THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRB6/09

Automatic XML Keyword Query Refinement

Zhifeng Bao, Jiaheng Lu, Tok Wang Ling
and Xiaofeng Meng

June 2009

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Automatic XML Keyword Query Refinement

Zhifeng Bao [#] Jiaheng Lu [*] Tok Wang Ling [#] Xiaofeng Meng [*]

[#] *School of Computing*
National University of Singapore

[*] *School of Information and DEKE, MOE*
Renmin University of China

***Abstract*— Existing XML keyword search methods focus on how to find relevant and meaningful data fragments for a keyword query, assuming each keyword is intended as part of it. However, user's queries usually contain irrelevant or mismatched terms, spelling errors etc, which causes the search results to be either empty or meaningless. In this paper, we introduce the problem of automatic XML keyword query refinement, where automatic means the search engine should be able to adaptively decide whether a query Q needs to be refined during the processing of Q, and at the same time find a list of promising refined query candidates and their matching results over XML data, without any user interaction or a second try. In order to achieve this goal, we build a primary framework which consists of two core parts: (1) we build a novel query ranking model to evaluate the quality of a refined query RQ, which takes into account of the relevance of RQ w.r.t Q over XML data, the morphological/semantical similarity between Q and RQ, and the dependence of keywords of RQ in XML data. (2) We integrate the exploration of RQ candidates and the generation of their matching results as a single problem at the same time of processing Q, which is fulfilled within a one-time scan of related keyword inverted lists optimally. Finally, an extensive empirical study verifies the efficiency and effectiveness of our framework.**

I. INTRODUCTION

The great success of web search engine has made keyword search a popular way for users to access information. As XML is becoming a standard for data exchange and representation, there is a growing research interest to support keyword search on XML [1], [2], [3], [4], [5], [6]. In contrast to the IR-style keyword search which employs a ranked retrieval paradigm for producing an ordered list of results according to their relevance with the query, XML keyword search enforces a conjunctive search semantics (i.e. all the keywords should be covered in each query result), such as LCA (Lowest Common Ancestor) [1] and its variants [2], [3], [4]. Among those proposals, SLCA (Smallest LCA) is widely adopted [3], where each SLCA result contains all query keywords but has no descendant whose subtree also contains all keywords.

Unfortunately, all of them assume each keyword in the query is intended as part of it. However, a user query may often be an imperfect description of their real information need. Even when the information need is well described, a search engine may not be able to return the results matching the query as stated possibly due to term mismatch, keyword ambiguity, unintentional spelling error etc, as shown in the example below.

Example 1: Consider a query Q = {database, publication} issued on a bibliographic document in Figure 1, which most likely intends to find all publications in database area. However, terms *inproceedings* and *article* which are semantically equal to *publication* are used in this XML data, causing the query to have no matching result. As another example, consider Q_3 in Table I with the same intention as Q. According to SLCA semantics, the subtree rooted at *author:0.0* is returned, while it is meaningless to user, as user's concern should be *inproceedings*. □

Besides, another frequently encountered scenario is, a user query may be too restrictive to have a meaningful matching result. E.g. consider Q_4 in Table I issued on Figure 1, intending to find John's publications about XML in year 2003. However, the only result covering all the keywords is the root node of XML document, which is meaningless to the user.

In contrast to the scenario of web search where query refinement is carried out to make the query result more specific, in this paper we focus on another extreme - the original query has no meaningful matching result over XML data, and needs to be refined to a closely related query that has meaningful matching results. Sponsored search [7], for example, is one of the application scenarios of our XML query refinement problem, in which we attempt to match enormous number of queries to a much smaller corpus of XML-formatted advertising lists, rather than the set of all possible web pages.

Therefore, the question becomes whether we can offer a solution during search which automatically refines the queries that have no meaningful matching result, in order to better represent users' search needs and help users more easily find the relevant information, without any initial result retrieval or any intervention on user part. This is what we mean by *automatic XML keyword query refinement* as addressed in this paper. This differs with XML keyword query relaxation through boolean OR search [8], which heavily relaxes the search intention of original queries.

In particular, there are four critical issues that an automatic XML keyword query refinement framework should address.

Issue 1: It should be able to adaptively decide whether the original query Q needs to be refined during the processing of Q, without an initial result retrieval.

Issue 2: It should be able to find a list of refined query (RQ) candidates where each RQ is assured to have meaningful matching result over the XML data, and meanwhile find the matching results for those RQs.

Issue 3: *Effectiveness* - It should be able to provide an objective and comprehensive query ranking model to evaluate the quality of the above RQs found in Issue 2.

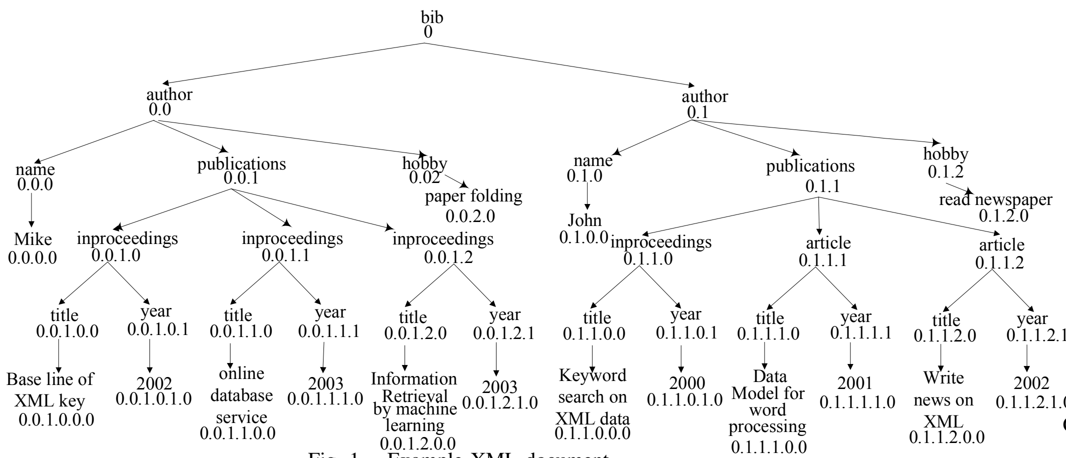


Fig. 1. Example XML document

Original query	Refined query
Q_1: IR, 2003, Mike	RQ_1: Information Retrieval,2003,Mike
Q_2: Mike,publication	RQ_2: Mike,publications
Q_3: Database,paper	RQ_3: Database, inproceedings
Q_4: XML,John,2003	RQ_4: XML,John
Q_5: mechin,learn	RQ_5: machine, learning
Q_6: hobby,news,paper	RQ_6: hobby, newspaper
Q_7: on,line,data,base	RQ_7: online, database

TABLE I
QUERY BEFORE AND AFTER REFINEMENT

Issue 4: *Efficiency* - In addressing the above three issues, it should be able to scan the corresponding keyword inverted lists as few times as possible (optimally only once).

Towards building an automatic query refinement framework that addresses the above four issues simultaneously, there are two major technical challenges.

The first challenge is to define what a *meaningful query result* should be for an XML keyword query, as it will be used to judge whether a query needs to be refined or not in both Issue 1 and Issue 2. Compared to the IR-style keyword search whose search target is usually the flat document, the search target of an XML keyword query is usually implicit or unfixed. Therefore, we propose to utilize the statistics of underlying data to identify the target node(s) that a keyword query intends to *search for* first, then propose an enhanced notion of SLCA by using the concept of target node(s) to further constrain the meaningfulness of a query result.

The second challenge is, it is unknown whether the refinement is required or not before processing the original query. A brute force approach needs to submit a query for an initial result retrieval, before deciding whether the refinement should be used [9]. However, it defeats the primary objective as mentioned in Issue 4, as it may cause multiple times' scan of the corresponding keyword inverted lists, and needs extra user intervention. Another approach is to statically find a ranked list of RQ candidates alone before running the search methods such as [10]; however, those RQ candidates may not necessarily have matching result in XML data.

In order to address the above challenges, a natural solution is to integrate the job of looking for the desired RQs of the original query and the job of finding the matching results of such RQs as a single problem, during the processing of original query. Our major contributions towards building this framework are summarized as below.

- We introduce the problem of automatic XML keyword query refinement.
- We define four major refinement operations in a rule-based way, where each refinement rule is associated with a dissimilarity score; moreover, we propose an enhanced notion of SLCA by taking the *search for node* of a query into account, to judge whether a query has meaningful matching result.
- To address Issue 3, we exploit the statistics of underlying XML data to design a novel query ranking model to evaluate the quality of RQ candidates, by taking into account of the semantical and morphological similarity between RQ and Q, the relevance of RQ w.r.t the original search intention, and the keyword dependencies of RQ in XML data.
- We design a dynamic programming solution to efficiently find the RQs with the minimum dissimilarity w.r.t the original query Q for a given refinement rule set.
- We propose three efficient solutions for XML keyword query refinement: a stack-based approach, a partition-based approach and short-list eager approach. Those algorithms need only one-time scan of the corresponding keyword inverted lists, and the partition and short-list eager approach are orthogonal to any existing SLCA computation methods.
- We implement all the above proposed techniques in a keyword search engine prototype called *XRefine*. Extensive experiments show its efficiency and effectiveness.

The rest of the paper is organized as below. Section II describes the related work. Section III presents some preliminaries. Section IV presents our query ranking model. Section V describes how to find the optimal RQ. Section VI presents three query refinement approaches. Section VII presents the index construction. Experimental result is reported in section VIII and we conclude in section IX.

