World Wide Web (2018) 21:1223–1257
https://doi.org/10.1007/s11280-017-0503-8

No-but-semantic-match: computing semantically matched xml keyword search results

Mehdi Naseriparsa[1] · Md. Saiful Islam[2] · Chengfei Liu[1] · Irene Moser[1]

Received: 26 April 2017 / Revised: 8 August 2017 / Accepted: 2 October 2017 /
Published online: 13 October 2017

Abstract Users are rarely familiar with the content of a data source they are querying, and therefore cannot avoid using keywords that do not exist in the data source. Traditional systems may respond with an empty result, causing dissatisfaction, while the data source in effect holds semantically related content. In this paper we study this no-but-semantic-match problem on XML keyword search and propose a solution which enables us to present the top-k semantically related results to the user. Our solution involves two steps: (a) extracting semantically related candidate queries from the original query and (b) processing candidate queries and retrieving the top-*k* semantically related results. Candidate queries are generated by replacement of non-mapped keywords with candidate keywords obtained from an ontological knowledge base. Candidate results are scored using their cohesiveness and their similarity to the original query. Since the number of queries to process can be large, with each result having to be analyzed, we propose pruning techniques to retrieve the top-*k* results efficiently. We develop two query processing algorithms based on our pruning techniques. Further, we exploit a property of the candidate queries to propose a technique for processing multiple queries in batch, which improves the performance substantially. Extensive experiments on two real datasets verify the effectiveness and efficiency of the proposed approaches.

✉ Mehdi Naseriparsa
mnaseriparsa@swin.edu.au

Md. Saiful Islam
saiful.islam@griffith.edu.au

Chengfei Liu
cliu@swin.edu.au

Irene Moser
imoser@swin.edu.au

[1] Swinburne University of Technology, Melbourne, Australia

[2] Griffith University, Gold Coast, Australia

Keywords XML keyword query · No-match · Semantics · Ontology

1 Introduction

Users who query data sources using keyword searches often are not familiar with the data source schema or the appropriate query language. For the query to succeed, the keywords have to have matches in the data source. Failing this, an empty result is returned even when semantically related content exists. When keywords have indirect mappings in a data source that cannot be found by traditional systems, the user faces the no-but-semantic-match problem.

Example 1 Consider a user submitted a keyword query $q_0 = \{Jack, lecturer, class\}$ on XML database given in Figure 1 and would like to find information about the professor $Jack$. Using conjunctive keyword search, traditional systems will show an empty result because there is no occurrence for the keywords $lecturer$ and $class$ in the data source. However, the keyword $lecturer$ has a semantic connection to $academic$ and $full\ professor$ while the keyword $class$ is semantically related to $course$, $grade$ and $event$ which exist in the data source and could generate results that might interest the user.

The XML keyword search has been addressed by researchers before. The concept of Lowest Common Ancestor (LCA) was first proposed by Guo et al. [16] to extract XML nodes which contain all query keywords within the same subtree. Xu and Papakonstantinou [40] introduced the concept of Smallest Lowest Common Ancestor (SLCA) to reduce the query result to the smallest tree that contains all keywords. Sun, Chan and Goenka [35] extended this work by applying the SLCA principle to logical OR searches. Hristidis et al. [18] explored the trees below LCA to provide information about the proximity of the keywords in the document. Zhou et al. [43] proposed a novel form of inverted list, namely IDList and set intersection problem for processing XML keyword queries efficiently. None of the existing studies use the SLCA semantics to provide a solution when one or more keywords do not exist in the database. In this paper, we adapt the widely-accepted SLCA semantics and algorithms to retrieve meaningful results when some non-mapped keywords are submitted to the system.

When a query encounters the no-but-semantic-match problem, we need to find candidate keywords for the non-mapped keywords to produce non-empty results. Even though the non-mapped keywords may be semantically close to some items in data source, traditional systems do not attempt to discover them. To produce an answer to the user's initial query, the

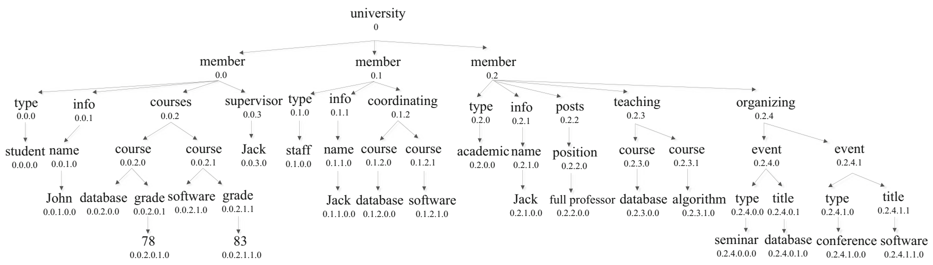


Figure 1 A part of XML data

candidate keywords must be semantically close to the non-mapped keywords. One way of fulfilling this requirement is to find substitutes for non-mapped keywords in an ontological knowledge base. Clearly, only candidate keywords that have a mapping in the data source can be selected as substitutes for a new query. Replacing each of the non-mapped keywords with one or more semantically related words that are known to exist in the database leads to a list of candidate queries. Depending on the number of available keywords, the number of potential queries and results can be impractically large. Hence the degree of semantic similarity with the original query is calculated for each candidate query before it is executed. Before the results can be presented to the user, results of poor quality in terms of cohesiveness must be eliminated to ensure all results are meaningful answers to the original query. Thus, to solve the no-but-semantic-match problem, two aspects are considered: (a) query similarity; and (b) result cohesiveness.

Example 2 Consider the keyword query $q_0 = \{Jack, lecturer, class\}$ presented in Example 1 on the database shown in Figure 1. Keywords $lecturer$ and $class$ do not have a mapping in the data source and the traditional system generates an empty result for it. The ontological knowledge base [29] has 44 semantic counterparts for $lecturer$ and 39 for $class$. All possible substitutions and their combinations are considered. In the extreme case when all candidate keywords are available in the data source, $44 \times 39 = 1716$ queries are generated and each query may have several answers that have to be considered. When a high number of keywords have to be replaced and these keywords have many semantic counterparts, we may face an unmanageably large number of combinations that have to be analyzed for semantic similarity with the original query. Hence, there is a need to identify and remove less promising candidate queries early.

In this paper, we present a novel two-step solution to the no-but-semantic-match problem in XML keyword search. In the first step, semantically related candidate queries are created by replacing non-mapped keywords in the original queries with semantic counterparts and in the second step, the queries are processed and the top-k semantically related results retrieved. In order to present the top-k results to the user for evaluation, each result retrieved from the queries is separately analyzed in terms of its similarity to the original query and its cohesiveness in data source. Since there may be a large number of semantically related results, retrieving the top-k results is potentially costly. Therefore, we propose two pruning techniques, inter-query and intra-query pruning. Since the candidate queries are generated by replacing non-mapped keywords, some keywords are shared between the candidate queries. We exploit this property to propose a more efficient batch query processing technique to improve the performance substantially. The issue of finding semantically related results for queries with no-but-semantic-match problem has not been addressed in the context of semi-structured data before. Our contributions are as follows:

1. We are the first to formulate the no-but-semantic-match problem in XML keyword search.
2. We propose two pruning methods and an efficient approach of processing the no-but-semantic-match query.
3. Based on keywords the candidate queries have in common, we also propose a method to process multiple queries in a batch which improves the performance substantially.
4. We conduct extensive experiments which verify the effectiveness and efficiency of our solutions on two real datasets.

The rest of the paper is organized as follows: Section 2 discusses XML keyword search and presents the no-but-semantic-match problem. Section 3 presents the details of our pruning ideas and the efficient processing of the no-but-semantic-match query. Section 4 presents the batch query processing scheme to further improve the performance. The experiments are presented in Section 5. Section 6 reviews the related work. Finally, Section 7 concludes our paper.

2 Background

2.1 Preliminaries

An XML document is an ordered tree T with labeled nodes and a designated root. All XML elements are treated as nodes containing information in T. There are parent-child and sibling relationships between the nodes. The depth of the tree is denoted as d, and the root node has a depth of 1. Each node v in the tree T is marked with a unique identifier in Dewey code, which describes the path from the root to the node v as a sequence of numbers separated by a dot ("."). Sibling nodes have Dewey codes of equal length with a unique last number.

Example 3 Figure 1 shows an XML tree which contains information about staff and students of a university. The root node's Dewey code is 0. Dewey code 0.0.1 refers to a node containing information about a member of the university and the code prefix 0.0 refers to its parent node.

Keyword match node A node m in the tree T is a match node for keyword k_i if it contains k_i. e.g., the match nodes for keyword $k_1 = database$ presented in Figure 1 are: $m_1^1 =$ [0.0.2.0.0], $m_1^2 =$ [0.1.2.0.0], $m_1^3 =$ [0.2.3.0.0], and $m_1^4 =$ [0.2.4.0.1.0].

Keyword inverted list Each keyword k_i corresponds to a list S_i of entries and each entry corresponds to a node m which contains k_i in the tree T. e.g., the keyword inverted list for keyword $k_1 = database$ is $S_1 =$ {[0.0.2.0.0], [0.1.2.0.0], [0.2.3. 0.0], [0.2.4.0.1.0]}.

Smallest Lowest Common Ancestor (SLCA) Let $lca(m_1, ..., m_n)$ returns the lowest common ancestor (LCA) of match nodes $m_1, ..., m_n$. Then LCAs of query q on T are defined as $LCA(q) = \{v|v = lca(m_1, ..., m_n), m_i \in S_i (1 \le i \le n)\}$. SLCAs are a subset of LCAs which do not have other LCAs as descendant nodes and defined as $SLCA(q)$.

Example 4 In Figure 2, for a keyword query q={$Jack$, $database$}, there are 4 LCA nodes which are computed as: $LCA(q) = \{lca($[0.0.3.0], [0.0.2.0.0]$)$, $lca($[0.0.3.0], [0.1 .2.0.0]$)$, $lca($[0.1.1.0.0], [0.1.2.0.0]$)$, $lca($[0.2.1.0.0], [0.2.3.0.0]$)\}$={[0], [0.0], [0.1] , [0.2]}. Since the LCA node [0] is the ancestor node of [0.0], [0.1] and [0.2], it is not an SLCA and should be removed. Therefore, $SLCA(q) =$ {[0.0], [0.1], [0.2]}.

Keyword query and subtree result In XML data, a keyword query q consists of a set of keywords $\{k_1, k_2, ..., k_n\}$. A result $r = (v_{slca}, \{m_1, m_2, ..., m_n\})$ for q is a subtree in T which contains all keywords $k_i \in q$. Here, we consider v_{slca}, the root of the subtree, an SLCA node, i.e. $v_{slca} \in SLCA(q)$.

Tightest SLCA subtree result For an SLCA node, there may exist several subtree results. This is because that under an SLCA node v_{slca}, we may find several match nodes $\{m_i^j\}$ for

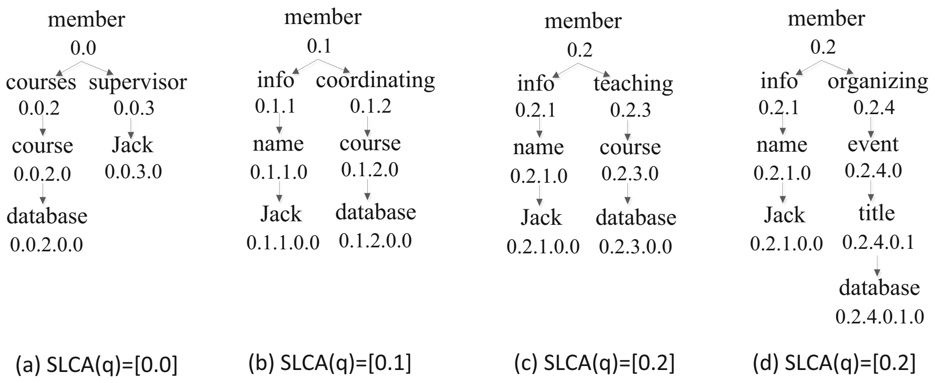


Figure 2 SLCA subtree results for query $q = \{Jack, database\}$ executed on data given in Figure 1

the keyword k_i, $(1 \le i \le n, 1 \le j \le n_i)$, where n_i is the number of match nodes for k_i under v_{slca}. Let $m_i^{l_i}$ be the closest match node from $\{m_i^j\}$ to v_{slca} for $k_i (1 \le i \le n)$, then we get the the tightest subtree result $r = (v_{slca}, \{m_1^{l_1}, ..., m_i^{l_i}, ..., m_n^{l_n}\})$.

For example, for $SLCA(q) = [0.2]$ in Figure 2, there are two subtree results, (c) and (d). The tightest subtree result is (c) $r =$ ([0.2], {[0.2.1.0.0], [0.2.3.0.0]}).

We argue to return only the tightest SLCA subtree results to the user as these results match the user's search intention better than the results containing the sparsely distributed keyword match nodes under v_{slca}. That is, a result is more likely to be meaningful when the result subtree is more tight and cohesive (for survey [15, 18]).

2.2 Problem statement

Definition 1 (**No-Match Problem**) Given a keyword query $q_0 = \{k_1, k_2, ... ,k_n\}$ on T, if $\exists$ $k_i \in q_0$ such that $S_i = \emptyset$, we say that query q_0 has a no-match problem over k_i.

