relational schema. Some work has been done to enhance information retrieval with the help of structure queries. VCAS [19] is an approach for vague content and structure retrieval, which partitions a user's structure query into a SCAS (strict content and structure) subquery and a CO (content only) subquery and produces results by combining the results of two subqueries. Amer-Yahia et al. [1] propose the relaxations of weighted tree pattern queries: generalizing nodes or edges, deleting leaf nodes, promoting subtrees. Weights are added to either edges and nodes for determining penalties in a straightforward manner. In FleXPath [2], they put forward a more systematic relaxation method by spanning the space of relaxation and generating a closure of relaxed queries for a tree pattern query. Retrieved documents are ranked based on either structure first, keyword first, or combined schemes. Recently, Zhou et al. [20] propose query relaxation by using malleable schemas where they first utilize the duplicates in different data sets to discover the correlations within a malleable schema and then relax users' queries based on the derived correlations.

Top-*k* query processing has also been extensively studied in the past. 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 [21] 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 [22,23] 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 [24–26], 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 [28], MPro [29] and TPUT [30] algorithms show that interleaving probes on tuples results in substantial savings in execution time. In addition, Upper [28] 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 [31,32], top-*k* keyword queries for XML have been studied via proposals extending the work of Fagin et al. [25,27] 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 [3] 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 cannot be determined until all candidates are evaluated.

Different from the above work, the purpose of our adaptive query relaxation is to find the set of most relevant answers that best match users' intention specified in a query from large number of heterogeneous data sources. It would be very time-consuming and thus not acceptable to relax the query blindly and wildly. As such, AQR chooses to relax users' queries "relative" to the data sources based on their DTDs, which guarantees the relaxed queries are best suited for those sources that conform to the DTDs. During query relaxation, no unqualified or unnecessary relaxed queries will be produced. AQR also provides extended sibling and recursive relationship relaxations which are not provided in FleXPath [2]. Besides, we compute the ranking score for each relaxed query by taking into account both the relaxing operations and the weights of edges, which allows to specify and adjust search requirements specific to users. Additionally, 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. Particularly 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.

9. Conclusions

In this paper, we presented a novel query relaxation approach—AQR that adaptively relaxed a query to different XML data sources based on their conformed DTDs and users' intentions. A set of schema-aware relaxation rules were designed, and a pertinent penalty model based on weight modification was developed. Adaptive relaxation algorithms and strategies for top-*k* query evaluation were implemented and illustrated through a comprehensive set of experiments to show the effectiveness and efficiency of AQR.

Acknowledgements

We would like to thank anonymous reviewers for their helpful comments on this article. This work was supported partly by the Australian Research Council Discovery Project under the Grant no. DP0878405 and the Research Grant Council of the Hong Kong SAR, China under the Grant no. 419109.

References

[1] S. Amer-Yahia, S. Cho, D. Srivastava, Tree pattern relaxation, in: EDBT, 2002, pp. 496–513.
[2] S. Amer-Yahia, L.V.S. Lakshmanan, S. Pandit, FleXPath: flexible structure and full-text querying for XML, in: SIGMOD, 2004, pp. 83–94.
[3] A. Marian, S. Amer-Yahia, N. Koudas, D. Srivastava, Adaptive processing of top-K queries in XML, in: ICDE, 2005, pp. 162–173.
[4] A.Y. Halevy, Answering queries using views: a survey, VLDB J. (2001) 270–294.
[5] A.P. Sheth, J.A. Larson, Federated database systems for managing distributed, heterogeneous, and autonomous databases, ACM Comput. Surv. (1990) 183–236.
[6] B. Ding, J.X. Yu, S. Wang, L. Qin, X. Zhang, X. Lin, Finding top-k min-cost connected trees in databases, in: ICDE, 2007, pp. 836–845.
[7] G.J. Bex, F. Neven, T. Schwentick, K. Tuyls, Inference of concise DTDs from XML data, in: VLDB, 2006, pp. 115–126.
[8] G.J. Bex, F. Neven, S. Vansummeren, Inferring XML schema definitions from XML data, in: VLDB, 2007, pp. 998–1009.
[9] H. Weinblatt, A new search algorithm for finding the simple cycles of a finite directed graph, JACM 19(1) (1972) 43–56.
[10] B. Choi, G. Cong, W. Fan, S.D. Viglas, Updating recursive XML views of relations, in: ICDE, 2007, pp. 766–775.
[11] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, Cambridge, MA, 2001.
[12] Wutka DTD parser 〈http://www.wutka.com/dtdparser.html〉.
[13] Oracle Berkeley DB XML 2.3 〈http://www.oracle.com/database/berkeley-db/xml/index.html〉.
[14] XMark XML data generator 〈http://monetdb.cwi.nl/xml/index.html〉.
[15] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, R. Busse, XMark: a benchmark for XML data management, in: VLDB, 2002, pp. 974–985.
[16] C. Delobel, M.-C. Rousset, A uniform approach for querying large tree-structured data through a mediated schema, in: International Workshop on Foundations of Models for Information Integration, 2001.
[17] T. Schlieder, Similarity search in XML data using cost-based query transformations, in: WebDB, 2001, pp. 19–24.
[18] N. Koudas, C. Li, A.K.H. Tung, R. Vernica, Relaxing join and selection queries, in: VLDB, 2006, pp. 199–210.
[19] S. Liu, W.W. Chu, R. Shahinian, Vague content and structure (VCAS) retrieval for document-centric XML collections, in: WebDB, 2005, pp. 79–84.
[20] X. Zhou, J. Gaugaz, W.-T. Balke, W. Nejdl, Query relaxation using malleable schemas, in: SIGMOD, 2007, pp. 545–556.
[21] M.J. Carey, D. Kossmann, On saying "Enough Already!" in SQL, in: SIGMOD, 1997, pp. 219–230.
[22] N. Bruno, S. Chaudhuri, L. Gravano, Top-k selection queries over relational databases: mapping strategies and performance evaluation, ACM Trans. Database Syst. 27 (2) (2002) 153–187.
[23] C.-M. Chen, Y. Ling, A sampling-based estimator for top-k query, in: ICDE, 2002, pp. 617–627.
[24] R. Fagin, Fuzzy queries in multimedia database systems, in: PODS, 1998, pp. 1–10.

[25] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, in: PODS, 2001, pp. 102–113.
[26] R. Fagin, Combining fuzzy information: an overview, SIGMOD Record 31 (2) (2002) 109–118.
[27] R. Fagin, Combining fuzzy information from multiple systems, in: PODS, 1996, pp. 216–226.
[28] A. Marian, N. Bruno, L. Gravano, Evaluating top-k queries over web-accessible databases, ACM Trans. Database Syst. 29 (2) (2004) 319–362.
[29] K.C.-C. Chang, S.-w. Hwang, Minimal probing: supporting expensive predicates for top-k queries, in: SIGMOD, 2002, pp. 346–357.
[30] P. Cao, Z. Wang, Efficient top-K query calculation in distributed networks, in: PODC, 2004, pp. 206–215.
[31] M. Theobald, R. Schenkel, G. Weikum, An efficient and versatile query engine for TopX search, in: VLDB, 2005, pp. 625–636.
[32] R. Kaushik, R. Krishnamurthy, J.F. Naughton, R. Ramakrishnan, On the integration of structure indexes and inverted lists, in: SIGMOD Conference, 2004, pp. 779–790.