II. RELATED WORK

Existing works on XML keyword search model XML data as a labeled tree, and focus on how to find the most relevant data fragments for a keyword query [1], [4], [3], [11]. In particular, LCA (lowest common ancestor) [1] is first proposed to find XML nodes, each of which is the lowest common ancestor containing all query keywords in its subtree. XSEarch [2] introduces the concept of interconnection as a variation of LCA, which claims two matches are interconnected if there are no two distinct nodes with the same tag name on the

path between these two nodes (through their LCA), excluding themselves. Subsequently, SLCA (smallest LCA [4], [3]) is proposed to find all nodes, each of which contains matches to all keywords in its subtree and none of its descendants does. In particular, Li et al. [4] incorporate Meaningful LCA search in XQuery; XKSearch [3] focuses on studying efficient algorithms to compute SLCA by designing an index lookup and scan eager algorithm. Multiway-SLCA [8] is proposed to further optimize the performance of finding SLCA by maximizing the skip of redundant LCA computations contributing to the same SLCA result. Based on SLCA, XSeek [5] infers the return nodes of a query by keyword match pattern in XML data and the concept of entities inferred from data schema. [6] proposes an XML TF*IDF to do result ranking.

Query refinement in IR is also related to this work. It can be divided into two main spectrums: (1) A fully automatic query refinement [12], [9], [13], which modifies and subsumes terms to queries according to a thesaurus or terms in documents, with no intervention on user part; the thesaurus itself may automatically or manually be generated. (2) An interactive query refinement [14], such as relevance feedback which requires user to manually identify relevant documents whose terms are removed from (or added to) the query. Our work follows the first spectrum, and takes two levels of automation: first, it automatically peruses the XML document and makes appropriate modifications on the query by exploiting the refinement rules; second, it can automatically generate the query results for both original and refined query within only one-time scan of keyword lists. This is convenient for users to quickly decide which list of results meets their search needs.

In XML retrieval, a dominant query refinement strategy is query expansion, which adds new terms highly correlated with the original query to enhance the result quality. In particular, [15] adopts pseudo-feedback to choose the expanded terms from the top ranked results of original query; [16] expands the original query based on the feedback of result relevance provided by users; TopX [17] generates potential expansion terms for queries by using a thesaurus database WordNet [18]. All of them are complementary to our work, as they are only applied on queries having *non-empty* result. Recently, [10] introduces a preprocessing stage to clean the raw text and extract a high quality keyword query, in order to significantly reduce the search space of a keyword query in relational database. Though, a potential problem is the cleaned query is not guaranteed to have matching results in database.

III. Preliminaries

Data Model We model XML data D as a rooted, labeled tree; each node n of the tree corresponds to an element in D, denoted as tag:deweyID, where tag is the tag name of n, and deweyID is the dewey label of n in D [19]. E.g. in Figure 1, the first *author* element under *bib* is assigned a label 0.0.

Keyword query Q={k_1, k_2, ..., k_n} is a set of keywords, each k_i may match the tag name or value term in XML data D.

A. Meaningful SLCA

Despite the inherent ambiguity of a keyword query, an XML search engine has to identify the target(s) that users desire to *search for*, namely *search for node*. It is first addressed in our recent work [6] by a statistics based approach and extended in this paper for a query that may need to be refined. We first describe two concepts defined in [6] and adopted in this paper: *node type* and XML document frequency (*XML DF*).

Definition 3.1: Node Type : The type of a node n in XML document D is the prefix path from the root node to n. ■

Similar to [6], to facilitate our discussion we use the tag name instead of the prefix path to represent the node type T.

Definition 3.2: ***XML DF*** f_k^T is the number of T-typed nodes containing keyword k in their subtrees in XML document. ■

f_k^T is defined in an analogous way to *DF* in original TF*IDF [20], except that we use f_k^T to distinguish statistics for subtrees rooted at different *node types*. E.g. in Figure 1, $f_{``XML"}^{inproceedings}$ = 2, as two inproceedings 0.0.1.0 and 0.1.1.0 contain "XML".

Intuitively speaking, a node type T is the desired *search for node* if (1) it is related to as many keywords as possible, and (2) XML nodes of type T should contain enough but non-overwhelming information. Therefore, we design Formula 1 to measure the confidence of a node type T to be the desired *search for node* w.r.t a given query Q.

$$C_{for}(T,Q) = \ln(1 + \sum_{k \in Q} f_k^T) * r^{depth(T)} \quad (1)$$

where r is a reduction factor ranging in (0,1), and $depth(T)$ represents the depth of T-typed nodes in XML data. Here, we use *Sum* of f_k^T to combine the statistics of all keywords for each node type T, as some query keywords may not appear in the XML document. By Formula 1, we can infer a list L of desired search for node candidates, based on which we propose an enhanced notion of SLCA to constrain the meaningfulness of an SLCA result of a query in Definition 3.3.

Definition 3.3: (**Meaningful SLCA**) A matching result r is a meaningful SLCA of query Q on XML document D, if $r \in$SLCA(Q,D) by SLCA definition in [3] (i.e. r contains all the keywords in either their labels or the labels of their descendants, and has no descendant that also contains all the keywords), and r is a self or descendant of some node type $T \in L$ inferred by Formula 1. ■

Definition 3.4: A keyword query Q is said to *need refinement* if Q has no *meaningful SLCA* on XML data D. ■

Consider Q_6 and RQ_6 (in Table I) issued on Figure 1. *author* is the only search for node candidate by Formula 1. The only SLCA of RQ_6 is *hobby:0.1.2*, which is the descendant of *author*, and thus is a meaningful SLCA by Definition 3.3. However, the only SLCA of Q_6 is the root node, i.e. Q_6 has no meaningful SLCA and needs refinement by Definition 3.4.

Lemma 1: Given two keyword sets S_1 and S_2, if S_1 is a subset of S_2 and there is at least one *meaningful SLCA* result in D based on S_2, then there is also at least one meaningful SLCA result in D based on S_1.

The proof is straightforward due to the conjunctive feature of SLCA search semantics. We do not show it in detail here.

B. Refinement Operations

As reported by the web query logs tracing users' search modifications [21], a frequently used strategy by users is deleting terms presumably to obtain greater coverage, and term substitution is the major strategy adopted by search engines. Similarly, we define four major refinement operations at term level for XML keyword search: (1) *term deletion*, (2) *term merging*, (3) *term split*, (4) *term substitution*. *Term merging* is needed when a keyword is mistakenly split by user, e.g. a query {online,newspaper} may often be written as {on,line,news,paper} by user. *Term split* is needed as users may mistakenly merge neighboring keywords in typing. *Term substitution* is wide ranging, which mainly includes spelling error correction (Q_5 in Table I), synonym substitution (Q_3), acronym expansion (Q_1) and word stemming (Q_2).

Definition 3.5: A refinement rule r associated with a refinement operation *op* is in form of: $S_1 \overrightarrow{op} S_2$, where S_1 and S_2 are two keyword sets, and r has an associated dissimilarity score ds_r, which models the similarity between S_1 and S_2. ■

TABLE II

SAMPLE REFINEMENT RULES WITH THEIR DISSIMILARITY SCORES

Rule#	Rule r	ds_r
r1	on,line → online	1
r2	data,base → database	1
r3	article → inproceedings	1
r4	learn,ing → learning	1
r5	mechin → machine	2
r6	WWW → world,wide,web	1
r7	online → on,line	1

Some typical refinement rules are listed in Table II. In particular, for term merging/split and spelling error correction, ds_r can be the variants of some morphological metric such as string edit distance from S_1 to S_2. E.g. the dissimilarity ds_r of a one-time term merging/split is 1, as a single space is removed/added, such as r1, r2, r4 and r7 in Table II. For r5, ds_{r5} = 2 as two string edits are needed to correct the spelling error. For synonym substitution such as r3, ds_r can be the similarity score provided by the semantic lexical database such as WordNet [18]. For acronym expansion such as r6, a score of 1 is designated. Since term deletion has the greatest potential in changing the meaning of original query, we adopt the principle that its dissimilarity score is greater than any other three rules throughout this paper[1]. Alternatively, the refinement rules can be obtained from document mining, query log analysis [21] or manual annotation [13].

Definition 3.6: Given a set R of refinement rules, the dissimilarity between Q and a RQ, denoted as $dSim(Q, RQ)$, is the minimal sum of the cumulated ds_r among all possible refinement sequences based on R. ■

In the rest of the paper, whenever we mention the "dissimilarity of RQ", it refers to $dSim(Q, RQ)$.

IV. RANKING OF REFINED QUERIES

In section V, *dSim(Q,RQ)* is defined as a preliminary quality metric for a RQ, mainly based on its lexical and morphological similarity w.r.t Q. However, it is inadequate without considering the local context (i.e. the XML data) on which RQ is issued, especially for those that have the same dissimilarity. Motivated by the ability of statistics in modeling patterns or drawing inferences about the underlying data, we utilize statistic knowledge to build an in-depth query ranking model. In general, the overall quality of a RQ can be evaluated in two complementary aspects: (1) the similarity of RQ which captures the relevance of RQ w.r.t the original search intention and (2) the dependence score of RQ which captures the keyword dependencies of RQ in XML data D.

We begin with describing some concepts used in the following discussion. As mentioned in section III-A, each query result is a subtree rooted at the search for node of type T, which is called as *subtree of type T*. $tf(k, T)$ is used to denote the *term count* of k in subtrees of type T; G_T represents the number of distinct keywords contained in subtrees of type T. E.g. in Figure 1, tf("XML",*author*)=3 meaning that there are three occurrences of "XML" within the subtrees rooted at *author*; and $G_{article}$=14 meaning that there are 14 distinct keywords within the subtrees rooted at *article*. All the statistics data used in this section can be pre-computed in parsing the XML data, similar to what we have done in [6].

A. Similarity Score of a RQ

Given a query Q issued on XML document D and a RQ candidate, we propose four intuitive guidelines in an incremental way to compute the similarity of RQ w.r.t the original search intention.