If a user submits a keyword query q_0 that has a no-match problem, traditional systems return an empty result set. However, the missing keyword k_i that causes the no-match problem may have semantic counterparts in the data source T which may produce results the user might be interested in, if the candidate keywords are sufficiently similar to $k_i \in q_0$. We use $\mathcal{K}_i$ to denote the list of candidate keywords that can be used to replace $k_i \in q_0$.

Example 5 Consider the keyword query $q_0 = \{Jack, lecturer, class\}$ presented in Example 1. It is easy to verify that the keywords $k_2 = lecturer$ and $k_3 = class$ cause a no-match problem for q_0. Candidate keywords that can be used instead of k_2 and k_3 for q_0 are: $\mathcal{K}_2 = \{academic, full\ professor\}$ and $\mathcal{K}_3 = \{course, grade, event\}$.

Definition 2 (**No-But-Semantic-Match Problem**) Given a keyword query $q_0 = \{k_1, k_2, ... ,k_n\}$ with a no-match problem on T, i.e., $\exists k_i \in q_0$ such that $S_i = \emptyset$, but $\mathcal{K}_i \neq \emptyset$, then we say that q_0 has a no-but-semantic-match problem over k_i.

The no-but-semantic-match problem is a special case of the no-match problem. The problem can be addressed in the following way: (a) find a candidate keyword list $\mathcal{K}_i$ that can be used to replace $k_i \in q_0$; (b) generate candidate queries q' for q_0 by replacing k_i with

Table 1 The list of symbols

Symbol	*Meaning*
$\lambda(q_0, q')$	Similarity score of q_0 to q'
α	Tuning parameter
σ^{min}	Threshold score
$\sigma(r, q', T)$	Total score of a result
q_0	User original query
q'	A candidate query
$\mathcal{Q}$	A set of candidate queries
$\mathcal{S}$	A set of Keyword Inverted lists
$\mathcal{B}$	Candidate query batch
$\mathcal{R}$	A set of results
$\mathcal{R}^*$	A set of top-k results
$\Delta(r_1, r_2)$	The score difference between r_1 and r_2
r	A result
v_{slca}	A subtree result root
m	A match node
n	Number of keywords in a query
m^l	Tightest match node
$\mathcal{P}$	An execution plan
$c(\mathcal{B})$	Cost of an execution plan
T	XML data
$d(r, T)$	Number of edges in a result r
$\theta(r, T)$	Cohesiveness score of a result r
$\mathcal{K}$	A set of candidate keywords
k	A query keyword

$k'_i \in \mathcal{K}_i$; (c) execute q' in the data source T to produce the semantically related results $\mathcal{R}$ for q_0; (d) score and rank the results $r \in \mathcal{R}$ to return only the top quality results to the user for evaluation. The list of the symbols is presented in Table 1.

Example 6 Consider the keyword query presented in Example 1. The query q_0 has no-but-semantic-match problem over $k_2 = lecturer$ and $k_3 = class$. The semantic counterparts for k_2 and k_3 are: $\mathcal{K}_2 = \{academic, full\ professor\}$ and $\mathcal{K}_3 = \{course, grade, event\}$. These candidate keywords are combined with the rest of the keywords to generate semantically related candidate queries for q_0. The generated candidate queries are: $q_1 = \{Jack, academic, course\}$, $q_2 = \{Jack, academic, grade\}$, $q_3 = \{Jack, full\ professor, course\}$, $q_4 = \{Jack, full\ professor, grade\}$, $q_5 = \{Jack, academic, event\}$, and $q_6 = \{Jack, full\ professor, event\}$.

We use $\mathcal{Q}$ to denote the list of candidate queries. As mentioned before, the candidate queries $q' \in \mathcal{Q}$ need to be executed against T to produce the semantically related result set $\mathcal{R}$ for q_0. We know that these candidate queries q' are generated by replacing k_i with $k'_i \in \mathcal{K}_i$. However, not all candidate keywords $k'_i \in \mathcal{K}_i$ are semantically similar to the user given keyword $k_i \in q_0$ and also, not all semantically related results $r \in \mathcal{R}$ are meaningful to the same degree. Therefore, we need to score the produced results $r \in \mathcal{R}$, denoted by $\sigma(r, q', T)$, as given as follows:

$$\sigma(r, q', T) = sim(q_0, q') \times coh(r, T) \tag{1}$$

where, r is a result for the candidate query q', the $sim(q_0, q')$ measures the similarity of q' with q_0 (based on (6)) and $coh(r, T)$ measures the cohesiveness of r in T (based on (8)). The rank of a result $r \in \mathcal{R}$ is calculated as follows:

$$rank(r, T) = |\{r' | \sigma(r', q'', T) > \sigma(r, q', T)\}| + 1 \tag{2}$$

Definition 3 Top-k Semantically Related Results Given a keyword query $q_0 = \{k_1, k_2, ..., k_n\}$ on T, having the no-but-semantic-match problem, we want to discover k results from $\mathcal{R}$ that maximize the scoring function given in (1) or in terms of ranking the results $\{r | rank(r, T) \leq k\}$.

3 Our approach

We propose a two phase approach to solve the no-but-semantic-match problem in XML data T. The schematic diagram of our approach is illustrated in Figure 3. In the first phase, the semantic counterparts for the non-mapped keywords of the user query are extracted from the ontological knowledge base. Next, the candidate queries are generated by replacing the non-mapped keywords with their semantic counterparts and the similarities between the candidate queries and the original query are computed. In the second phase, the candidate queries are executed against the data source T. The results are scored based on (1) and finally, only the top-k results are presented to the user for evaluation. The results are scored based on the followings: (a) similarity of the candidate queries to the user given query; and (b) the cohesiveness of the results.

As the candidate queries are generated using the ontological knowledge base, we use the *ontological similarity* of the candidate query to the user given query as the measure of similarity for the first parameter. The details for computing this similarity is presented in Section 3.1. The details for computing the *cohesiveness* of the results is presented in Section 3.2. Since there are a number of candidate queries that should be executed against the data source T and each candidate query may have several results that needs to be scored, we propose efficient pruning techniques to avoid unnecessary computations and terminate early. The pruning ideas and the details of our candidate query processing technique are presented in Section 3.4. We also propose a *batch* query processing technique to speed up

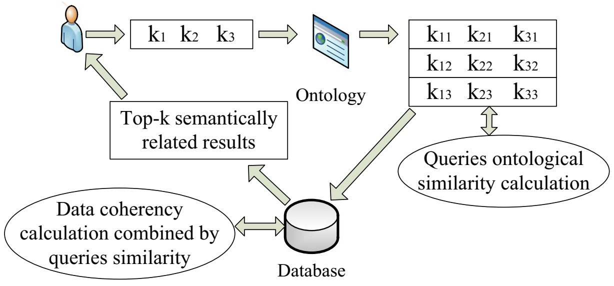

k1 k2 k3
Ontology
k11 k21 k31
k12 k22 k32
k13 k23 k33
Top-k semantically related results
Queries ontological similarity calculation
Data coherency calculation combined by queries similarity
Database

Figure 3 Schematic diagram of our approach for solving no-but-semantic-match problem in XML data

the computations further by sharing the computations among the candidate queries, which is described in Section 4.

3.1 Candidate queries

This section describes how to generate the candidate queries $\mathcal{Q}$ for the user query q_0 and compute their similarity to q_0.

3.1.1 Generating candidate queries

To adhere closely to the user's intentions, the candidate keywords $k'_i \in \mathcal{K}_i$ must be as close as possible to the non-mapped keywords $k_i \in q_0$. In this study we use WordNet, which is widely used in the literature [6] for finding semantic counterparts for $k_i \in q_0$. We categorize our semantic candidate keywords derived from WordNet into four groups [29]: (a) synonyms denoted as $Syn(k_i)$, (b) coordinate terms denoted as $Cot(k_i)$, (c) hyponyms denoted as $Hpo(k_i)$, and (d) hypernyms denoted as $Hpe(k_i)$. The candidate keyword list $\mathcal{K}_i$ for a keyword $k_i \in q_0$ contains all types of ontological counterparts as shown in (3).

$$\mathcal{K}_i = Syn(k_i) \cup Cot(k_i) \cup Hpo(k_i) \cup Hpe(k_i) \tag{3}$$

However, not all candidate keywords extracted from the ontological knowledge base are available in T. To avoid generating non-related queries from the non-existing candidate keywords that may produce non-sense results, the keyword list has to be reduced to the candidates which have direct mapping in the data source. To do this, an inverted keyword list using hash indices can be queried in $\mathcal{O}(1)$ time. Finally, the candidate queries $\mathcal{Q}$ are generated by replacing the non-mapped keywords $k_i \in q_0$ with each of their semantic counterparts $k'_i \in \mathcal{K}_i$ in turn.

3.1.2 Measuring candidate query similarity

In order to measure the similarity between a candidate query $q' \in \mathcal{Q}$ and the user's original query q_0, firstly we measure the individual similarity between the candidate keyword $k'_i \in q'$ and the corresponding non-mapped keyword $k_i \in q_0$ using Wu and Palmer's metric [38]. This metric establishes the depths of both keywords and their least common subsumer (LCS) according to the WordNet structure and produces the degree of similarity between these two keywords, $SimWP(k_i, k'_i)$, as shown in (4).

$$SimWP\left(k_i, k'_i\right) = \frac{2 \times dep(LCS)}{dep(k_i) + dep\left(k'_i\right)} \tag{4}$$

where $dep(k_i)$ returns the depth of the keyword k_i in the WordNet structure. This metric is symmetric, i.e., $SimWP\left(k_i, k'_i\right) = SimWP\left(k'_i, k_i\right)$. However, WordNet has a hierarchical structure. That is, the candidate keyword $k'_i \in q'$ could be a more special type (e.g., hyponyms) or a more general type (e.g., hypernyms) for the non-mapped keyword $k_i \in q_0$ in WordNet. Therefore, we incorporate the specialization/generalization aspect of $k_i \in q_0$ into the Wu and Palmer similarity metric as given as follows:

$$DSim\left(k_i, k'_i\right) = \frac{dep\left(k'_i\right)}{max\left(dep(k_i), dep\left(k'_i\right)\right)} \times SimWP\left(k_i, k'_i\right) \tag{5}$$

where $DSim(k_i, k'_i)$ is the directional similarity of keyword $k_i \in q_0$ to keyword $k'_i \in q'$. The directional similarity $DSim\left(k_i, k'_i\right)$ penalizes the more general keyword types of $k_i \in$

q_0 by weighting $SimWP\left(k_i, k_i'\right)$ with $\frac{dep(k_i')}{max\left(dep(k_i), dep(k_i')\right)}$. Finally, the similarity of q' to the original query q_0 is computed by considering all the replacements in q' as follows:

$$\lambda(q_0, q') = \prod_{i=1}^{n} DSim\left(k_i \in q_0, k_i' \in q'\right) \tag{6}$$

where n is the number of keywords in q_0 that have been replaced to generate q' $(1 \leq n \leq |q|)$.

Example 7 Consider the keyword query $q_0 = \{Jack, lecturer, class\}$ presented in Example 1. Here, the second and the third keywords cause the no-match problem for q_0. The candidate keyword list for these two non-mapped keywords are: $\mathcal{K}_2 = \{academic : 0.91, full\ professor : 0.84\}$ and $\mathcal{K}_3 = \{course : 1, grade : 1, event : 0.35\}$, where each candidate keyword is labeled with their corresponding *DSim* scores. Now, the candidate queries are generated by replacing the non-mapped keywords in q_0 with their candidate keywords and scored as follows:
$q_1 = \{Jack, academic, course\}$, $\lambda(q_0, q_1)$=0.91 $\times$ 1 = 0.91,
$q_2 = \{Jack, academic, grade\}$, $\lambda(q_0, q_2)$=0.91 $\times$ 1 = 0.91,
$q_3 = \{Jack, fullprof, course\}$, $\lambda(q_0, q_3)$=0.84 $\times$ 1 = 0.84,
$q_4 = \{Jack, fullprof, grade\}$, $\lambda(q_0, q_4)$=0.84 $\times$ 1 = 0.84,
$q_5 = \{Jack, academic, event\}$, $\lambda(q_0, q_5)$=0.91 $\times$ 0.35 = 0.31 ,and
$q_6 = \{Jack, fullprof, event\}$, $\lambda(q_0, q_6)$=0.84 $\times$ 0.35 = 0.29.

From the above, it is easy to verify that q_1 and q_2 are the most similar candidate queries to the original query.

3.2 Cohesiveness of results

There can be potentially many candidate queries with a large number of results and not all results $r \in \mathcal{R}$ are meaningful to the same degree. In XML, if the match nodes in the result r are near to each other, the result is considered to be more cohesive. Intuitively, when a result subtree is more cohesive, it is more likely to be relevant and meaningful (for survey [15, 18]). To measure the cohesiveness of a result r, we firstly compute the distance between each match node m and the root v_{slca} of the result subtree r. Then, we compute the overall distance of a result r w.r.t. the data source T as given as follows [15]:

$$d(r, T) = \sum (l_m - l_{v_{slca}}) \tag{7}$$

where r is the result subtree, v_{slca} denotes the root of the result r, m is the match node, l_m is the level of m, and $l_{v_{slca}}$ is the level of the root in r. Equation (7) counts the number of edges from each tightest match node to the root in the subtree result independently. This equation may sometimes count the sharing edges more than one time but can also improve the performance due to its light computations. Moreover, the side effect of sharing path computations independently is minimized since we only consider the tightest match nodes in the subtree result. Furthermore, our solutions are independent of this equation and any kind of equation could be used instead. Clearly, the larger this distance is, the lower the cohesiveness score for the result r should be. Therefore, we compute the cohesiveness of a result subtree r w.r.t. the data source T as given as follows [15]:

$$\theta(r, T) = \frac{1}{\log_\alpha(d(r, T) + 1) + 1} \tag{8}$$

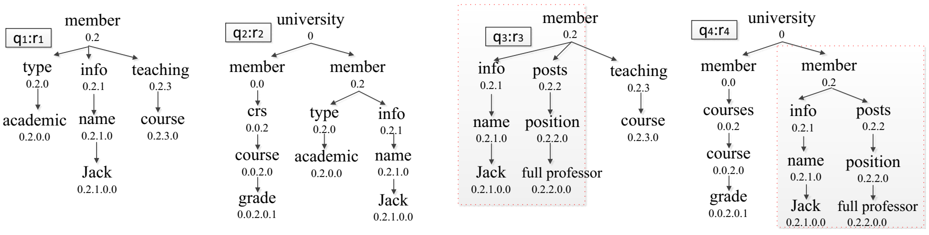


Figure 4 Subtree results for candidate queries $\mathcal{Q}$ in Example 7 after executing them against data given in Figure 1

where α is the tuning parameter by which the user can trade off between the similarity of the candidate queries and the cohesiveness of the results. If we set α to a larger value, the sensitivity to the cohesiveness of the results gets smaller. That is, the total score σ of a result r is more dependent on the similarity of the query $\lambda(q_0, q')$ than its cohesiveness.

Example 8 Consider the candidate keyword queries $\mathcal{Q}$ in Example 7. The result subtrees of these queries are illustrated in Figure 4. Using $\alpha = 4$, we compute the cohesiveness of the results as follows:
$d(r_1, T) = 7, \theta(r_1, T) = \frac{1}{2.5} = 0.4$,
$d(r_2, T) = 10, \theta(r_2, T) = \frac{1}{2.72} = 0.36$,
$d(r_3, T) = 8, \theta(r_3, T) = \frac{1}{2.58} = 0.38$,
and $d(r_4, T) = 11, \theta(r_4, T) = \frac{1}{2.79} = 0.35$.

3.3 Effect of tuning parameter

Now, we provide two fine-grained case studies as follows: (1) the influence of the tuning parameter α on the ranking of the retrieved results and (2) trading off between query similarity and result cohesiveness in the final top-k results based on α.

Case Study-1 This case study demonstrates that our approach is tolerant to the settings of α if one result beats another one in terms of both query similarity and result cohesiveness. This is also expected as the user might explore the top cohesive results with better similarity first. Consider the queries given in Example 7. Here, we extract r_1 from q_1 and r_4 from q_4 as shown in Figure 4. From Table 2, we see that r_1 will always be ranked better than r_4 as r_1 has the higher overall score σ than r_4 for all settings of α.

Case Study-2 This case study demonstrates how the user can trade off between query similarity and the result cohesiveness based on α. Assume a user would like to explore the results with better cohesiveness first than those with higher similarity. Consider a candidate

Table 2 Ranking of r_1 is tolerant to α as it is better than r_4 in terms of both query similarity and result cohesiveness

α	$d(r_1, T)$	$\sigma(r_1, q_1, T)$	$d(r_4, T)$	$\sigma(r_4, q_4, T)$	$\Delta(r_1, r_4)$
2	7	0.2291	8	0.2029	0.0262
3	7	0.3168	8	0.282	0.0348
4	7	0.3666	8	0.3273	0.0393
8	7	0.4583	8	0.4114	0.0469
16	7	0.5238	8	0.472	0.0518

Table 3 Trading off query similarity and result cohesiveness in r_2 and r_7

α	$d(r_2, T)$	$\sigma(r_2, q_2, T)$	$d(r_7, T)$	$\sigma(r_7, q_7, T)$	$\Delta(r_2, r_7)$
2	11	0.1845	7	0.1887	−0.0042
3	11	0.2593	7	0.2608	−0.0015
4	11	0.303	7	0.3019	0.0011
8	11	0.3854	7	0.3774	0.008
16	11	0.4462	7	0.4313	0.0149

query $q_7 = \{Jack, academic, position\}$ with $\lambda(q_0, q_7) = 0.7549$ and a result r_7 from q_7 with $d(r_7, T) = 7$. Now, we compare it with r_2 with $d(r_2, T) = 11$ from q_2 with $\lambda(q_0, q_2) = 0.8462$ as given in Example 7. From Table 3, we observe that r_7 outranks r_2 for $\alpha = [2, 3]$. Now, a user needs to set $\alpha > 3$ to explore r_2 before r_7 in the result list by putting more emphasis on query similarity than result cohesiveness.

3.4 Processing of candidate queries

A naïve approach to processing the no-but-semantic-match query q_0 first generates the candidate queries $\mathcal{Q}$ and then computes all semantically related results $r \in \mathcal{R}$ by executing the queries $q' \in \mathcal{Q}$ against the data source T. Then it applies (6) to the results $\mathcal{R}$ to establish the similarity of the corresponding candidate query q' to the original query q_0 and determines the cohesiveness in the data source T according to (8). The results are then sorted based on their total score σ (1) to obtain the top-k ranked (2) semantically related results.

Assume that the initial query $q_0 = \{k_1, ..., k_n\}$ has the no-but-semantic-match problem for all $k_i \in q_0$, $|\mathcal{Q}|$ is the maximal number of candidate queries produced for q_0, d is the depth of the tree T, $|S|$ and $|S_1|$ are the maximal and minimal sizes of the inverted keyword lists for the semantic counterparts k_i', respectively and $|\mathcal{R}|$ is the maximal number of semantically related results for the candidate queries in $\mathcal{Q}$, then the complexity of the naive approach becomes $|\mathcal{Q}| \times nd|S_1| \log |S| + |\mathcal{R}| \log |\mathcal{R}|$. However, both $|\mathcal{Q}|$ and $|\mathcal{R}|$ could be potentially large, which makes the naïve approach impractical. We propose two efficient pruning techniques, called the *inter-query* pruning and *intra-query* pruning to significantly reduce the sizes of $\mathcal{Q}$ and $\mathcal{R}$, respectively.

3.4.1 Inter-query pruning

The main mechanism to stop processing unnecessary queries is our *inter-query* pruning technique. In fact, the *inter-query* pruning decides whether a query results can beat the results in the current top-*k* list. If not, the processing of the candidate queries stops and top-*k* list returns. Thus, the *inter-query* pruning technique guarantees to return the exact result. Assume the candidate queries $\mathcal{Q} = \{q_1, q_2, ..., q_l, q_{l+1}, ... q_{|\mathcal{Q}|}\}$ are sorted based on their similarities $\lambda(q_0, q_i)$ to the original query q_0. We obtain the following lemma.

Lemma 1 *Assume R^* is the k results of $\mathcal{Q}' = \{q_1, q_2, \ldots, q_l\}$ and σ^{min} is the min-score of the results R^*. Then, we can stop processing the rest of the candidate queries $\mathcal{Q}'' = \{q_{l+1}, ...q_{|\mathcal{Q}|}\}$ if $\lambda(q_0, q_{l+1}) < \sigma^{min}$.*

Proof Assume that r is a min-scored result in R^* for $q' \in \mathcal{Q}'$, i.e., $\sigma^{min} = \sigma(r, q', T)$ and r' is a result of the candidate query q_{l+1} whose similarity score is higher than any result of the queries $\mathcal{Q}'' = \{ql + 1, ..., q|Q|\}$. Now, assume that $\sigma(r', q_{l+1}, T) > \sigma(r, q', T)$. We prove that this can not happen if $\lambda(q_0, q_{l+1}) < \sigma^{min}$. To be scored higher than r, r'

must satisfy the following: $\lambda(q_0, q_{l+1}) > \frac{\sigma^{min}}{\theta(r',T)}$. However, the highest possible value of θ for any result in T is 1. By putting this into the above, we get $\lambda(q_0, q_{l+1}) > \sigma^{min}$, which contradicts the assumption. Therefore, R^* consists of the top-k semantically related results according to Definition 3 whose ranks are $\leq k$. □

Example 9 Consider the candidate queries $\mathcal{Q}$ given in Example 7. If we want to present the top-1 result to the user, and after processing the candidate queries up to q_4 we get $\sigma^{min} = 0.91 \times 0.4 = 0.36$, then we do not need to process q_5 as $\lambda(q_0, q_5) = 0.31 < \sigma^{min}$. Thus, from q_5 to the end of the list of $\mathcal{Q}$, no queries can score higher than σ^{min} and therefore, we can stop processing them.

3.4.2 Intra-query pruning

Although the *inter-query pruning* technique does not execute all of the candidate queries $q' \in \mathcal{Q}$ against T, it employs the pruning technique only in the first phase of the framework. That is, once we start processing a candidate query q', we compute all of its results. Consider the candidate queries $\mathcal{Q}$ of q_0 given in Example 7 and assume that the user requests only the top-1 result for q_0. Also, assume that the candidate queries in $\mathcal{Q}$ are sorted based on their similarities with q_0 and we have already processed the candidate queries from q_1 to q_5. The current top-1 result is r_1 (see in Figure 4) and σ^{min} is 0.36. Now, while processing the candidate query q_3, we can discard the result r_3 of q_3 (as shown in Figure 4) while generating it. That is, while reading through the keyword inverted lists S_{Jack}, $S_{fullprofessor}$ and S_{course} for q_3, we can partially compute r_3 consisting of the keywords $\{Jack, full\ professor\}$ for q_3 only (as highlighted in Figure 4), denoted by r_3^p, and compare the score 0.34 of r_3^p with the current σ^{min}, we can decide that the complete r_3 consisting of keywords $\{Jack, full\ professor, course\}$, denoted by r_3^c, can never outrank r_1 as $\theta(r_3^c, T) \leq \theta(r_3^p, T)$. The same

Algorithm 1 The Framework

```
   Input  : User Query $q_0$, Tuning Parameter $\alpha$, Keyword Inverted Lists $\mathcal{S}$
   Output: Top-k Semantically Related Results $\mathcal{R}^*$
1  $\mathcal{Q} \leftarrow generateCandidateQueries(q_0)$;
2  while $q' \leftarrow \mathcal{Q}\cdot getNext() \neq null$ do
3      $q'\cdot sim \leftarrow \lambda(q_0, q')$;                    // according to Eq.6
4  $\mathcal{Q} \leftarrow sortCandidates(\mathcal{Q})$;                  // based on $q'\cdot$sim
5  $\mathcal{R}^* \leftarrow null$ ;                                    // $\mathcal{R}^*$ is a min heap
6  $\sigma^{min} \leftarrow$MAXVAL ;                                   // max value
7  while $q' \leftarrow \mathcal{Q}\cdot getNext() \neq null$ do
8      if $\mathcal{R}^* \cdot getSize() = k$ and $q' \cdot sim < \sigma^{min}$ then
9          break ;                                              // inter-query pruning
10     if $\mathcal{R}^* \cdot getSize() = k$ then
11         $root \leftarrow \mathcal{R}^* \cdot root()$; $\sigma^{min} \leftarrow root \cdot score$;
12     $\mathcal{S}' \leftarrow \{\}$;
13     foreach $k_i \in q'$ do
14         $S_i \leftarrow retriveKeywordInvertedList(k_i, \mathcal{S})$;
15         $\mathcal{S}' \leftarrow \mathcal{S}' \cup S_i$;
16     $\mathcal{R}^* \leftarrow processQuery(\mathcal{R}^*, \sigma^{min}, \alpha, q', \mathcal{S}')$;
17 return $\mathcal{R}^*$
```

occurs while computing r_4 of q_4 and we can discard r_4 before generating the ultimate result. We call the above query pruning technique as the *intra-query pruning*.

3.4.3 The framework

Algorithm 1 presents the framework for processing the no-but-semantic-match query q_0 submitted by the user. First, it generates the candidate queries $\mathcal{Q}$ for q_0 as explained in Section 3.1.1, shown on line 1. The lines 2-4 compute the similarity between the candidate queries $\mathcal{Q}$ and the user query q_0 as explained in Section 3.1.2 and sort them. Then, a min-heap is initialized with $\mathcal{R}^*$ to $null$ and the min-score σ^{min} to $MAXVAL$ in lines 5-6. In lines 8-9, we stop processing the candidate queries in $\mathcal{Q}$ as soon as we find a query $q' \in \mathcal{Q}$ if $|\mathcal{R}^*| = k$ and $q'.sim < \sigma^{min}$. Otherwise, if $|\mathcal{R}^*| = k$, we update σ^{min} by reading the root entry of the heap $\mathcal{R}^*$ and adding its score $root.score$ to σ^{min} in lines 10-11. Then, for each keyword $k_i \in q'$ we retrieve their corresponding inverted lists and pass it to the $processQuery$ method (which is explained in detail in the following section) in line 16 to retrieve the results of q' and insert the eligible results into $\mathcal{R}^*$.