Guideline 1: The more frequently the keyword in RQ appears within the search for node T, the more important RQ is. □

Following Guideline 1, Formula 2 is designed to accumulate the term count of all keywords in RQ, and G_T is chosen as a normalization factor to prevent a bias to a search for node T whose subtrees are of large size.

$$Imp(RQ, T) = \sum_{k \in RQ} \frac{tf(k, T)}{G_T} \qquad (2)$$

In addition, we notice that each keyword in a query has its own ability to discriminate the query results, i.e. each $k \in Q$ as a constraint of Q actually has different importance, as shown in Example 2 and addressed in Guideline 2.

Example 2: Consider a query Q={XML, twig, pattern, join} issued on DBLP, where no matching result is found. The desired search for node is *inproceedings* by Formula 1. RQ_1={XML, twig, pattern} and RQ_2={XML, twig, join} are two candidates, and the keywords "join" and "pattern", which are deleted to form RQ_1 and RQ_2 respectively, actually have different importance as a constraint of Q, as $f_{pattern}^{inproceedings}$=17297 and $f_{join}^{inproceedings}$=946[2]. □

Guideline 2: The more discriminative a keyword k_i (that is either deleted from the original query Q or newly generated by merging/split/substitution rules) is w.r.t the original query Q, the lower the rank of RQ, which is resulted from the corresponding refinement involving k_i, should be assigned.□

[1]To facilitate our discussion, the dissimilarity score of a single term deletion rule is 2 throughout all examples in this paper.

[2]These statistics can be validated by an online demo of our previous work [22] at http://xmldb.ddns.comp.nus.edu.sg

By Definition 3.2, the *XML DF* f_k^T provides an effective way to measure the discriminative power of a keyword k, as guideline 2 is in line with the design intuition of document frequency. In other words, the less *XML DF* of a keyword k_i (i.e. $f_{k_i}^T$) is, the more discriminative this k_i is w.r.t. Q. As a result, Formula 2 is designed to address Guideline 2 alone.

$$Imp_{k_i}(Q,T) = \log_e \frac{N_T}{1+f_{k_i}^T} \quad (3)$$

where N_T is the total number of nodes of type T in D. *log* function is applied to normalize the raw ratio.

Take term deletion as an example, k_i is one of the keywords that are deleted from Q to form a RQ. Thus, the less frequent k_i appears in the subtrees rooted at T-typed nodes, the more discriminative k_i is to Q, i.e. the RQ resulted from deleting this k_i from Q is less favored.

Guideline 1 favors the RQ whose keywords have large term frequencies, which is analogous to the intuition of *TF* part in *TF*IDF* definition; while Guideline 2 favors the RQ whose deleted or newly generated keyword has the largest importance in original query, which is analogous to the intuition of *IDF* part. Therefore, we define the similarity $\rho_{RQ}(Q,T)$ of a refined query RQ w.r.t Q, for a given search for node T in Formula 4, where the first multiplier addresses guideline 1, and the second multiplier addresses guideline 2 by accumulating the importance of k_i involved in refining Q to RQ.

$$\rho(RQ,Q|T) = Imp(RQ,T) * \sum_{k_i \in (RQ \triangle Q)} Imp_{k_i}(Q,T) \quad (4)$$

where $RQ \triangle Q$ denotes a set of keywords that are either deleted from Q or newly generated by term merging/split/substitution rules to produce RQ.

So far, we only consider the case that the desired search for node T of a query is unique. However, the keyword ambiguity problem [6] may cause more than one T to have comparable search for confidence, according to our statistics analysis in Formula 1. Therefore, we introduce guideline 3.

Guideline 3: For an original query Q that has multiple desired search for node type T, the confidence $C_{for}(T,Q)$ of each such T should be taken into account. The higher the confidence of T as a desired *search for node* is, the more important its associated similarity $\rho(RQ,Q|T)$ is. □

To address guideline 3, we incorporate the confidence of T as the desired *search for node* (see Formula 1) and its similarity, as shown in Formula 5.

$$\rho(RQ,Q) = \sum_{T \in T_{for}} C_{for}(T,Q) * \rho(RQ,Q|T) \quad (5)$$

where T_{for} denotes a set of candidates for the desired search for node. Note that, Guideline 3 holds based on the principle that both the original and refined query share the same search for node(s).

Lastly, by taking the semantic and morphological dissimilarity $dSim(Q,RQ)$ between Q and RQ into account, whose intuition is mentioned in guideline 4, the similarity of a RQ w.r.t original query Q is presented in Formula 6

Guideline 4: The smaller the dissimilarity between Q and RQ is, RQ is closer to Q in term of search intention. □

$$\rho(RQ,Q) = w^{(dSim(Q,RQ))} * \sum_{t \in T_{for}} C_{for}(T,Q) * \rho(RQ,Q,T) \quad (6)$$

where $dSim(Q,RQ)$ denotes the dissimilarity between Q and RQ as described in Definition 3.6. w is a decay factor ranging in (0,1) to enforce Guideline 4, and w=0.8 is a good choice as evident by our empirical study in section VIII-C.

B. Dependence Score of a RQ

In evaluating the quality of a RQ, the above similarity function emphasizes the relevance between Q and RQ, which has a limitation: the query terms are assumed to be mutually independent. As a complementation of the similarity score, the dependence relationships between the keywords in RQ are captured, namely as the dependence score of RQ.

Guideline 5: A RQ is effective for a certain search for node T, if RQ has as many keywords as possible that co-occur frequently in the subtrees of type T. □

As the desired search for node T may not be unique, for convenience we first discuss the case that T is unique. In order to quantify the dependence of keywords in a refined query RQ, we novelly utilize a variant of *association rule* [23]. For each keyword $k_i \in RQ$, we measure how often another keyword $k \in RQ$ appears in the subtrees of type T that contain k_i, as shown in Formula 7:

$$C(k_i \Rightarrow k) = \frac{f_{k,k_i}^T}{f_{k_i}^T} \quad (7)$$

where $f_{k_i}^T$ represents the number of subtrees of type T that contain keyword k_i, and f_{k,k_i}^T denotes the number of subtrees of type T that contain both k_i and k.

As shown in Formula 8, the inner sum is a cumulation of how often each other keyword $k_i \in RQ$ appear together with k, while the outer sum cumulates such score for all keywords in RQ. In addition, as Guideline 5 usually favors the RQ with large size, a normalization factor $|RQ|$ is introduced.

$$Dep(RQ,Q|T) = \sum_{k \in RQ} \frac{\sum_{k_i \in RQ, k_i != k} C(k_i \Rightarrow k)}{|RQ|} \quad (8)$$

Lastly, when multiple search for nodes have comparable confidence scores by Formula 1, we adopt Guideline 3 again to get the overall dependence score $Dep(RQ,Q)$ as below.

$$Dep(RQ,Q) = \sum_{T \in T_{for}} (C_{for}(T,Q) * Dep(RQ,Q|T)) \quad (9)$$

Finally, our query ranking model is completed by a weighted sum of the similarity score and dependence score of a refined query RQ forms the overall rank of RQ in Formula 10.

$$Rank(RQ) = \alpha * \rho(RQ,Q) + \beta * Dep(RQ) \quad (10)$$

where α and β are tunable weights that reflect the importance of each metric. α=β=1 is the default choice, and the impact of different choices will be evaluated in section VIII-C.

V. Dynamic Programming for Refined Queries

Since the RQs that have both minimum $dSim(Q, RQ)$ and meaningful matching result over XML data D is unknown ahead of processing Q, we should adaptively explore those RQs during the processing of Q. As can be seen in any query refinement algorithm proposed in section VI later, a fundamental problem encountered is: we can obtain a set T of keywords, each of which is from a rule set R and original query Q and does exist in XML data D (see line 1 and 9 of Algorithm 2). However, it remains a challenge to efficiently materialize a RQ from T, such that RQ has the minimum dissimilarity(Q,RQ). This is what we mean the *exploration of optimal RQ*, as defined below:

Problem Formulation: Given two keyword sets S={k_1,k_2,...,k_s} (S denotes the original query Q) and T={k'_1,k'_2,...,k'_t}, and a set R of refinement rules. We aim to find a RQ, which is a subset of T and $\forall RQ' \subseteq T$, $dSim(Q,RQ) \leq dSim(Q,RQ')$. We develop a bottom-up dynamic programming method, namely *getOptimalRQ(S,T)*, to resolve this problem.

Sub-problems: We create subproblems as below. Let $0<i\leq s$ be an integer. Let $S[1,i]$= {k_1,k_2,...,k_i} be a sub-sequence of S. Let C be an array of length ($|S|$+1), where $C[i]$ is the minimum dissimilarity between $S[1,i]$ and some $RQ \subseteq T$. Our final goal is to compute a value for $C[|S|]$, which is the minimum dissimilarity between S to some $RQ \subseteq T$.

Notations: Each $k_i \in S$ is associated with a set of refinement rules, denoted as $R(k_i)$, $R(k_i)$={r | r=<$*k_i \rightarrow k'_m$,...,k'_n >}, where $*k_i$={k_j, k_{j+1},...,k_{i-1}, k_i} is a subset of S ended with k_i and k'_m,...,$k'_n \subseteq T$}. For each rule r, its left and right hand side are denoted as $LHS(r)$ and $RHS(r)$ respectively.

Initialization: $C[0]$ = 0, which means the dissimilarity between an empty query and any other query is 0.

Recurrence Function: Considering the subproblem of computing $C[i]$ where $0<i\leq|S|$, we have three options.

Option 1: when the ith keyword $k_i \in S$ also appears in T, then the dissimilarity score remains unchanged.

Option 2: when k_i does not appear in T, and term deletion is applied to delete k_i.

Option 3: for a refinement rule r, if $LHS(r) = *k$ and $RHS(r) \subseteq T$, C[i] should be equal to a sum of $C[i-|LHS(r)|]$ and ds_r. If more than one rule can be applied here, the one with the minimum sum is selected. This case is used to handle term merging, split and/or substitution.