3.4.4 The processQuery method

In order to process the no-but-semantic-match query q_0, we need to execute each candidate keyword query $q' \in \mathcal{Q}$ against the data source T in the $processQuery$ method. There are two benchmark algorithms in the literature to compute the keyword query results on XML data T as given as follows: (a) scan eager [40] and (b) anchor based [35] algorithms. However, these two benchmark algorithms are not readily available to implement our $processQuery$ method. These algorithms only find the root of the subtree results in T, but ignore the distribution of the keyword match nodes in the subtree. To implement our $processQuery$ method with these benchmark algorithms, we need to address the following issues which are specific to our problem: (a) finding the tightest nodes under the confirmed SLCA root v_{slca} and (b) scoring the result partially based on its candidate query similarity and cohesiveness to apply *intra-query* pruning. We propose two techniques to implement the $processQuery$ method based on the benchmark algorithms as follows:

1. **S**can **E**ager based **Q**uery **P**rocessing (SE-QP) and
2. **AN**chor based **Q**uery **P**rocessing (AN-QP).

SE-QP algorithm Like scan-eager algorithm [40], SE-QP firstly sorts the inverted lists of the keywords in q'. Then, it picks a match node m_1 from the shortest inverted list S_1 and then, finds the closest match nodes to m_1 from other lists to compute the result root v_{slca}. However, the tightest subtree result computation and result scoring is delayed until we can confirm that this v_{slca} can be an actual SLCA node. Therefore, the cursor of each of the inverted list is retained until we decide that this v_{slca} cannot be an ancestor of any other result roots. Once we confirm that this v_{slca} is an actual SLCA node, we compute the tightest subtree result for it and score the result. While scoring the result, we also apply *intra-query* pruning here. Then, we advance the cursors of all of the lists and continue the above steps until we access all nodes in the list S_1.

Algorithm 2 SE-QP

```
   Input  : $\mathcal{R}^*, \sigma^{min}, \alpha, q', \mathcal{S}'$
   Output: Result Set:$\mathcal{R}^*$
1  $sortLists(\mathcal{S}')$; $r \leftarrow null$; $v_{slca} \leftarrow null$;
2  while $m_1 \leftarrow getNext(S_1) \neq null$ do
3      $m_i \leftarrow closest(m_1, S_i), \forall i \in [2, n]$;
4      $v^u_{slca} \leftarrow lca(m_1, ..., m_n)$;
5      if $r \neq null$ and $v_{slca} \not\prec_a v^u_{slca}$ then
6          for $i = 1 \rightarrow n$ do
7              $m^{li}_i \leftarrow getTight(S_i, r.cursor_i)$;
8              $d_i \leftarrow getDist(v_{slca}, m^{li}_i)$; $d \leftarrow d + d_i$;
9              $r \cdot score \leftarrow q' \cdot sim \times \frac{1}{log_\alpha(d+1)+1}$;
10             if $r \cdot score < \sigma^{min}$ then
11                 stop reading lists and jump to line 14.
12         $r \leftarrow (v_{slca}, \{m^{li}_i, \forall i \in [1, n]\})$;
13         update top-k list $\mathcal{R}^*$ with $r$;
14     $r \leftarrow null$;
15     if $v^u_{slca} \not\prec_a v_{slca}$ then
16         $v_{slca} \leftarrow v^u_{slca}$; $r.add(S_i.cursor, \forall i \in [1, n])$;
17 if $v^u_{slca} \not\prec_a v_{slca}$ then
18     $r \leftarrow (v^u_{slca}, \{m^{li}_i, \forall i \in [1, n]\})$;
19     score $r$ and update $\mathcal{R}^*$ with $r$ if $r \cdot score > \sigma^{min}$;
20 return $\mathcal{R}^*$
```

The SE-QP query processing technique is pseudocoded in Algorithm 2. In line 1, we sort the inverted list of all keywords in q' and do the initialization. In lines 2-3, it finds the match node m_1 from the shortest list S_1 and the match nodes m_i from other lists. In line 4, we find the potential root result v^u_{slca} which then should be confirmed as an actual SLCA node. In line 5, we check v^u_{slca} with the previous result root v_{slca}. If v_{slca} is not an ancestor for v^u_{slca}, denoted as $v_{slca} \not\prec_a v^u_{slca}$, v_{slca} is confirmed as the SLCA result root. Then, the corresponding tightest subtree result for v_{slca} is retrieved. To do so, we scan each list S_i by moving its cursor $r.cursor_i$ backward and forward to find the closest match nodes under v_{slca}, which is implemented in function $getTight$ of line 7. For each closest match node m^{li}_i, the distance of m^{li}_i with v_{slca} is computed by $getDist$ function in line 8 and the score of the result $r.score$ is computed partially in lines 8-9. We stop scanning other lists if the partial score cannot beat σ^{min} (intra-query pruning) and jump to line 14, which is given in lines 10-11. Otherwise, we keep scanning all lists to compute the tightest subtree result and the ultimate score of r for the v_{slca}. We update the min heap $\mathcal{R}^*$ by this result r in line 13. Now, we update v_{slca} with the current result root v^u_{slca} if $v^u_{slca} \not\prec_a v_{slca}$ in lines 15-16. In lines 17-19, if the last result root node is an actual SLCA, the similar steps are conducted to score its tightest subtree result and if promising, is used to update $\mathcal{R}^*$.

AN-QP algorithm Like scan-eager algorithm [40], SE-QP performs worse when the inverted lists have similar sizes (e.g., match node distribution). Also, it incurs many redundant computations when the data distribution is skewed in T. For example, if the match nodes are mostly distributed in one part of the XML tree in an inverted list, SE-QP reads all the nodes in S_1 and computes their LCAs to finalize the corresponding SLCAs. However, lots of these nodes can be skipped because they are far from the nodes in other inverted lists and cannot create SLCA nodes.

In order to skip the non-promising match nodes, like [35], AN-QP considers only the *anchor* match nodes for computing SLCA nodes. A set of match nodes $M = \{m_1, ...m_n\}$ for q' is said to be *anchored* by a match node $m_a \in M$ if for each $m_i \in M \setminus \{m_a\}$, $m_i = closest(m_a, S_i)$, where $closest(m_a, S_i)$ returns the match nodes in the list S_i which is closest to the node m_a[35]. Unlike SE-QP, the *anchor* match node m_a is picked from among inverted lists (not necessarily from the shortest one) so that it can maximize the skipping of redundant computations. Similar to SE-QP, AN-QP first extracts the SLCA result root v_{slca} and thereafter, finds the tightest subtree result under v_{slca} and partially score the result to apply *intra-query* pruning.

Algorithm 3 AN-QP

```
Input  : $\mathcal{R}^*, \sigma^{min}, \alpha, q', \mathcal{S}'$
Output: Result Set:$\mathcal{R}^*$
1  $r \leftarrow null$; $v_{slca} \leftarrow null$;
2  $m_a \leftarrow getAnchor(\{getNext(S_i), \forall i \in [1, n]\})$;
3  while $m_a \neq null$ do
4      $m_i \leftarrow closest(m_a, S_i), \forall i \in [1, n] \& i \neq a$;
5      $v^u_{slca} \leftarrow lca(m_1, ..., m_n)$;
6      if $r \neq null$ and $v_{slca} \not\prec_a v^u_{slca}$ then
7          for $i = 1 \rightarrow n$ do
8              $m^{li}_i \leftarrow getTight(S_i, r.cursor_i)$;
9              $d_i \leftarrow getDist(v_{slca}, m^{li}_i)$; $d \leftarrow d + d_i$
10             $r \cdot score \leftarrow q' \cdot sim \times \frac{1}{log_\alpha(d+1)+1}$;
11             if $r \cdot score < \sigma^{min}$ then
12                 stop reading lists and jump to line 15.
13         $r \leftarrow (v_{slca}, \{m^{li}_i, \forall i \in [1, n]\})$;
14         update top-k list $\mathcal{R}^*$ with $r$;
15     $r \leftarrow null$;
16     if $v^u_{slca} \not\prec_a v_{slca}$ then
17         $v_{slca} \leftarrow v^u_{slca}$; $r.add(S_i.cursor), \forall i \in [1, n]$;
18     $m_a \leftarrow getAnchor(\{getNext(S_i), \forall i \in [1, n]\})$;
19 if $v^u_{slca} \not\prec_a v_{slca}$ then
20     $r \leftarrow (v^u_{slca}, \{m^{li}_i, \forall i \in [1, n]\})$;
21     score $r$ and update $\mathcal{R}^*$ with $r$ if $r \cdot score > \sigma^{min}$;
22 return $\mathcal{R}^*$
```

The AN-QP technique is pseudocoded in Algorithm 3. Line 2 finds the anchor node m_a from the the inverted lists $\{S_1, ..., S_n\}$, which is implemented in function $getAnchor$. The potential result root v^u_{slca} is computed after finding the closest nodes $m_i \in S_i$ to m_a, $\forall i \in [1, n]$ and $i \neq a$ in lines 4-5. Here, if $v_{slca} \not\prec_a v^u_{slca}$, v_{slca} is confirmed as the SLCA result root as given in line 6. Therefore, we retrieve the closest match nodes m^{li}_i, $\forall i \in [1, n]$ under v_{slca}. Similar to SE-QP, the function $getTight$ of line 8, scans each list S_i by moving its cursor $r.cursor_i$ backward and forward to compute the tightest subtree result for v_{slca}. Then, we find the distance of each m^{li}_i with v_{slca} by the function $getDist$ and add it to the total distance of r in line 9. Then, we compute the score of the result $r.score$ partially in line 10. Here, we apply intra-query pruning if the partial score cannot beat σ^{min} and jump to line 15, which is given in lines 11-12. Otherwise, we compute the ultimate score of the result by scanning all lists to retrieve the tightest subtree result under v_{slca} and update r with its corresponding data in line 13. In line 14, the promising result r is inserted into $\mathcal{R}^*$. The current result root v^u_{slca} is saved into v_{slca}, if $v^u_{slca} \not\prec_a v_{slca}$ in lines 16-17. We update the anchor node m_a in line 18 for the next iteration. We check the last result root node and retrieve its tightest subtree result if it is an actual SLCA node in lines 19-20. Finally, we score it and update $\mathcal{R}^*$ with it if $r \cdot score > \sigma^{min}$ in line 21.

4 Batch processing

This section investigates a more efficient method for processing the candidate queries $\mathcal{Q}$ obtained from the initial query q_0 with the no-but-semantic-match problem. As the candidate queries in $\mathcal{Q}$ are generated by replacing the keywords of the initial query, they usually share a subset of keywords. It is possible to compute the results of these shared subsets of keywords among the queries in $\mathcal{Q}$ and then merge the results of the shared part with the exclusive part of each query $q' \in \mathcal{Q}$ [42]. However, two challenges arise here: (a) finding the groups of queries, called *batches*, which share a subset of keywords among them and can be executed efficiently; and (b) finding the tightest SLCA results for the shared part which can be ultimately merged with the exclusive part of each candidate query in a batch.

4.1 Constructing the candidate query batch

Given two candidate queries $q_1, q_2 \in \mathcal{Q}$, assume that $\mathcal{K}^s(q_1, q_2)$ denotes the set of keywords shared by them, i.e., $\mathcal{K}^s(q_1, q_2) \subseteq q_1, q_2$. We can achieve the best performance by putting q_1 and q_2 in a batch if they share the maximal set of keywords, i.e., $|\mathcal{K}^s(q_1, q_2)| = |q_1| - 1 = |q_2| - 1$ [42]. A candidate query batch is defined as given below:

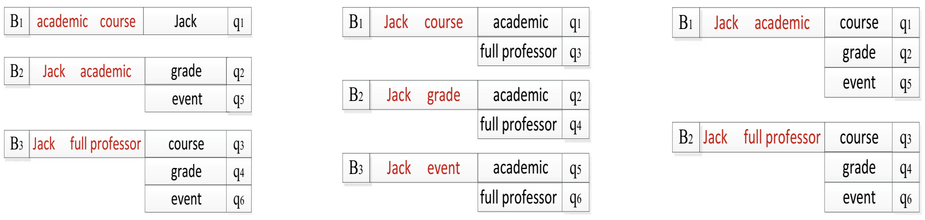
Definition 4 A candidate query batch, denoted by $\mathcal{B}$, is a subset of $\mathcal{Q}$ such that the following conditions hold: (a) $\forall q_1, q_2 \in \mathcal{B}, |\mathcal{K}^s(q_1, q_2)| = |q_1| - 1 = |q_2| - 1$; (b) $\forall q_1, q_2, q_3 \in \mathcal{B}$, $\mathcal{K}^s(q_1, q_2) = \mathcal{K}^s(q_2, q_3) = \mathcal{K}^s(q_1, q_3)$; and (c) $1 \leq |\mathcal{B}| \leq |\mathcal{Q}|$.

To execute the candidate queries in $\mathcal{Q}$, we have to construct the set of batches that can cover all queries in $\mathcal{Q}$. We call the set of candidate query batches that cover the queries in $\mathcal{Q}$ an *execution plan*, which is defined below:

Definition 5 An execution plan, denoted by $\mathcal{P}$, is a set of candidate query batches such that: (a) $\mathcal{Q} = \bigcup_{i=1}^{|\mathcal{P}|} \mathcal{B}_i \in \mathcal{P}$ and (b) $\forall \mathcal{B}_1, \mathcal{B}_2 \in \mathcal{P}, \mathcal{B}_1 \cap \mathcal{B}_2 = \emptyset$.

An execution plan $\mathcal{P}$ has its evaluation cost which is the summation of the execution costs of its constituent batches. The execution cost of a batch $\mathcal{B}$ directly depends on the inverted keyword lists which have to be accessed. Assume that $\mathcal{K}^u$ is the set of keywords of all candidate queries in a batch $\mathcal{B}$ that has not been covered by $\mathcal{K}^s$, i.e., $\bigcup_{\forall q_1 \in \mathcal{B}} q_1 \setminus \mathcal{K}^s$. We estimate the cost of executing $\mathcal{B}$, denoted by $c(\mathcal{B})$ as:

$$c(\mathcal{B}) = \sum_{k_1 \in \mathcal{K}^s} |S_{k_1}| + \sum_{k_2 \in \mathcal{K}^u} (|min(S_{\mathcal{K}^s})| + |S_{k_2}|) \quad (9)$$



B1 academic course Jack q1
B2 Jack academic grade q2
event q5
B3 Jack full professor course q3
grade q4
event q6
B1 Jack course academic q1
full professor q3
B2 Jack grade academic q2
full professor q4
B3 Jack event academic q5
full professor q6
B1 Jack academic course q1
grade q2
event q5
B2 Jack full professor course q3
grade q4
event q6

(a) Plan $\mathcal{P}_1$, $c(\mathcal{P}_1) = 38$ (b) Plan $\mathcal{P}_2$, $c(\mathcal{P}_2) = 39$ (c) Plan $\mathcal{P}_3$, $c(\mathcal{P}_3) = 34$

Figure 5 A part of the possible execution plans for processing the candidate queries in $\mathcal{Q}$

where $min(S_{\mathcal{K}^s})$ returns the shortest inverted keyword list size among the keywords in $\mathcal{K}^s$. However, there exist many plans for $\mathcal{Q}$ as shown in Figure 5. The optimal plan has the least cost. Discovering this optimal plan is a combinatorial optimization problem as there are many ways of constructing the candidate query batches from $\mathcal{Q}$. Here, we propose a *greedy approach* for discovering a sub-optimal plan which consists of the following steps: (a) retrieve the topmost similar query $q_1 \in \mathcal{Q}$ to q_0; (b) construct all plausible batches for q_1 as follows: (i) remove a keyword $k_1 \in q_1$ and construct $\mathcal{K}^s$ as $q_1 \setminus k_1$; (ii) retrieve all $q_2 \in \mathcal{Q}$ such that $\mathcal{K}^s \subset q_2$; (iii) insert q_1 and all q_2 into a plausible batch $\mathcal{B}_1$; (c) make the batch $\mathcal{B}_1$ as the actual batch that has the least unit cost $\frac{c(\mathcal{B}_1)}{|\mathcal{B}_1|}$; (d) remove all queries $\mathcal{B}_1$ from $\mathcal{Q}$; and (e) repeat the above steps until $\mathcal{Q}$ is empty.

Example 10 Consider the candidate queries $\mathcal{Q}$ presented in Example 7 and the sizes of the inverted lists as follows: $|S_{Jack}| = 3$, $|S_{academic}| = 1$, $|S_{fullprofessor}| = 1$, $|S_{course}| = 6$, $|S_{grade}| = 2$, $|S_{event}| = 2$.
We start with the topmost query $q_1 \in \mathcal{Q}$ and remove one keyword at a time from q_1 to create the plausible candidate query batches as follows:
$\mathcal{B}_{1.1}$ ($\mathcal{K}^s = \{Jack, academic\}$, $\mathcal{K}^u$=$\{course, grade, event\}$)
$\mathcal{B}_{1.2}$ ($\mathcal{K}^s = \{Jack, course\}$, $\mathcal{K}^u$=$\{academic, fullprofessor\}$)
$\mathcal{B}_{1.3}$ ($\mathcal{K}^s = \{academic, course\}$, $\mathcal{K}^u = \{Jack\}$)
We estimate the costs of the candidate query batches as follows: $\frac{c(\mathcal{B}_{1.1})}{|\mathcal{B}_{1.1}|} = 5.6$, $\frac{c(\mathcal{B}_{1.2})}{|\mathcal{B}_{1.2}|} = 8.5$, $\frac{c(\mathcal{B}_{1.3})}{|\mathcal{B}_{1.3}|} = 11$
Therefore, $\mathcal{B}_{1.1}$ is the initial candidate query batch. After excluding the queries in $\mathcal{B}_{1.1}$ from $\mathcal{Q}$, the next topmost similar candidate query q_3 is considered and the following plausible batches are constructed:
$\mathcal{B}_{2.1}$ ($\mathcal{K}^s = \{Jack, fullprofessor\}$, $\mathcal{K}^u$=$\{course, grade, event\}$)
$\mathcal{B}_{2.2}$($\mathcal{K}^s = \{Jack, course\}$, $\mathcal{K}^u = \{fullprofessor\}$)
$\mathcal{B}_{2.3}$($\mathcal{K}^s = \{fullprofessor, course\}$, $\mathcal{K}^u = \{Jack\}$)
The costs of the above plausible batches are as follows:
$\frac{c(\mathcal{B}_{2.1})}{|\mathcal{B}_{2.1}|} = 5.6$, $\frac{c(\mathcal{B}_{2.2})}{|\mathcal{B}_{2.2}|} = 13$, $\frac{c(\mathcal{B}_{2.3})}{|\mathcal{B}_{2.3}|} = 11$
Hence, $\mathcal{B}_{2.1}$ is the next candidate query batch. Now, if we exclude all queries in $\mathcal{B}_{2.1}$ from $\mathcal{Q}$, $\mathcal{Q}$ becomes empty and the process stops. Finally, plan $\mathcal{P}_3$ is our execution plan as shown in Figure 5c, which is sub-optimal.

4.2 Processing the candidate query batch

To process a candidate query batch $\mathcal{B}$, we need to compute the results of the shared part $\mathcal{K}^s$ first. Then, we need to merge these shared part results with the non-shared keywords $\mathcal{K}^u$ for

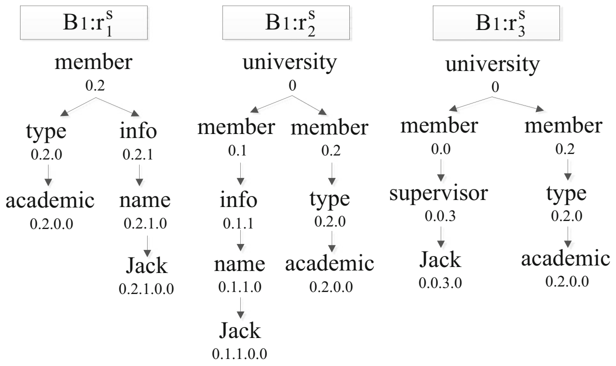


Figure 6 Shared part processing for query batch $\mathcal{B}_1$ of $\mathcal{P}_3$

$q'' \in \mathcal{B}$. However, computing the shared part results that can be merged with the unshared part is a non-trivial problem. This is because, the SLCA result roots of the shared part $\mathcal{K}^s$ do not guarantee to be the SLCA result roots for $q'' \in \mathcal{B}$. The result roots of q'' may ascend to higher levels in the tree T when the shared part results are merged with the unshared part $q'' \setminus \mathcal{K}^s$.

Consider the potential shared part results of $\{Jack, academic\}$ for the batch $\mathcal{B}_1$ of plan $\mathcal{P}_3$ as presented in Figure 6. Now, when we merge these results with the keyword *course* for $q_1 \in \mathcal{B}_1$, r^s_1 contributes to the final SLCA result root, which is $v^{r_1}_{slca} = [0.2]$ (see in Figure 4). However, when we merge these potential shared part results with the keyword *grade* for $q_2 \in \mathcal{B}_1$, r^s_3 contributes to the final SLCA result and the result root ascends to $v^{r_3}_{slca} = [0]$ (see in Figure 4). This indicates that we need to retain r^s_1 as well as r^s_3 as the actual shared part results, though the root of r^s_3 is not the SLCA result root of the shared part $\{Jack, academic\}$, but the root of r^s_1 is. Here, we do not need to retain r^s_2 as $d(r^s_2, T) = 7 > d(r^s_3, T) = 6$ and both r^s_2 and r^s_3 share the same root.

Lemma 2 *Assume $q'' \in \mathcal{B}$ and v^s_{slca} is the SLCA result root of the shared part of $\mathcal{B}$. Then, the tightest match nodes of the final result of q'' are under v^s_{slca} or under one of the ancestors of v^s_{slca}.*

Therefore, to process a candidate query batch shared part $\mathcal{K}^s$, we find the closest match nodes under the shared part result root v^s_{slca} as well as under all of its ancestors.

4.2.1 Inter and intra-batch pruning

Assume the candidate queries in a batch $\mathcal{B}_i$ are sorted based on their similarities to q_0 as follows: $\{q_1, q_2, ..., q_{l-1}, q_l, ..., q_{|\mathcal{B}_i|}\}$. Also, $\mathcal{R}^s_{\mathcal{B}_i}$ is the set of shared part results for $\mathcal{B}_i$. Then, the lower bound of the overall *distances* of the results $\mathcal{R}_{\mathcal{B}_i}$ of $\mathcal{B}_i$, denoted by $\underline{d}(\mathcal{R}_{\mathcal{B}_i}, T)$, is $min\{d(r^s \in \mathcal{R}^s_{\mathcal{B}_i}, T)\}$. Now, the upper bound of the *cohesiveness* of $\mathcal{R}_{\mathcal{B}_i}$, denoted by $\overline{\theta}(\mathcal{R}_{\mathcal{B}_i}, T)$ is computed using $\underline{d}(\mathcal{R}_{\mathcal{B}_i}, T)$ in Eq. 8. The upper bound of the σ of a result r_l of query $q_l \in \mathcal{B}_i$ is:

$$\overline{\sigma}(r_l, q_l, T) = \lambda(q_0, q_l) \times \overline{\theta}(\mathcal{R}_{\mathcal{B}_i}, T) \tag{10}$$

Assume $\mathcal{B}'_i = \{q_1, q_2, ..., q_{l-1}\}$ and $\mathcal{R}^*$ is the k results of $\mathcal{Q}' = \{\mathcal{B}_1, ..., \mathcal{B}_{i-1}, \mathcal{B}'_i\}$. Also, σ^{min} is the min-score for $\mathcal{R}^*$. Then, we can stop processing the candidate queries

$\{q_l, ..., q_{|\mathcal{B}_i|}\} \in \mathcal{B}_i$ if $\overline{\sigma}(r_l, q_l, T) < \sigma^{min}$. We call the above pruning technique as *intra-batch* pruning if $l > 1$, otherwise, we call it *inter-batch* pruning (prune the entire batch).

Algorithm 4 BA-QP

```
Input  : User Query $q_0$, Tuning Parameter $\alpha$, Keyword Inverted Lists $\mathcal{S}$
Output: Top-k Semantically Related Result $\mathcal{R}^*$
while $q' \leftarrow getNext(\mathcal{Q}) \neq null$ do
    if $\mathcal{R}^* \cdot getSize() = k$ and $q' \cdot sim < \sigma^{min}$ then
        break ;                                   // inter-query pruning
    for $i = 1 \rightarrow n$ do
        $\mathcal{K}^s \leftarrow q' \setminus k_i$
        $\mathcal{B} \leftarrow getBatch(\mathcal{Q}, \mathcal{K}^s)$
        $\mathcal{K}^u \leftarrow getUnique(\mathcal{B}, \mathcal{K}^s)$
        $\mathcal{B}.add(\mathcal{B}_i, c(\mathcal{K}^s, \mathcal{K}^u))$
    $\mathcal{B}^* \leftarrow getMinCost(\mathcal{B})$; $\mathcal{S}' \leftarrow \{\}$;
    retrieve Inverted Lists $\forall k_i \in \mathcal{B}^*.\mathcal{K}^s$ into $S'$;
    $\mathcal{R}^s \leftarrow sharedPartComputation(\sigma^{min}, \alpha, q'', \mathcal{S}')$
    while $q'' \leftarrow getNext(\mathcal{B}^*) \neq null$ do
        $\overline{r \cdot score} \leftarrow q'' \cdot sim \times \overline{coh(\mathcal{R}^s, T)}$;
        if $\mathcal{R}^* \cdot getSize() = k$ and $\overline{r \cdot score} < \sigma^{min}$ then
            break;                     // inter and intra-batch pruning
        if $\mathcal{R}^* \cdot getSize() = k$ then
            $root \leftarrow \mathcal{R}^* \cdot root()$; $\sigma^{min} \leftarrow root \cdot score$;
        $k^u \leftarrow q'' \setminus \mathcal{B}^*.\mathcal{K}^s$;
        $S_1 \leftarrow retrieveKeywordInvertedList(k^u, \mathcal{S})$;
        $\mathcal{R}^* \leftarrow mergeResults(\mathcal{R}^*, \mathcal{R}^s, \sigma^{min}, \alpha, q'', S_1)$;
    $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \mathcal{B}^*$
return $\mathcal{R}^*$
```

4.2.2 *The framework*

Algorithm 4 presents the framework for **ba**tch **q**uery **p**rocessing technique (BA-QP) of the no-but-semantic-match problem. Similar to first 6 lines of Algorithm 1, it generates the candidate queries, measures their similarity, sorts them and does the initializations in line 1 to apply *inter-query* pruning first. That is, at each step of iteration, we take the topmost similar query $q' \in \mathcal{Q}$ in line 2 and stop processing the candidate queries in $\mathcal{Q}$ as soon as we find a query $q' \in \mathcal{Q}$ such that $|\mathcal{R}^*| = k$ and $q'.sim < \sigma^{min}$ as given in lines 3-4.