Among these three options, the one with the minimum value is assigned to $C[i]$, as summarized in Formula 11.

$$C[i] = min \begin{cases} C[i-1] & \text{if } k_i \in T \\ C[i-1] + cost\ of\ deleting\ k_i & \text{if } k_i \notin T \\ C[i-|LHS(r)|] + min\{ds_r\} & \\ \quad \text{if } k_i \notin T \text{ AND } LHS(r) = *k_i \text{ AND} & \\ \quad RHS(r) \subseteq T, \text{for each } r \in R(k_i) & \end{cases} \quad (11)$$

Lemma 2: The refined query RQ generated by Function *GetDSimilarity* satisfies the following properties: (1) RQ is a subset of T; (2) RQ has the smallest dissimilarity among all possible RQ candidates resulted by applying rules in R; (3) RQ have at least one meaningful SLCA result over D.

Note that, *getOptimalRQ* is insensitive to the order of keywords in S, and a running example is shown as below.

Example 3: Given a query Q={WWW,article,machine, learn,ing} and a keyword set T={machine,inproceedings, learning,worldwide,Web,World,Wide}, and three relevant rules r3, r4 and r6 in Table II are identified. Figure 2 shows how array C is filled during the process of *getOptimalRQ(Q,T)*. To compute C[1], option 2 offers a cost of C[0]+cost of deleting "WWW"=2, while option 3 offers a cost of C[0]+ds_{r6}=1. So C[1]=min(2,1)=1. Similarly, C[2]=C[1]+1=2, C[3]=2 as "machine" exists in T, C[4]=2+2=4, and C[5]=min(C[3]+1, C[4]+2)=3. Finally, the optimal RQ={World,Wide,Web,inproceedings,learning}, and $dSim(RQ, Q)$ = 5. □

	WWW	article	machine	learn	ing
0	1	2	2	4	3

Fig. 2. A running example of finding the optimal RQ

Time Complexity: *getOptimalRQ* runs in $|Q|$ loops, where in the worst case, each subset of Q is related to a certain refinement rule r. Since the cost of locating such r is O($log|R|$), then its time complexity is: O($\sum_{i=1}^{|Q|}\sum_{j=1}^{i} log|R|$) = O($|Q|^3 log|R|$).

As a summary, *getOptimalRQ* serves two purposes here. *First*, it generates the optimal RQ together with its $dSim(Q,RQ)$. *Second*, as a side product, a ranked list of some (but not all) non-optimal RQ candidates by $dSim(Q, RQ)$ can also be obtained, as they are in fact the intermediate results kept during the processing of *getOptimalRQ*. They will be used as the candidates for Top-K RQs later in section VI.

VI. Automatic Query Refinement Algorithms

The main challenge towards an *automatic* query refinement is that it is unknown whether any refinement is needed ahead of processing the original query, as each RQ must have meaningful SLCA results over XML data by Definition 3.4. A straightforward solution is to infer all potential RQ candidates based on the given refinement rule set, and try them one by one until the desired RQ is found. However, it causes multiple times scan of the corresponding keyword inverted lists, and needs intervention on user part.

Therefore, we propose to integrate the job of looking for the refined queries of Q and generating their matching results together to guarantee the existence of meaningful SLCA result for each RQ found, and meanwhile scan the keyword inverted lists as few times as possible (optimally only once). In particular, three algorithms are designed.

A. Stack-based Query Refinement

As a basic solution to address the above challenge, we extend the stack-based algorithm in [3] to find the optimal RQ (in term of $dSim(Q, RQ)$) and meanwhile generate its SLCA results in Algorithm 1.

The structure of the stack is similar to that in [3]. Each stack entry contains *id* and an array $keywords$ of boolean values. Different from [3], $keywords$ here includes not only the keywords of Q, but also the new keywords generated after a consultation on a pertinent refinement rule set (line

3). A stack entry e denotes the node whose Dewey label is composed of the *id*s from the bottom entry up to e. In a certain entry e, $keywords[i]$=T means the sub-tree rooted at the node denoted by e (directly or indirectly) contains keyword $k_i \in KS$; otherwise, $keywords[i]$=F. E.g. the top entry of stack in Figure 3(a) shows node 0.0.1.0.0.0 contains "line" and "base". $result$ is employed to preserve the current optimal RQ candidate(s) and their SLCA results. Readers can refer to [3] for details of SLCA computation. Here, we only emphasize the significant difference of our method with that in [3].

Algorithm 1: Stack-based Query Refinement

input : Q={k_1,k_2,...,k_n}, refinement rule set R, XML document D
output : result={(RQ_{min}, SLCA(RQ_{min}))}

1 Boolean needRefine = true;
2 Let min = ∞, RQ_{min} = $\emptyset$; result $\leftarrow \emptyset$; stack $\leftarrow \emptyset$;
3 Let KS = getNewKeywords(Q) + Q; bool $keywords$ = [$|KS|$];
4 {S_1,S_2,...,S_m} $\leftarrow$ getInvertedLists(KS);
5 **while** *(!end(S_i) or stack is not empty)* **do**
6 v_s = getSmallestNode(); /* $1 \leq s \leq |KS|$ */
7 p = lca(v_s, stack);
8 **while** *(stack.size>p.length)* **do**
9 entry e = stack.pop(); KS$\leftarrow$keywords of witness T in e;
10 **if** *(isMSLCA(e,Q))* **then**
11 result.add(Q,e); needRefine = false;
12 reset all entries in $keywords$ to false;
13 **if** *(needRefine)* **then**
14 <RQ, $dSim(Q,RQ)$> = getOptimalRQ(Q,KS);
15 **if** *(($dSim(Q,RQ)$<min) $\cap$ (isMSLCA(e,RQ)))* **then**
16 RQ$_{min}$ = RQ; min = $dSim(Q,RQ)$;
17 result.clear(); result.add(RQ,e);
18 reset terms only in RQ to false for all entries;
19 pass the rest witness T to current top entry e;
20 **for** *(j = p.length to v_s.length)* **do**
21 stack.push(v_s[p],v_s[length]);

In Algorithm 1, it first finds a set KS of keywords that appear in either R or Q by applying the relevant rules in R (line 3), and merges the inverted lists for each keyword in KS (line 4). Each time a node v_s (from the sort merged input lists) is visited, the following action is repeated: LCA of v_s and the node denoted by the top entry in stack is computed (line 6-7).

For each entry e popped from stack, KS is reset to be a set of keywords contained by e (line 9). Then, $isMSLCA()$ checks whether e is a meaningful SLCA of original query Q by Definition 3.3. If so, Q does not need to be refined, and all entries of $keywords$ of any stack entry are set to be false (line 10-12); otherwise, *getOptimalRQ* is invoked to get the $RQ(\subseteq KS)$ with minimum dissimilarity for e (line 14); RQ_{min}, min and the optimal RQ candidate in $result$ are updated accordingly (line 16-17). Another difference with [3] is at the updating of witness information in e's parent entry: only the keywords that are unique to RQ is set to false, while those that are in common with other RQ candidates or Q are kept as true (line 18-19).

A running example of Algorithm 1 is given as below.

Example 4: Consider a Q={on,line,data,base} issued on Figure 1. Two relevant rules r1 and r2 in Table II are identified; thus, KS={on,line,data,base,online,database}, and each stack entry is of size 6. For convenience, letter *o, l, d, b, ol, db* denote "on", "line", "data", "base", "online" and "database".

As Figure 3(a) shows, node 0.0.1.0.0.0 is visited first, and the entry for l and b are set to true, and RQ={l,b} with $dSim(Q,RQ)$=4 is returned by *getOptimalRQ* (line 14), as term deletion is applied on Q twice. As shown in Figure 3(b), when node 0.0.1.1.0.0 is visited, a better RQ={ol,db} with $dSim(Q,RQ)$=2 is found due to twice term merging, so RQ_{min} is updated, and its SLCA result is stored into $result$. Since *ol* and *db* are also part of other RQs, we don't reset their entries to false; instead, their witness is passed down to their parent. Through traversing node 0.1.1.0.0.0, 0.1.1.1.0.0 and 0.1.1.2.0.0, each RQ candidate detected has larger dissimilarity than current RQ_{min}. After the last node 0.1.1.2.0.0 is visited, each entry is popped from stack, and it is found that the SLCA result for RQ_1={online,data,base} and RQ_2={on,line,database} are the root bib:0. Although both RQ_1 and RQ_2 have smaller dissimilarity than RQ, neither of them is unqualified as they do not have any meaningful SLCA result in Figure 1 by Definition 3.3. □

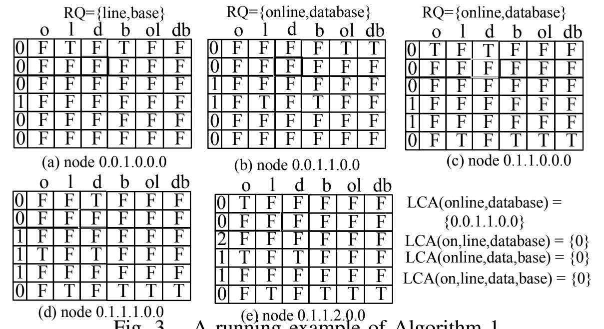


Fig. 3. A running example of Algorithm 1

Theorem 1: Given a query Q issued on XML document D, Algorithm 1 is able to find the optimal RQ (in term of $dSim(Q,RQ)$) which has meaningful SLCA result within a one-time scan of the corresponding keyword inverted lists.