Otherwise, we construct the sub-optimal candidate query batch $\mathcal{B}^*$ for q' as given in lines 5-10. In line 12, the shared part $\mathcal{K}^s$ of the candidate query batch $\mathcal{B}^*$ is processed and its results are stored in $\mathcal{R}^s$. Then, for each query $q'' \in \mathcal{B}^*$, the upper bound of its actual results' score $\overline{r.score}$ is computed in line 14. In lines 15-16, we apply *inter and intra-batch* pruning where we stop processing the candidate query batch $\mathcal{B}^*$ if $|\mathcal{R}^*| = k$ and $\overline{r.score} < \sigma^{min}$. Otherwise, if $|\mathcal{R}^*| = k$, we update σ^{min} by reading the root entry of the heap $\mathcal{R}^*$ in lines 17-18. In lines 19-20, the unshared keyword part k^u of the batch query q'' and the corresponding inverted list S_1 are retrieved. The procedure *mergeResults* in line 21 generates the final results for q'' by merging shared part results $\mathcal{R}^s$ with the unshared part k^u and inserts the promising results into $\mathcal{R}^*$. Finally, the queries $q'' \in \mathcal{B}^*$ are excluded from the $\mathcal{Q}$ as pseudocoded in line 22. The above steps continue until $\mathcal{Q}$ becomes empty.

Algorithm 5 BA-QP (sharedPartComputation)

```
Input  : $\sigma^{min}, \alpha, q', \mathcal{S}'$
Output: Shared Result $\mathcal{R}^s$
1  $\mathcal{R}^s \leftarrow$null;
2  $m_a \leftarrow getAnchor(\{getNext(S_i), \forall i \in [1, n-1]\})$;
3  while $m_a \neq null$ do
4      $m_i \leftarrow closest(m_a, S_i), \forall i \in [1, n-1]$ and $i \neq a$;
5      $v_{slca} \leftarrow lca(m_1, ..., m_{n-1})$;
6      $\mathcal{A} \leftarrow v_{slca} \cup getAncestors(v_{slca})$;
7      while $v^s_{slca} \leftarrow getNext(\mathcal{A}) \neq null$ do
8          if $v^s_{slca} \notin R^s$ then
9              for $i = 1 \to n-1$ do
10                 $m_i^{li} \leftarrow getTight(S_i, S_i.cursor)$;
11                 $d_i \leftarrow getDist(v^s_{slca}, m_i^{li})$; $r.d \leftarrow r.d + d_i$;
12                 $r \cdot score \leftarrow q' \cdot sim \times \frac{1}{log_\alpha(r.d+1)+1}$;
13                 if $r \cdot score < \sigma^{min}$ then
14                     stop reading lists, jump to line 6.
15             $r \leftarrow (v^s_{slca}, \{m_i^{li}, \forall i \in [1, n]\})$;
16             $\mathcal{R}^s \cdot insert(r)$;
17     $m_a \leftarrow getAnchor(\{getNext(S_i), \forall i \in [1, n-1]\})$;
18 return $\mathcal{R}^s$
```

The details of the $sharedPartComputation$ method in line 12 of Algorithm 4 is presented in Algorithm 5. In lines 2-5, we compute the result root v_{slca}. Then, we compute all the ancestors of v_{slca} which is implemented in the function $getAncestors$ and put them in the potential result roots $\mathcal{A}$ in line 6. For each potential result root $v^s_{slca} \in \mathcal{A}$, we check if it is already processed and is in memory $\mathcal{R}^s$ in line 8. If $v^s_{slca} \notin \mathcal{R}^s$, this result has not been processed and thus, we compute the closest nodes under v^s_{slca} in the inverted lists $S_i, \forall i \in [1, n-1]$, their distance to v_{slca}, and their score (lines 9-12). In lines 13-14, we check if the score can beat σ^{min} (intra-query pruning). In lines 15-16, r is updated and inserted into $\mathcal{R}^s$ because it is promising. In line 17, we update the anchor match node m_a for the next iteration.

The details of the $mergeResults$ method in line 21 of Algorithm 4 is presented in Algorithm 6. In lines 2-5, we compute the potential result root v^u_{slca} which is yet to be confirmed. If the previous result root v_{slca} is not an ancestor of v^u_{slca}, then v_{slca} is confirmed as the SLCA result root in line 6. Thus, from $\mathcal{R}^s$, we retrieve the precomputed closest match nodes $m_i^{li}, \forall i \in [1, n-1]$ of the shared part which is implemented in function $retrieveNode$ in line 8 and the distance of the shared part which is implemented in the function $retrieveDist$ in line 9. Then, in lines 10-11 we compute the closest match node m_n^{ln} for the unshared part and its distance to v_{slca} and finally, add this distance to the total distance of r. In lines 13-15, we compute $r.score$ and if $r.score > \sigma^{min}$, the result is added to $\mathcal{R}^*$. We update v_{slca} with the current result root v^u_{slca} if $v^u_{slca} \not\prec_a v_{slca}$ in lines 16-17. Then, the anchor match node m_a is updated in line 18. In lines 19-21, if the last result root is SLCA and is not considered, similar steps are taken and the corresponding result is inserted into $\mathcal{R}^*$ if it is promising.

Algorithm 6 BA-QP (mergeResults)

```
Input  : $\mathcal{R}^*, \mathcal{R}^s, \sigma^{min}, \alpha, q'', S_1$
Output: Top-k Result $\mathcal{R}^*$
1  $r \leftarrow null; v_{slca} \leftarrow null;$
2  $m_a \leftarrow getAnchor(\{getNext(S_1), getNext(S_2)\});$
3  while $m_a \neq null$ do
4      $m_i \leftarrow closest(m_a, S_i), \forall i \in [1, 2]$ and $i \neq a;$
5      $v^u_{slca} \leftarrow lca(m_1, m_2);$
6      if $r \neq null$ and $v_{slca} \not\prec_a v^u_{slca}$ then
7          if $v_{slca} \in \mathcal{R}^s$ then
8              $m^{li}_i \leftarrow retrieveNode(r, \mathcal{R}^s), \forall i \in [1, n-1];$
9              $d \leftarrow retrieveDist(r, \mathcal{R}^s);$
10             $m^{ln}_n \leftarrow getTight(S_1, r.cursor_1);$
11             $d \leftarrow d + getDist(v_{slca}, m^{ln}_n);$
12             $r \leftarrow (v_{slca}, \{m^{li}_i, \forall i \in [1, n]\});$
13             $r \cdot score \leftarrow q'' \cdot sim \times \frac{1}{log_\alpha(d+1)+1};$
14             if $r \cdot score > \sigma^{min}$ then
15                 update top-k list $\mathcal{R}^*$ with $r$;
16         else if $v^u_{slca} \not\prec_a v_{slca}$ then
17             $v_{slca} \leftarrow v^u_{slca}; r.add(S_1.cursor);$
18     $m_a \leftarrow getAnchor(\{getNext(S_i), \forall i \in [1, 2]\});$
19 if $v^u_{slca} \not\prec_a v_{slca}$ then
20     make $r$ by repeating lines 8-12;
21     score $r$ and insert to $\mathcal{R}^*$ if promising;
22 return $\mathcal{R}^s$
```

5 Experiments

This section evaluates the effectiveness and the efficiency of our approach for solving no-but-semantic-match problem in XML keyword search. To compare our results, we adapt and implement the closely related existing XOntoRank method [14] which uses ontology to enhance the XML keyword search on medical datasets. The adaptation is achieved as follows. We create a hash map for the ontologically relevant keywords to the original keywords by using ontological knowledge base. Then for each keyword $k_i \in q_0$ we look up the hash map and find the candidates and put them in the set of candidate keywords $\mathcal{K}$. Then we create Onto-DIL for each keyword $k_i \in q_0$ and all of its associated candidates in $\mathcal{K}$. Afterward, we compute the node score for each entry based on the relevance degree of the original keyword k_i to the candidate keywords in $\mathcal{K}$. Finally, the inverted list is added to Onto-DIL $\mathcal{S}_{onto}$. After creating $\mathcal{S}_{onto}$, we compute LCAs by using Onto-DIL index and for each result r, the score is computed using the node score of its matched nodes. If the result score $r.score$ is better than the threshold σ^{min}, the result is added to top-k list $\mathcal{R}^*$. The above is pseudocoded in Algorithm 7. We term this method as XO-QP in this paper.

Algorithm 7 XO-QP

```
Input  : User Query $q_0$, Data $T$, Ontology $O$
Output: Top-k Semantically Related Results $\mathcal{R}^*$
1  $H \leftarrow createHashMap(q_0, O)$;
2  $\sigma^{min} \leftarrow 0$;
3  foreach $k_i \in q_0$ do
4      $\mathcal{K} \leftarrow findRelevant(k_i, H)$;
5      $S' \leftarrow createDIL(k_i \cup \mathcal{K}, T)$;
6      $S'.add(NS(k_i, k)), \forall k \in \mathcal{K}$;
7      $\mathcal{S}_{onto} \leftarrow \mathcal{S}_{onto} \cup S'$;
8  while inverted lists in $\mathcal{S}_{onto} \neq null$ do
9      $r \leftarrow computeLCA(\mathcal{S}_{onto})$;
10     $r.score \leftarrow \sum m.NS, \forall m \in r$;
11     if $r.score > \sigma^{min}$ then
12         update top-k list $\mathcal{R}^*$ with $r$;
13         $update(\sigma^{min})$;
14 return $\mathcal{R}^*$
```

5.1 Settings

Datasets and queries We evaluate our algorithms on two real datasets: (a) IMDB 170MB, that includes around 150,000 recent movies and TV series. (b) DBLP 650MB, which contains publications in major journals and proceedings. We use a wide range of queries to test the efficiency of our proposed methods for each dataset. For each test query, we choose keywords which satisfy the followings: (a) a keyword should be used often by the users and (b) a non-existing keyword should have some semantic counterparts, which have direct mapping in the data source. The test queries have no-but-semantic-match problem. Table 4 presents a part of the test queries called sample queries for detailed analysis of efficiency and effectiveness of our methods.

Environment All algorithms are implemented in *C#* and the experiments are conducted on a PC with 3.2 GHz CPU, 8 GB memory running 64-bit windows 7.

5.2 Effectiveness

This section evaluates the effectiveness of our approach from different perspectives.

5.2.1 Our approach versus intuitive solution

When a user issues a query and faces an empty result, she might try to change the initial query to obtain some results. However, being unfamiliar with the data source, the user is likely to think of a few synonyms of the keywords of the initial query. Assume the user is a kind of expert and succeeds in constructing the 10 topmost similar queries when changing the initial query. These queries may not produce good results but we consider this *intuitive solution* as a benchmark to gauge the effectiveness of the technique we propose. Here, we compare the top-10 results retrieved by our approach which processes all possible candidate queries with the *intuitive solution* which processes only the 10 topmost similar queries for

Table 4 A Sample of test queries for IMDB and DBLP datasets

Dataset	#	Query	$\|\mathcal{Q}\|$
IMDB	q_0.1	ghost, badgering, movie	12
IMDB	q_0.2	battler, spanish, drama	272
IMDB	q_0.3	slump, federal, reserve, harshness, documentary	285
IMDB	q_0.4	reproach, trespasser, fight, drama	357
IMDB	q_0.5	treasonist, zombie, shiver	1295
IMDB	q_0.6	research, outlander, universe	3528
IMDB	q_0.7	mass murder, horror, perfidy	11900
IMDB	q_0.8	victory, exaltation, drama	851
IMDB	q_0.9	partiality, perfidy, fear, english	442
IMDB	q_0.10	criminal, overcharge, loneliness	12240
IMDB	q_0.11	criminal, fear, spanish	1152
IMDB	q_0.12	victory, exuberance, drama, fight	437
DBLP	q_0.1	exigency, analysis, system	252
DBLP	q_0.2	academic, fraudulence, threat	4500
DBLP	q_0.3	information, ordination, track	360
DBLP	q_0.4	involvement, neuroscience, indicant, information	350
DBLP	q_0.5	mutter, alarm, analysis	2079
DBLP	q_0.6	deceit, type, analysis	8190
DBLP	q_0.7	online, trust, selling	276
DBLP	q_0.8	aftermath, type, analysis, system	225
DBLP	q_0.9	psychopathy, symptom, visualization, science	3276
DBLP	q_0.10	interloper, search , analysis, system	352
DBLP	q_0.11	trace, audit , system	238
DBLP	q_0.12	trace, type, analysis	255

all test queries given in Table 4. In Figures 7a and 8a, the average candidate query similarity of the top-10 results for the two approaches are presented for each dataset. Clearly for all the test queries, the average similarity of the results retrieved by our approach is very close to the average candidate query similarity of the results of the *intuitive solution*. Figures 7b and 8b show, on average, smaller number of edges (indicate better cohesiveness) for the top-10 results retrieved from our approach compared to the *intuitive solution*. This shows that an ad-hoc approach, even if it is suggested by an expert, is unlikely to find results that are superior to those provided by the systematic method suggested in this study. In comparison with the *intuitive solution*, our approach also does not retrieve results from the candidate queries that are far from the user given initial query in terms of candidate query similarity. In addition of it, our approach provides the flexibility of trading of the above two aspects, e.g., sometimes users may prioritize cohesive results over similarity to the original query. The tuning parameter α in our approach can be used to balance the weight of these two aspects. Figures 7c and 8c, show that as α becomes smaller, the similarity of the results decreases because more priority is given to result cohesiveness and therefore, the cohesiveness score of the results improves. That is, by setting α to smaller value, our approach effectively retrieves more cohesive results while the similarities of their contributing candidate queries are not very far from the initial query.

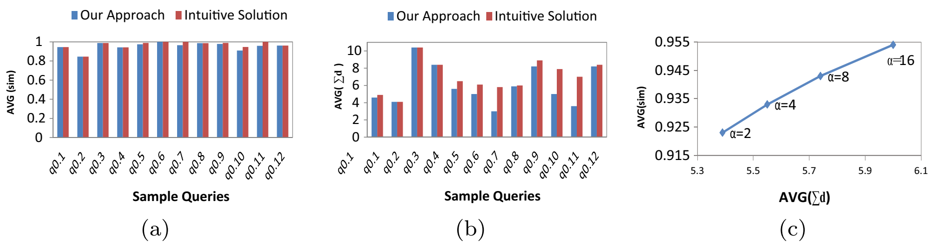


Figure 7 **a** Average similarity of the candidate queries and **b** average distance of top-10 results for our approach and intuitive solution on IMDB; **c** Average query similarity versus average result distance with varying α

5.2.2 Evaluation of quality

In this section, we evaluate the quality of our approach and compare it with XO-QP. Here, we select some sample queries with no-but-semantic-match problem for both datasets to conduct a comprehensive user study and thereafter, evaluate the overall quality of our approach. In order to carry out a fair user study, we select the users among both experts who have worked in XML keyword search areas and naive users who are graduate computer science students. To do the study, we present to the users with the original queries and their top-10 candidate queries/results retrieved from the top result list $\mathcal{R}^*$. After that, we ask the users to assess the quality of each candidate query with regards to their semantic similarity to the original query by scoring the candidate queries/results using Cumulated Gain metric [22]. They score each candidate query/result from 0 to 5 points (5 means the best and 0 means the worst).

Ranking scheme comparison The average quality scores of the top-5 and top-10 queries /results for our approach and the existing counterpart XO-QP are presented in Figures 9 and 10 respectively. From Figure 9, we observe that our proposed method suggests reasonable results for the no-match query for both top-5 and top-10 results. Also, we see that the average quality of top-5 results are always better than the average quality of top-10 results which indicates that our ranking function successfully ranks more similar and meaningful results higher than the rest of the results. However, we observe that in many cases for the existing method XO-QP, the average quality of results for top-10 is higher than top-5 results (for instance for $q_{0.2}, q_{0.4}, q_{0.11}$ in IMDB and for $q_{0.3}, q_{0.7}, q_{0.10}$ in DBLP) as shown in Figure 10. This indicates that XO-QP sometimes ranks some less similar and meaningful

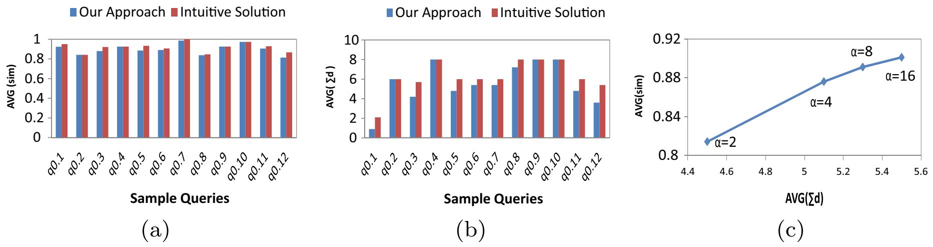


Figure 8 **a** Average similarity of the candidate queries and **b** average distance of top-10 results for our approach and intuitive solution on DBLP; **c** Average query similarity versus average result distance with varying α

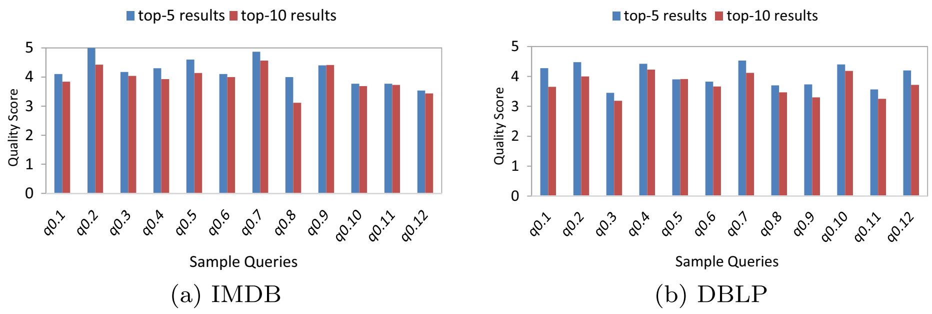


Figure 9 Average quality of results in our approach: top-5 vs. top-10

results higher. This is probably because of adapting XRank scheme into XO-QP. We conclude that our approach addresses the no-but-semantic-match problem better than the existing XO-QP method.

Precision In this section, we compare the precision of our proposed method with the XO-QP method. In order to make a comparison, we count the number of meaningful results in top-10 results. A result is regarded as meaningful if the average quality score of the assessors is not less than 3. We see that the precision of our approach and XO-QP is presented on both IMDB and DBLP as shown in Figure 11. In IMDB, we see that the precision of our proposed method is better than XO-QP. This is because we use both semantic similarity and cohesiveness scores to rank the results and retrieve the tightest SLCA results with the maximum similarity to the original query keywords. Also in DBLP, we observe that the precision of our method is better than XO-QP in many cases specially in $q_{0.1}$, $q_{0.3}$ and $q_{0.8}$. That's because our method can effectively retrieve the most similar results that are cohesive and make a meaningful combination in terms of data cohesiveness. However, in some cases like $q_{0.11}$ and $q_{0.12}$, we see that the precision of our method is smaller than XO-QP because it uses less similar keywords in the candidate query to retrieve more cohesive results. But the precision is not largely deteriorated as we see in XO-QP in many cases.

5.3 Efficiency

To demonstrate the efficiency of our approach, we use a baseline method which applies inter query pruning only, and uses scan eager for query processing. We call this baseline query

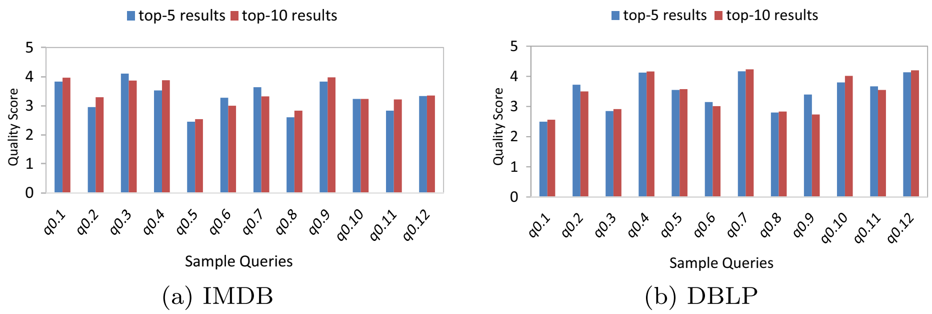


Figure 10 Average quality of results in XO-QP: top-5 vs. top-10

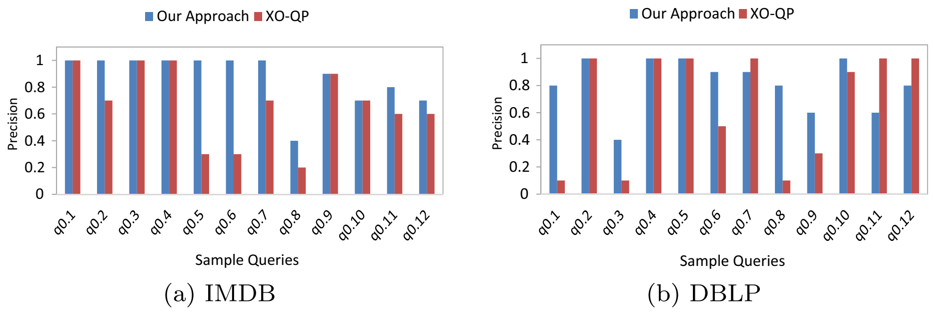


Figure 11 Precision comparison for computing top-10 results

processing as BL-QP. The purpose of this baseline is to demonstrate the efficiency of the intra-query pruning scheme in our proposed methods. Overall, we compare the performance of the following methods: (a)BL-QP, (b) SE-QP, (c) AN-QP, and (d) BA-QP, and (e) XO-QP.

5.3.1 *Processing time*

Figure 12 shows the response time of computing the top-10 semantically related results for the sample queries (presented in Table 4) on the IMDB and DBLP datasets. According to the results, BA-QP achieves the best performance. The reason behind this is that we apply inter and intra batch prunings in BA-QP in addition of inter and intra-query prunings. Also, we share the partial results of the shared keywords among the candidate queries in a batch. On average, BA-QP consumed 30 percent time of the time needed by the SE-QP and AN-QP methods. The difference, however, depends on the number of candidate queries. In Figure 12a, the response time for processing the test query $q_0.1$ in BA-QP is very close to those of SE-QP and AN-QP due to the small number of candidate queries for $q_0.1$ which is 12 and insufficient number of batches (in this case we have only one batch) to apply inter-batch pruning. Moreover, if the cost of merging the shared part $\mathcal{K}^s$ with the unshared part $\mathcal{K}^u$ is relatively high in most of the batches, BA-QP may not outperform the other methods as we see in $q_0.5$ on DBLP. In IMDB $q_0.5$, however, AN-QP outperforms SE-QP by a large margin because the data distributions in most of the inverted lists are skewed. In such case, many nodes are skipped in AN-QP, which improves the performance. Clearly, BL-QP has the worst performance on most cases because it does not apply the intra query pruning,

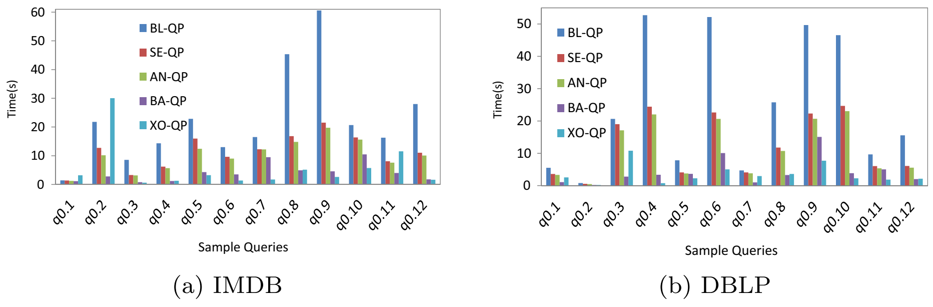


Figure 12 Processing time of the sample queries for computing top-10 results

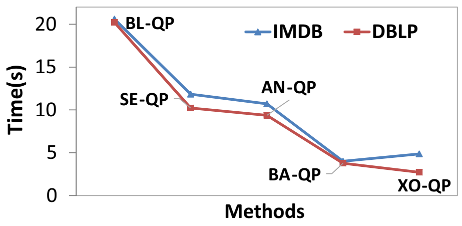


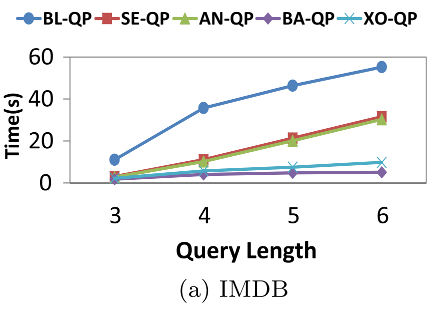
Figure 13 Average processing time on all test queries

therefore, incurs unnecessary computations. The XO-QP, however, is comparable to BA-QP in many cases because we set the node score threshold to 0.9 which means that only the relevant nodes to the keywords with the similarity degree bigger than 0.9 are selected. If we set this value to a smaller number, XO-QP performance deteriorates and gets closer to AN-QP and SE-QP. Moreover, XO-QP builds a special inverted list called XOnto-DIL for the keywords which takes additional time and space for its creation while our proposed methods do not use such indexing and use the normal DIL for query processing.