Proof: [Sketch] In Algorithm 1, line 5 guarantees a one-time scan, line 10 and 15 guarantee the optimal RQ must have at least one meaningful SLCA result. ■

B. Partition-based Algorithm for Top-K Query Refinement

As we can see from Algorithm 1, it has two limitations. *First*, for each node v_s visited, *getOptimalRQ* has to be invoked to find the RQ that has SLCA results within the subtree rooted at v_s. That may cause the same RQ which has multiple matchings in XML document to be generated (through *getOptimalRQ*) multiple times, which is a redundant computation. *Second*, by Definition 3.3 the document root is a typical meaningless SLCA, as users are only interested in the fragments of XML data, while Algorithm 1 does not take advantage of this point. Therefore, we propose a partition-based approach to further improve the efficiency. The basic idea is to partition the XML document into a list of ordered partitions as defined below:

Definition 6.1: (**Document Partition**) Given an XML document tree D, a subtree D_i is a document partition of D if the root node R_{D_i} of D_i is the ith child of D's root node.

E.g. in Figure 1, there are 2 document partitions of D: D_1 rooted at author:0.0 and D_2 rooted at author:0.1.

As most users concern on the Top-K RQs, we extend our approach to support Top-K query refinement. The main procedure is as follows: we first maintain a ranked list to store the approximate Top-2K RQ candidates in term of $dSim(Q, RQ)$ during the processing of Q. In the end, we apply the complete query ranking model (proposed in section IV) to generate the final Top-K RQs from the $2K$ candidates. ***RQSortedList*** is developed to store the up-to-date Top-$2K$ RQs during the procedure of query refinement. It is implemented as a sorted list with a B-tree index built on the dissimilarity of RQ, where method *insert* and *remove* can be done in O(log2K) time. Besides, method *hasRQ*, which is used to check whether a RQ to be inserted is already in the list, can be done in O(1) time by maintaining a separate hashtable whose key is RQ itself.

Algorithm 2 presents the details of partition-based approach.

Algorithm 2: Partition-based Top-K query refinement

```
input  : Q={k_1,...,k_n}, refine rule set R, XML document D
output : result={(RQ_1,SLCA(RQ_1)),...,(RQ_K,SLCA(RQ_K))}
1  Let result ← ∅;   Let KS = getNewKeywords(Q) + Q;
2  Let RQSortedList = a list of RQs sorted by dSim(Q,RQ);
3  {S_1,S_2,...,S_m} ← getInvertedLists(KS);
4  while (!end(S_i) for each i∈[1,m]) do
5      v_s = getSmallestNode();              /* 1≤s≤m      */
6      D_pid = getDocPartition(v_s);
7      {S'_1, S'_2,..., S'_m} ← getKLPartition(pid);
8      move cursor of S_i to the node next to the end of each S'_i;
9      Let T = {k_i | S_i' is not empty};
10     {<RQ_i,dSim(Q,RQ_i)>|i∈[1,2K]}=getOptimalRQ(Q,T,2K);
11     foreach RQ_i do
12         if (dSim(Q,RQ_i) < RQSortedList.max) then
13             if (!RQSortedList.hasRQ(RQ_i)) then
14                 RQSortedList.remove(2K);
15                 RQSortedList.insert(<RQ_i,dSim(Q,RQ_i)>);
16             slca_RQ_i = computeSLCAs({S'_1, S'_2,..., S'_m},RQ_i);
17             result.add(RQ_i,slca_RQ_i);
18     reset S'_1, S'_2,..., S'_m to empty;
19 Apply Formula 10 on result to get final Top-K RQs;
```

Firstly, similar to stack-based approach, all new keywords are generated via a consultation on a given pertinent refinement rule set R (line 1). A cursor is maintained for each keyword list S_i. The algorithm runs in an iterative way: as long as the end of all keyword lists haven't been reached, the smallest node v_s in document order is selected (line 5), and the document partition that contains v_s is located by Definition 6.1, denoted as D_{pid}, where pid is the label of this partition's root node (line 6). Function *getKLPartition* is responsible for identifying the corresponding sublist S'_i of each keyword list S_i within partition D_{pid}, based on the property that pid is the prefix of the dewey label of each node in each S'_i (line 7). Accordingly, the cursor of each S_i is moved to the node next to the end of S'_i (line 8).

An extension of Function *getOptimalRQ(Q,KS,2K)* is invoked to find the Top-2K RQ candidates (if they do exist) within partition D_{pid} (line 9-10); this extension is easy to achieve, as those RQ candidates are in fact preserved as the intermediate results during the exploration of optimal RQ. For each RQ_i in the Top-2K RQs found, if the dissimilarity of RQ_i is smaller than that of the lowest-ranked query in $RQSortedList$ and RQ_i has not been inserted before, then RQ_i is inserted into $RQSortedList$ (line 13-15), and any existing SLCA computation method (such as [3], [8]) can be employed to find the SLCAs of RQ_i within partition D_{pid} (line 16) and add them into result (line 17). Lastly, the overall query ranking model is applied to do an elaborate ranking on those $2K$ RQ candidates, to get the final Top-K RQs and their SLCA results (line 19).

Time Complexity: If indexed lookup in [3] is adopted for SLCA computation, the SLCA computation costs O(F*$KlogK$*$|S'_1|md$log$|S'|$), where $S'(S'_1)$ is the max (min) size throughout lists S'_1 to S'_m, F is the fanout of document root, m is number of keywords involved and d is the document depth. The total cost by *getOptimalRQ* is O(F*m^3). Thus, the total cost is O(F*($KlogK$*$|S'_1|mdlog|S'|$+m^3)).

Advantages: (1) Compared to the stack-based one, all computations whose SLCA result is the document root can be skipped, as shown in Example 5. (2) Within each partition, we are able to decide the current Top-2K RQ candidates before computing their SLCA results. As shown in line 12-17, for partition D_j whose associated RQ candidates have larger dissimilarity than that of the current Top-2K RQs, we can skip the SLCA computation of such RQ candidates on D_j, which is an important optimization. (3) Within each partition D_i, $getOptimalRQ(Q,RQ)$ is employed once for those RQ candidates that have multiple matching results in D_i.

Example 5: Consider a query Q={article,online,data,base} issued on Figure 1, and Top-2 RQs are expected. Rules r2, r3 and r7 in Table II are found to be relevant to Q. For illustrative purpose, we only list five *RQ candidates* in an ascending order of its dissimilarity.

RQ_1: {article,online,database}; RQ_2: {article,on,line,database}
RQ_3: {inproceedings,online,database} (1 merge, 1 substitution).
RQ_4: {inproceedings,on,line,data,base} (1 split, 1 substitution).
RQ_5: {inproceedings,online,base} (1 substitution, 1 deletion).

For partition D_1 in Figure 1, the partitioned keyword lists are: S'_{online}=$S'_{database}$={0.0.1.1.0.0}, S'_{on}=S'_{data}=$S_{article}$={}, $S_{inproceedins}$={0.0.1.0, 0.0.1.1, 0.0.1.2}, S'_{line}=S'_{base}={0.0.1.0.0.0}. *getOptimalRQ* returns RQ_3 (with $dSim(Q, RQ_3)$=2) and RQ_5 (with $dSim(Q, RQ_5)$=3) as Top-2 RQs. As $RQSortedList$ is empty, both RQ_3 and RQ_5 are inserted and their SLCA results are computed. Then, we move to next partition D_2, where S'_{on}={0.1.1.0.0.0}, S'_{line}=S'_{base}=S'_{online}=$S'_{database}$={}, S'_{data} = {0.1.1.1.0.0}, $S'_{article}$ = {0.1.1.1, 0.1.1.2} and $S'_{inproceedings}$ = {0.1.1.0}. Now, *getOptimalRQ* only returns RQ = {article} with $dSim(Q, RQ)$ = 4 (as two term deletions are applied on Q), which is even larger than the dissimilarity of K-th RQ, i.e. 3; so we do not need to invoke any SLCA computation for D_2. Lastly, $result$={<RQ_3, inproceedings:0.0.1.1>, <RQ_5, publications:0.0.1>} is returned as the Top-2 RQs. □

Features: Algorithm 2 has two salient features.
(1) It is orthogonal to any existing methods of computing the SLCA results of a query on a certain XML document.
(2) The job of exploring the Top-K refined queries and generating their SLCA matching results are fulfilled within a one-time scan of the inverted lists for each keyword in KS (see line 3 and 4).

Lemma 3: Algorithm 2's query refinement is orthogonal to any existing approaches of generating the SLCA results of a query on a certain XML document.

Theorem 2: Algorithm 2 is able to return the Top-K RQ according to their dissimilarity, and meanwhile generate their associated results within a one-time scan of keyword inverted lists.

Proof: [Sketch] Firstly, line 5 guarantees each involved keyword list is traversed only once. Secondly by Lemma 2, Function *getOptimalRQ* is able to find the Top-K RQs, and $RQSortedList$ stores the up-to-date Top-K RQs during keyword list traversal at each partition. Lastly, by Lemma 3 the correctness of SLCA results is guaranteed. Therefore, Algorithm 2 is able to correctly fulfill Top-K query refinement. ■

C. Short-List Eager Algorithm for Top-K Query Refinement

As we can see, both Algorithm 1 and 2 require a full scan of the corresponding keyword lists, although they need only one-time scan. In practice, however, the frequencies of query keywords typically vary significantly [3]. Therefore, during the exploration of Top-K RQs, if we can start from the RQ candidates that contain the keyword of the shortest inverted list first, it is possible to skip the full scan of all inverted lists involved. This idea is presented in Algorithm 3, which runs in two main steps. In step 1, Top-K RQs are found (line 4-16). In step 2, existing methods is invoked to compute SLCA results of each RQ found in step 1 (line 17-18). The input and output are the same as those of Algorithm 2.

Algorithm 3: Short-List Eager Algorithm

1 Let $RQSortedList$ be a list of RQs sorted by dissimilarity;
2 Let $KSet$ = getNewKeywords(Q) + Q; Let $C_{potential}$=0;
3 $\{S_1,S_2,...,S_m\} \leftarrow$ getInvertedLists(allKeywords);
4 **while** *($C_{potential} \leq RQSortedList.max$)* **do**
5   k_i = the keyword in $KSet$ with the shortest inverted list S_i;
6   **foreach** *partition D_{pid} in S_i* **do**
7   **foreach** *keyword $k \in KSet$ other than k_i* **do**
8   **if** *(k appears in Partition D_{pid})* **then**
9   insert k into KS_{pid};
10         $\{<RQ_i, dSim(Q,RQ_i)>|i\in[1,K]\}$ = getOptimalRQ(Q,KS_{pid},K);
11   **foreach** RQ_i **do**
12   **if** *($dSim(Q,RQ_i) < RQSortedList.max$)* **then**
13                 $RQSortedList$.removeQuery(K);
14                 $RQSortedList$.insert($<RQ_i, dSim(Q,RQ_i)>$);
15     $KSet$ = $KSet$ - k_i; remove S_i from $\{S_1,...,S_m\}$;
16   Compute $C_{potential}$ = getOptimalRQ(Q,$KSet$);
17 **foreach** $RQ_i \in RQSortedList$ **do**
18   result.add(RQ_i, computeSLCAs(RQ_i));
19 Apply Formula 10 on *result* to get final Top-K RQs;

The core part of Algorithm 3 is how to set the stop condition for Top-K RQ exploration, i.e. whether the potentially minimum dissimilarity is larger than the dissimilarity of the K-th query in $RQSortedList$ when $RQSortedList$ is already full (line 4). $C_{potential}$ denotes the potentially minimum dissimilarity for those RQ candidates *unexplored* yet. If it is greater than the dissimilarity of the K-th RQ in $RQSortedList$, then any RQ candidate found later can never be one of the final Top-K RQs, and we can safely stop Step 1. Otherwise, the current shortest list S_i is selected, and for each partition D_{pid} containing k_i, keyword sequence KS_{pid} collects all keywords covered in D_{pid} by random accessing each other keyword list (line 7-9). Then *getOptimalRQ* is invoked to find Top-K RQs within D_{pid}, and qualified RQs are put into $RQSortedList$ (line 10-14).

A salient feature of this approach is line 15-16: at the end of each iteration, all refined queries that contain k_i have been identified, so the shortest list S_i is removed, and k_i is removed from $KSet$ accordingly. Lastly, the potentially minimum dissimilarity $C_{potential}$ between Q and some RQ (which is a subset of the updated $KSet$) is computed, which is used in the stop condition checking of next iteration.

Discussion. First, the orthogonality to SLCA computation methods holds for Algorithm 3. Second, the performance of Algorithm 3 depends on two factors: (1) whether the RQs that cover the keyword with the shortest inverted list are among the final Top-K RQs. (2) how early the first match of each RQ in the final Top-K RQs appears in XML data. Based on this analysis, we can have a smarter choice of k_i and S_i in each iteration (line 6): the k_i which either appears in the RHS of refinement rules related to Q or never appears in the LHS of any rule related to Q (i.e. the keyword that does not need any refinement), and also has the shortest inverted list should be chosen first. In this way, the RQ containing such k_i should have a high probability to be one of the final Top-K RQs, and thus the exploration of Top-K RQs can finish earlier.

Example 6: Consider Top-2 query refinement on Q_4 in Table I, where the only applicable refinement rule is term deletion. Initially, S_{XML} =<0.0.1.0.0.0, 0.1.1.0.0.0>, S_{John}=<0.1.0.0>, S_{2003}=<0.0.1.1.1.0, 0.0.1.2.1.0>, and the shortest list is S_{John}, which is contained in the partition with pid 0.1. Then we search S_{XML} and S_{2003}, and find that the current minimum dissimilarity is 2 (as a term deletion is needed), with keywords {XML,John}. Then we scan the next shortest list S_{2003}, and identify the partition with pid 0.0. Similarly, keywords "2003" and "XML" are found to exist in partition 0.0. As a result, these two queries are the RQs with the minimum dissimilarity.□

Time Complexity. In the worst case, each keyword in $KSet$ is involved in Top-K RQ exploration, and let m=$|KSet|$. In each loop, the cost of finding all partitions covering k_j is $|S_j|$ (line 7), so let P_{k_j} denote number of partitions containing k_j; random accesses to other keyword lists cost $\sum_{i=j+1}^{|KSet|} log|S_i|$ (line 8-10) (assuming keyword lists are sorted by length ahead, i.e. $|S_j| \leq |S_i|$, $\forall j<i$); *getOptimalRQ* costs $2|Q|^3$(line 11,17);

operations on $RQSortedList$ is O(1). Thus, its time complexity is $\sum_{j=1}^{m}(|S_j| + P_{k_j} * (\sum_{i=j+1}^{m}(|Q|^3 + log|S_i|) + T_{slca}))$, where T_{slca} denotes SLCA computation time for Top-K RQs, depending on the concrete algorithm adopted.

VII. Index Construction

When parsing the input XML document, we pre-compute and collect the below information for each node v visited: (1) assign a Dewey label $DeweyID$ [19] to v; (2) store the prefix path $prefixPath$ of v as its node type T in a global hash table, so that any two nodes sharing the same $prefixPath$ have the same node type; (3) at the same time we incrementally compute three basic statistics: $tf(k,T)$ and G_T (in Formula 2), N_T (in Formula 3), $f^T k_i, k_j$ and f_k^T (in Formula 7). In addition, several indices are built in order to speedup the computation of the ranks of the refined queries at the end of each refinement algorithm proposed in section VI. Those statistics data may need to be collected via multiple traversal of XML document. Note that all the indices built in this paper are stored in Berkeley DB [24].

The first index built is called keyword inverted list, which retrieves a list of XML nodes v in document order, each of which contains the input keyword in either their tag name or value term, and is in form of : $<DeweyID, prefixPath>$. It provides the following two operations:

- getDeweyID(v,k) returns the Dewey label of the node v.
- getPrefix(v,k) returns the prefix path of v in XML data.

The second index built is called *frequency table*, which stores both the XML document frequency f_k^T and the XML term frequency $tf(k,T)$, for each combination of keyword k and node type T in XML document, and in the worst case its space complexity is O($T*K$) where K is the number of distinct keywords and T is the number of node types in XML data; while for real dataset which has well organized structures, the size of frequency table is comparable to that of the keyword inverted lists. A Berkeley DB B+ tree index is built on the keyword, so it costs O(log$K$) for index index lookup, and supports the fetch of f_k^T.

The third index built is called *co-occur frequency table*, which stores the co-occurrence frequency f_{k_i,k_j}^T for each combination of keyword pair $< k_i, k_j >$ and node type T in XML document, and its worst case space complexity is O(K^2*T). A B+-tree index is built to support efficient fetching of f_{k_i,k_j}^T.

VIII. Experiments

In this section, we investigate the efficiency and scalability of three refinement algorithms (i.e. stack-refine, short-list eager and Partition) proposed in section VI, and the effectiveness of our query ranking model proposed in section IV.

Equipment. All experiments are performed on a 1.9 GHz AMD DualCore PC running Windows XP with 3GB memory. All codes are implemented in Java, and Berkeley DB Java Edition [24] is used to store the keyword inverted lists.

Data set and Query Set. Two real data sets DBLP [25] (420MB, up to 2007/12/10) and Baseball[3] are used.

TABLE III
Sample Query Sets for Term Deletion

ID	Original Query	Suggested Refinements	Size
Q_{D1}	Ling,Tok,Wang,twig,pattern,join	delete "pattern" or "join"	2+5
Q_{D2}	Yufei,Tao,skyline,2000	delete "2000"	5
Q_{D3}	Tan,Kian,Lee,keyword,search	delete "keyword"	8
Q_{D4}	XML,view,model,1995	delete "XML" or "1995"	4+8
Q_{D5}	XML,graph,keyword,search	delete "XML" or "graph"	1+22
Q_{D6}	Ooi,Beng,Chin,Jagadish,index	delete "Jagadish" or "index"	8+11
Q_{D7}	Yannis,graph,keyword,search	delete "Yannis" or "graph" or "keyword"	1+10+1

TABLE IV
Sample Query Sets for Term Merging

ID	Original Query	Suggested Refinements	Size
Q_{M1}	Jia,wei,han,2006	Jiawei	35
Q_{M2}	Xiao,fang,zhou,2005	Xiaofang	16
Q_{M3}	on,line,news,paper	online,newspaper	6
Q_{M4}	electronic,text,book	textbook	6
Q_{M5}	xml,key,word,search	keyword	21
Q_{M6}	online,hand,writing	handwriting	47
Q_{M7}	work,shop,data,management,korea	workshop	2
Q_{M8}	net,work,routing,protocol	network	59
Q_{M9}	micro,array,gene,classification,selection	microarray	21
Q_{M10}	over,lay,routing,cost	overlay	3

TABLE V
Sample Query Sets for Term Split

ID	Original Query	Suggested Refinements	Size
Q_{P1}	adhoc,search	ad,hoc	14
Q_{P2}	webpage,filtering,2006	web,page	2
Q_{P3}	fulltext,search,networks	full,text	3
Q_{P4}	floatingpoint,function	floating,point,function	10
Q_{P5}	multiquery,processing	multi,query	24
Q_{P6}	realtime,application,analysis	real,time	11
Q_{P7}	hengtao,shen,video,2007	heng,tao	5

TABLE VI
Sample Query Sets for Term Substitution

ID	Original Query	Suggested Refinements	Size
Q_{S1}	Jagadish,VLBD	VLDB	41
Q_{S2}	machin,learning,technique	machine	9
Q_{S3}	Jim,Gary,VLDB	Gray	8
Q_{S4}	principle,component,neural,network	principal	18
Q_{S5}	xml,document,object,model	xml,DOM	11
Q_{S6}	extensible,markup,language,application	XML,application	71
Q_{S7}	privacy,preserving,cluster	clustering	24
Q_{S8}	fuzy,database,search	fuzzy	4
Q_{S9}	DASFA,2007,XML	DASFAA	11
Q_{S10}	distributed,allocation,chanel	channel	42
Q_{S11}	search,bundary,constraints	boundary	2

In order to minimize the subjectivity in experimental evaluation, the most recent 1000 queries are selected from the query log of an DBLP online demo[4] of our previous work [22], out of which 219 queries (with an average length of 3.92 keywords) that have empty result are selected to form a pool of queries that need to be refined, which coincides with the primary motivation of this paper. Besides, we randomly pick 100 queries that have matching results and add them into the query pool, in order to increase the variety of queries.

Same as the existing literatures in IR query refinement [13], we build the refinement rule set for the four refinement operations adopted in this paper by asking two human annotators to manually refine the above 219 queries. Regarding to the dissimilarity score ds_r of a refinement rule r, we adopt the

[3] *http://www.ibiblio.org/xml/books/biblegold/examples/baseball/*
[4] *http://xmldb.ddns.comp.nus.edu.sg*

same metrics as described in section III-B, and $ds_r = 2$ is assigned for a single term deletion.

Sample queries with a refinement using a typical operation (but not restricted to a single refinement, as a new refined query can always be produced by term deletion) are shown in Table III-VI, where the 4th column of each table records the result size of the corresponding RQ. In addition, sample queries involving multiple mixed refinements, together with one of the possible refinement solutions are listed as below.

Q_{X1}:{eficient, key, word, search} can be refined by substituting "efficient" for "eficient", followed by a merging of "key" and "word".

Q_{X2}:{eficient, sky, line, computation}, where a desired refinement is {efficient, skyline, computation}.

Q_{X3}:{worldwide, web, search, engine} can be refined by either a split of "worldwide" to "world" and "wide", or substituting "worldwide web" for "www", etc.

Q_{X4}:{inproceeding, xml, twig, match} can be refined by substituting "inproceedings" for "inproceeding", "matching" for "match".

Notations. To facilitate our discussion, some common notations are introduced. (i) The stack-based, short-list eager and partition-based algorithm proposed in section VI are called stack-refine, SLE and Partition respectively. (ii) stack-slca (scan-slca) refers to the stack (scan-eager) approach proposed in [3] for SLCA computation. Note that both Partition and SLE employ the scan-eager approach in computing SLCA results.

A. Efficiency

In this section, we evaluate the query response time of the above three approaches, which is measured by the timestamp difference between a query is issued and its Top-K RQs with their associated SLCA results are returned.

1) Efficiency on sample queries: We first evaluate the efficiency of the three proposed algorithms in Top-1 query refinement on all sample queries in Table III-V plus Q_{X1}-Q_{X4}, and compare with that of stack-slca and scan-slca in answering the original query. Figure 4 shows the time for all sample queries in Table III-V plus Q_{X1}-Q_{X4} on hot cache. We have the following observations.

(1) Partition outperforms both stack-refine and SLE for almost all queries. (2) Stack-refine is the least efficient one, because for each node n visited, it has to find the current optimal RQ that has SLCA result within n. (3) Compared with scan-eager, Partition spends 30% extra time on average in deciding the Top-1 RQ and generating its SLCA results. Considering the fact that the average result size of each RQ is greater than 10 (check the 4th column of Table III-V), the refinement is worthwhile. (4) SLE does not outperform stack-refine too much as expected, because SLE adopts scan-eager for SLCA computation. (5) For Q_{M10}, Q_{S3} and Q_{D7}, Partition is even more efficient than stack-slca and scan-slca which only compute the SLCAs of original query, because the extra cost spent by Partition on computing the rank of RQ candidates is even smaller than the cost saved on SLCA computation leading to the root node. (6) For queries having spelling errors

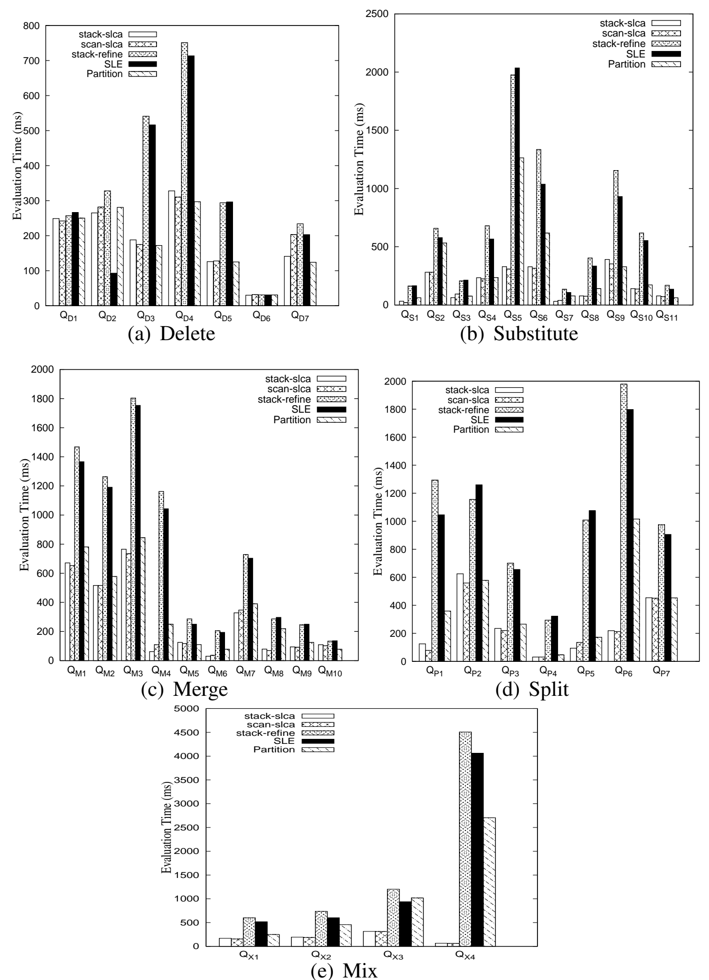


Fig. 4. Top-1 sample query refinement on DBLP

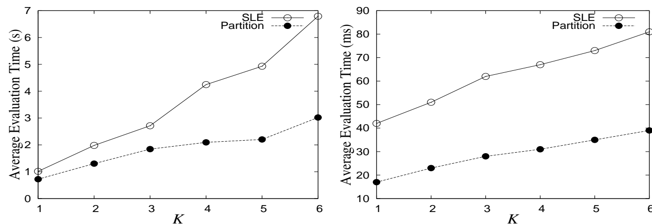
or term mismatch such as Q_{S1} and Q_{X4}, both stack-slca and scan-slca cost much less time, as no or few SLCA computation is needed. (7) SLE outperforms Partition for Q_{D2} and Q_{X3}, because the keyword with the shortest list is also in Top-1 RQ. Lastly, the results of the RQ generated by Stack, Partition and SLE over all queries are verified to be correct.

B. Scalability

In order to test the scalability of SLE and Partition in Top-K query refinement, we design two experiments.

Firstly, we measure the effects of different choices of K on the evaluation time of Top-K query refinement, where K∈[1,6]. A batch of 40 random queries with an average length of 3.71 for DBLP and 20 random queries with an average length of 3.48 for Baseball (with diverse refinements) are tried, and the average time of those queries in five executions are recorded in in Figure 5. As evident from Figure 5(a), Partition scales well (i.e. its time increases slowly), while SLE's time increases much faster when K>3. Because SLE has to find all Top-K RQs before evaluating them, so the larger the K is, the more extra time on dissimilarity computation is, and more times of keyword lists scan are needed in employing existing algorithm to find SLCA results. In contrast, for Partition

approach, the higher the K is, the more possible that the lower-ranked RQs and their SLCA results (that are detected before the higher-ranked queries) are preserved (rather than cleared away), so less extra cost is introduced. Besides, both algorithms scale well on Baseball, as shown in Figure 5(b).



(a) DBLP (b) Baseball

Fig. 5. Effects of K on Top-K Query Refinement

Secondly, we measure the response time of Top-3 query refinement by SLE and Partition over data sets of different size, which are obtained from DBLP, and a batch of 40 queries same as the first experiment are used. As shown in Figure 6, both have a good scalability over data size. Besides, SLE has a significant increase from 60% to 80%, as SLE's efficiency relies heavily on how early the Top-K RQs are detected, the earlier, the less random access to inverted lists caused.

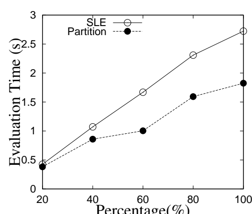


Fig. 6. Effects of data size on Top-3 RQ Computation

C. Effectiveness of Query Refinement

In this section, we evaluate the effectiveness of the query ranking model proposed in section IV.

1) Evaluation method: We first give a brief introduction on the evaluation method we adopt. In state of the art database keyword search [5], [26], [6], some traditional IR evaluation methods have been adopted for their effectiveness evaluation, such as precision, recall, F-measure, reciprocal rank etc. However, all these methods are based on a binary judgement (which judges a result to be either relevant or irrelevant). *Cumulated Gain-based evaluation* (CG) [27], which is popularly used in query refinement works in IR field [13], is proposed to take into account of the fact that all documents are not of equal relevance to users, and combines the degree of relevance of documents and their rank (affected by their possibility of relevance) in a coherent way, no matter what the recall base size is. In particular, given a ranked result list retrieved by search engine, [27] turns the list to a gained value vector G[i], which denotes the relevance score of the ith result retrieved; then a cumulated gain vector CG is defined recursively as shown in Formula 12, where CG[i] is computed by summing G[1] to G[i]. Discounted CG (DCG) is designed to model user persistence to weigh down the gain from results found later in examining long ranked result lists, see [27] for details. In our experiment, we adopt CG rather than DCG, as the ranked query list is usually not too long and all users participated in experiment are patient.

$$CG[i] = \begin{cases} G[i] & \text{if } i = 1 \\ CG[i-1] + G[i] & \text{otherwise} \end{cases} \quad (12)$$

2) Effectiveness test: For each query tested, we extract its Top-4 RQs. 6 researchers in computer science are called up for relevance judgement of query refinement on DBLP, as DBLP is one of the few large real XML data sets, and all the judges use DBLP to find papers frequently, which helps make their judgement more reliable. They are asked to look into each RQ and its matching results carefully, and judgements are done on a four-point scale as: (1) irrelevant, (2) marginally relevant (when few results of RQ partially match the search intention), (3) fairly relevant (when some few results of RQ fully match the intention), (4) highly relevant (when almost all results of RQ contain the themes of topic exhaustively). As mentioned in [27], a proper choice of relevance score depends on the evaluation context. Thus we use *moderate relevance scores* (say, 0-1-2-3) for the above four-point scale in experiment, as our users are assumed to be patient to dig down the results of low-ranked RQs.

TABLE VII

TOP-4 RANKED RQS WITH THEIR RESULT NUMBER

Q	RQ_1	RQ_2	RQ_3	RQ_4
Q_{M1}	jiawei,h,2006; 35	h,w,2006; 45	j,w,2006; 29	h,j,2006; 9
Q_{M2}	xiaofang,z,2005; 16	xiaofang,z; 91	x,z,2005; 27	f,z,2005; 7
Q_{M9}	microarray,g,c,s; 21	microarray,g,s; 60	array,g,c,s; 2	m,a,c,s; 1
Q_{S3}	J,Gray,VLDB; 8	J,Gary;21	J,VLDB; 11	G,VLDB; 4
Q_{S5}	XML,DOM; 11	d,o,m;9	XML,o,m; 5	XML,d,m;8
Q_{S6}	XML,a; 71	m,l,a;6	e,m,l; 22	l,a; 189
Q_{P3}	full,text,s,n; 3	t,s,n; 7	f,t,n; 5	f,s,n; 3
Q_{P6}	real,time,ap,an; 11	ap,an; 1187	realtime,ap;5	realtime,an; 2
Q_{X1}	efficient,keyword,s; 19	efficient,k,s; 4	word,s; 21	key,w,s; 1
Q_{X2}	efficient,skyline,c; 8	skyline,c; 13	eff,skyline; 17	efficient,l,c;4
Q_{X3}	world,wide,w,s,e;9	www,s,e;39	web,s,e;156	w,w,w,s;43

Table VII shows the Top-4 RQs with their matching result number generated by our query ranking model (i.e. Formula 10 where $\alpha = \beta = 1$), for some sample queries in Table III-VI. For simplicity, each keyword is denoted by its first letter if no ambiguity is caused. For each query in Table VII, all 6 judges have an agreement that the rank-1 refined query RQ_1 is the most appropriate refinement.

Next, we make an in-depth analysis of the query ranking model. As the overall rank of a RQ consists of two complementary parts, i.e. *similarity score* and *dependence score*, we conduct two sets of experiments to test their respective effects.

In the *first* experiment, we investigate the query ranking model that takes the similarity score into account alone. In particular, the effects of the four guidelines that constitute the similarity score are tested respectively. Let RS_0 denote the original ranking scheme, and RS_i denote a variant of RS_0 by removing `Guideline` i from consideration for $i \in [1,4]$. In our experiment, we adopt the above CG evaluation but the input now is a ranked list of RQs (associated with their matching results). 50 queries that have no meaningful results on DBLP, involve various refinement(s) and have at least 4 possible RQ candidates are chosen from our query pool. Table

VIII shows a summary of the number of queries that involve the four refinement operations adopted in this paper. The decay factor w in Formula 6 is set to 0.7.

TABLE VIII
QUERY STATISTICS

Refinement	Num
Term Merging	18
Term Split	10
Term Substitution	31
Term Deletion	4

TABLE IX
CG@4 BY DIFFERENT RANKING MODELS

Variants	$CG[1]$	$CG[2]$	$CG[3]$	$CG[4]$
RS_0	2.631	3.562	4.233	4.539
RS_1	2.343	3.491	4.127	4.516
RS_2	2.416	3.525	4.161	4.525
RS_3	2.427	3.509	4.058	4.497
RS_4	2.305	3.456	4.16	4.521

Table IX shows the results of average CG values judged by the above 6 users for Top-K RQs, for K=1 to 4. *(1)* From column 1 of Table IX, we find the original ranking model, i.e. RS_0 is the most effective one that can capture the most relevant result as Top-1 RQ, compared to all its four variants. *(2)* By comparing CG[i] of each approach for $i\in$[0,4], we find the original ranking model outperforms all four variants in finding the Top-K RQs for any $K\in$[1,4]. *(3)* In finding the Top-1 RQ, Guideline 4 plays a more important role than other guidelines, as GC[1] of RS_4 has the smallest value. *(4)* From last column of Table IX, we find both the original ranking model and its four variants have similar value for CG[4], which means all of them are able to find the desired Top-4 RQs, although the relative rank of these RQs vary in each variant.

TABLE X
CG@4 BY DIFFERENT WEIGHTS

[α,β]	CG[1]	CG[2]	CG[3]	CG[4]
[1,2]	2.626	3.56	4.217	4.532
[2,1]	2.64	3.565	4.241	4.537
[1,1]	2.675	3.569	4.236	4.543
[1,0]	2.631	3.562	4.233	4.539

In the *second* experiment, we test a combined effect of similarity score and dependence score of a RQ. The importance of these two factors are investigated by varying the choice of the tunable parameters α and β in Formula 10. From the result shown in Table X, we have the following observations. (1) By comparing variants [1,1] and [1,0], we find the consideration of the dependence score does improve the overall effectiveness of our query ranking model. (2) By comparing the CG[1] for all variants, we find the similarity score is more effective than the dependence score in inferring the Top-1 RQ.

IX. CONCLUSION AND FUTURE WORK

In this paper, we introduce the problem of automatic XML keyword query refinement, and integrate the job of finding the desired refined queries and generating their matching results as a single problem, with no intervention on user part. We first describe what a meaningful matching result should be, and introduce the concept of dissimilarity as a preliminary quality metric of a refined query RQ. Then we propose an in-depth query ranking model which takes into account of both the keyword dependencies in RQ and the relevance of RQ w.r.t original search intention. Next, we design a dynamic programming method to find the optimal RQ in term of its dissimilarity, and further propose three efficient query refinement algorithms that need only one-time scan of the keyword inverted lists. Lastly, experiments show the efficiency and effectiveness of our approach. In future, we would like to study another extreme of our work - how to refine a query which has "too many" matching results over XML data.

REFERENCES

[1] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked keyword search over XML documents," in *SIGMOD*, 2003.
[2] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSEarch: A semantic search engine for XML," in *VLDB*, 2003.
[3] Y. Xu and Y. Papakonstantinou, "Efficient keyword search for smallest LCAs in XML databases," in *SIGMOD*, 2005.
[4] Y. Li, C. Yu, and H. Jagadish, "Schema-free XQuery," in *VLDB 2004*.
[5] Z. Liu and Y. Chen, "Identifying meaningful return information for xml keyword search," in *SIGMOD*, 2007.
[6] Z. Bao, T. W. Ling, B. Chen, and J. Lu, "Effective xml keyword search with relevance oriented ranking," in *ICDE*, 2009.
[7] D. C. Fain and J. O. Pedersen, "Sponsored search," in *Bulletin of the American Society for Information Science and Technology*, 2005.
[8] C. Sun, C. Y. Chan, and A. K. Goenka, "Multiway slca-based keyword search in xml data," in *WWW*, 2007.
[9] R. Jones, B. Rey, O. Madani, and W. Greiner, "Generating query substitutions," in *WWW*, 2006.
[10] K. Q. Pu and X. Yu, "Keyword uery cleaning," in *VLDB*.
[11] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava, "Keyword proximity search in XML trees," in *TKDE*, 2006.
[12] J. Xu and W. B. Croft, "Improving the effectiveness of information retrieval with local context analysis," *ACM Trans. Inf. Syst.*, 2000.
[13] J. Guo, G. Xu, H. Li, and X. Cheng, "A unified and discriminative model for query refinement," in *SIGIR*, 2008.
[14] I. Ruthven, "Re-examining the potential effectiveness of interactive query expansion," in *SIGIR*, 2003.
[15] Y. Mass and M. Mandelbrod, "Component ranking and automatic query refinement for xml retrieval," in *INEX*, 2004.
[16] H. Pan, A. Theobald, and R. Schenkel, "Query refinement by relevance feedback in an xml retrieval system," in *ER*, 2004.
[17] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum, "Topx: efficient and versatile top-k query processing for semistructured data," *The VLDB Journal*, vol. 17, no. 1, 2008.
[18] C. Fellbaum, "Wordnet: an electronic lexical database."
[19] V. Vesper., "Let's do dewey," http://www.mtsu.edu/ vvesper/dewey.html.
[20] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.
[21] R. Jones and D. Fain, "Query word deletion prediction," in *SIGIR03*.
[22] Z. Bao, B. Chen, T. W. Ling, and J. Lu, "Demonstrating effective ranked xml keyword search with meaningful result display," in *DASFAA*, 2009.
[23] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD*, 1993.
[24] "Berkeley DB," http://www.sleepycat.com/.
[25] M. Ley, "http://www.informatik.uni-trier.de/ ley/db/."
[26] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: Efficient and adaptive keyword search on unstructured, semi-structured and structured data," in *SIGMOD*, 2008.
[27] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Trans. Inf. Syst.*