In Figure 13, the average processing time of different methods for all test queries are presented. Clearly the baseline method which does not use intra query pruning spends the maximum time for processing on both datasets. The SE-QP and AN-QP spend less time comparing to BL-QP because we apply inter and intra query pruning and therefore, the processing terminates early when we reach to the global σ^{min}. Moreover, there is a narrow improvement in the AN-QP over SE-QP due to using the anchor node processing which skips many redundant computations and expedites the efficiency. The BA-QP processing improves sharply on both datasets because in BA-QP, we execute queries in batch and share the computations among the queries in the batch and thereafter, we reach to the global σ^{min} before SE-QP and AN-QP can do. Also, we can apply inter and intra-batch pruning which expedite its efficiency. In XO-QP, however, the performance is close to BA-QP on both datasets because we build the XOnto-DIL on the nodes with the relevance degree no less than 0.9. In such condition, only the highly relevant nodes are selected to replace the keywords, therefore, the processing time does not grow considerably due to small number of candidate keywords. In contrast, by setting the threshold to lower numbers, the processing time will grow exponentially and gets closer to SE-QP. Furthermore, our methods do not need to build special indexing, therefore, avoid using additional space and offline processing time for building such indexing.

5.3.2 *Effect of query length*

In this experiment, we choose test queries that have at least 100 candidate queries. At each step, we set their length to a number of settings ($\{3, 4, 5, 6\}$) and compute the average response time of all queries when processing 100 candidate queries while setting k to 10. In each step, for the test queries that have smaller number of keywords, we add additional keywords to increase their length. We also use the same queries to conduct experiments and compare the results with XO-QP method. In Figure 14 the effect of growing the length of queries on the response time is analyzed. The response time of BA-QP is almost fixed compared to XO-QP, BL-QP, SE-QP, and AN-QP on IMDB. Similarly in DBLP, BA-QP has the slowest growth in response time when the number of keywords increases while BL-QP, SE-QP and AN-QP show a big jump in processing times. Also, XO-QP shows a jump in the processing time when the query length increases. Thus, the performance of XO-QP is sensitive to the query length parameter. The sharpest increase in the processing time is related



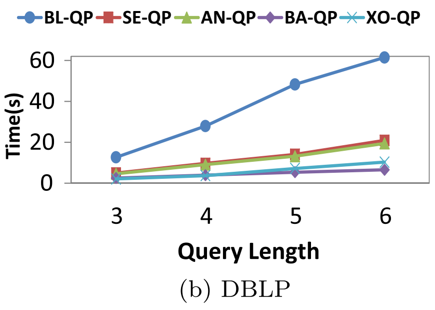


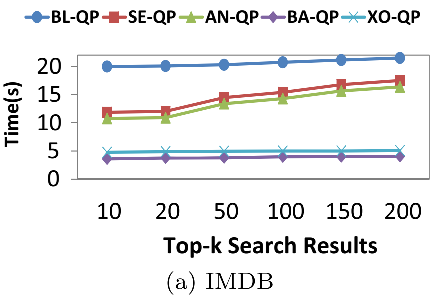
Figure 14 Average processing time of the test queries with varying query length (i.e., number of keywords)

to BL-QP because it does not apply intra-query pruning, therefore, all the inverted lists are accessed during the processing and this incurs many useless computations specifically when the query length increases.

In summary, we can conclude that BA-QP method is not sensitive to the number of keywords due to sharing the computations while the performance of SE-QP and AN-QP is highly sensitive to the query length.

5.3.3 *Effect of top-k list size k*

In this experiment, we vary the top-k size k and compute the average processing time. In Figure 15, we observe the effect of top-k size k on the average processing time for different methods. Clearly, the BA-QP and XO-QP methods are not that sensitive to the value of k. By increasing the value of k, processing times of BA-QP and XO-QP show only a small increase or almost fixed. On the other hand, for SE-QP and AN-QP methods, any increase on k, leads to a considerable jump on the average processing time. However, this increase is not big when we change k from 10 to 20. When k is selected as a bigger number, σ^{min} will be smaller and this causes the inter query pruning in SE-QP and AN-QP to be less effective. That is, the application of inter-query pruning is delayed in SE-QP and AN-QP for big k. However in BA-QP, we execute a candidate query batch by sharing the computations among the queries in it and thereafter, we reach to the global σ^{min} before SE-QP and AN-QP can do. Also, we can apply inter and intra-batch pruning with this early found global σ^{min} which helps BA-QP to expedite its efficiency.



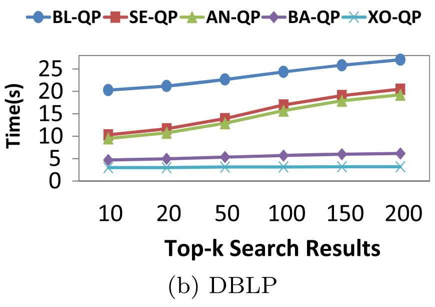
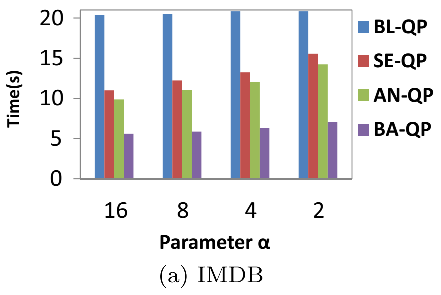


Figure 15 Average processing time of the test queries with varying k in top-k list



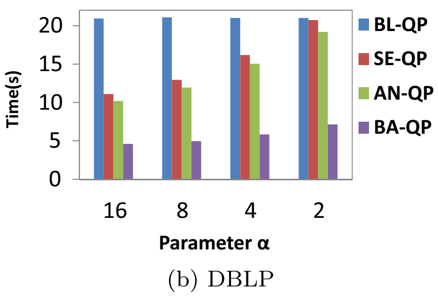


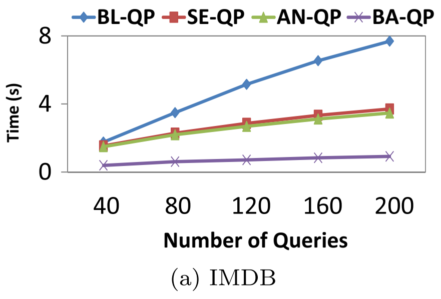
Figure 16 Average processing time of the test queries with varying tuning parameter α

5.3.4 *Effect of tuning parameter* α

In this experiment, we vary α between a number of settings ($\{2, 4, 8, 16\}$) and compute the average processing time of the test queries when $k = 10$. Figure 16 presents the effect of choosing different values for α. Larger values of α reduce the sensitivity to data cohesiveness. This usually leads to a more effective intra-query pruning and decreases the processing time. On the contrary, as α decreases, the possibility for the partial results scores to be smaller than the σ^{min} also decreases. In this case, the intra-query pruning becomes less effective and therefore, the processing time increases, specifically, in SE-QP and AN-QP. In BL-QP, however, there is no significant difference in the processing time when α changes to smaller number. This is because BL-QP does not apply intra query pruning and therefore, the processing time is not affected that much by α.

5.3.5 *Effect of candidate queries number* $|\mathcal{Q}|$

For this experiment, we choose test queries which have at least 200 candidate queries. At each step we process a certain number ($\{40, 80, 120, 160, 200\}$) of their candidate queries and compute the average processing time when $k = 10$. In Figure 17, the effect of growing the number of candidate queries $|\mathcal{Q}|$ on the response time is analyzed. The response time of BA-QP method grows slowly compared to BL-QP, SE-QP, and AN-QP which show a sharp rise in each step. In BL-QP, the growth in the query processing time is the maximum among the methods because it does not apply intra query pruning, therefore, it incurs many unnecessary computations during execution of the candidate queries. We can conclude that BA-QP is not that sensitive to $|\mathcal{Q}|$. This is because BA-QP not only shares the computations



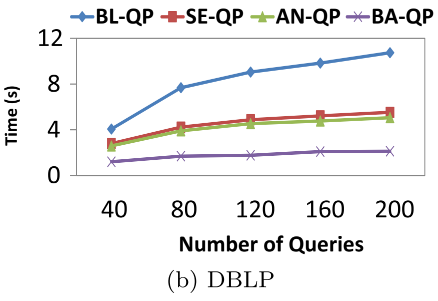


Figure 17 Processing time with varying number of candidate queries $|\mathcal{Q}|$

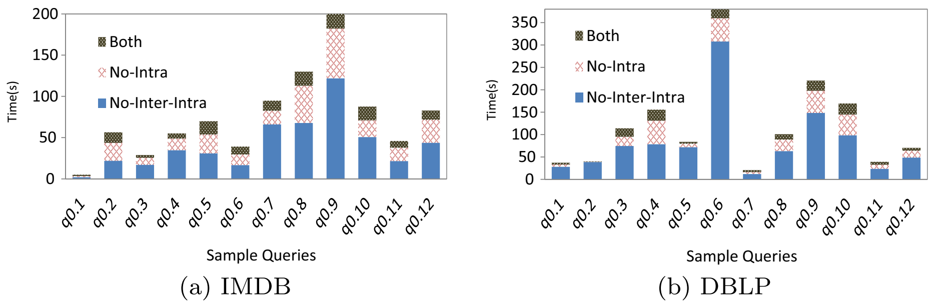


Figure 18 Inter and Intra Query Pruning Improvement on Sample Queries

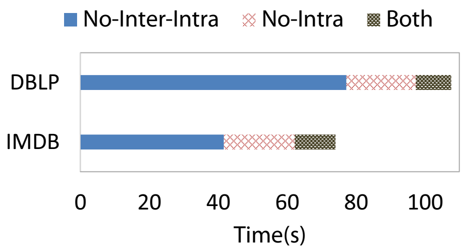
among the candidate query batch but also applies inter and intra batch pruning with the early-found global σ^{min} to expedite its performance.

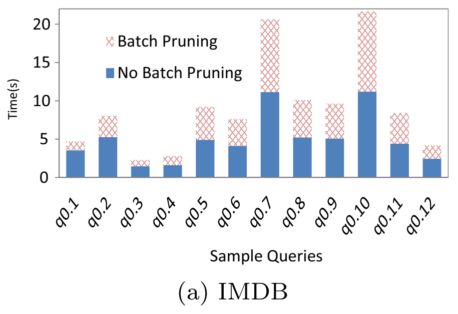
5.3.6 Pruning improvement

In this section, we show the pruning effect on the processing time. From Figure 18, the sample queries processing time are shown for 3 scenarios: (a) when there is no pruning for processing the queries, (b) when only the inter query pruning is implemented, and (c) when both inter and intra pruning methods are implemented. The processing time are measured using SE-QP method. Clearly, the inter query pruning shows the most effective method to cut the processing time on most of the sample queries. If the number of candidate queries is large and the breaking point occurs when most of the queries are not executed, then inter query has the best performance. e.g., in the sample queries $q_{0.9}$ and $q_{0.10}$ on IMDB, the number of executed queries are $\frac{225}{442}$ and $\frac{4918}{12240}$ respectively and in the sample queries $q_{0.6}$ and $q_{0.9}$ on DBLP, the number of executed queries are $\frac{912}{8190}$ and $\frac{1287}{3276}$ respectively. Therefore in these cases, most of the queries are not executed by using inter query pruning and the processing time reduced sharply. The intra query pruning is more effective when the number of query keywords is bigger or the inverted list that is not accessed due to pruning is big sized. In such condition, some inverted lists are not accessed when the result is not able to beat the σ^{min} and this expedites the processing time. For example, in the sample queries $q_{0.8}$ and $q_{0.9}$ on IMDB and $q_{0.4}$ and $q_{0.9}$ on DBLP, the query keywords are from 4 to 5 keywords and include some big sized inverted lists that are not accessed, therefore the processing time reduced considerably.

Figure 19 presents the average processing time for the set of test queries for 3 scenarios: (a) no pruning is implemented, (b) only inter query pruning is implemented, and (c) both pruning techniques are implemented. Clearly, the processing time for the case with no

Figure 19 Average pruning improvement





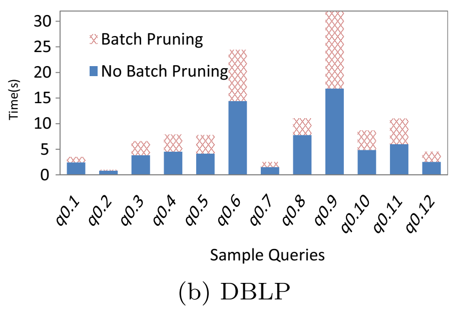


Figure 20 Batch Pruning Improvement on Sample Queries

pruning is maximum on both datasets. This shows that inter query pruning has the most tangible effect on the processing time by avoiding to execute the queries that cannot contribute to the $\mathcal{R}^*$. The efficiency improvement on IMDB and DBLP is 2 and 3 times respectively. After that, intra query pruning expedites the efficiency by avoiding to access all inverted lists when the result cannot beat σ^{min}.

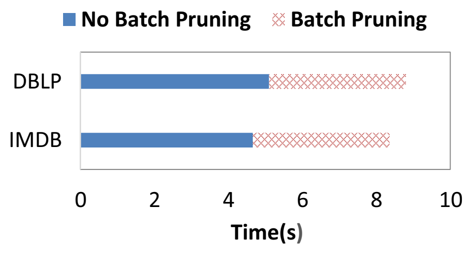
Figure 20 presents the sample queries processing time for BA-QP in 2 scenarios: (a) when no batch pruning is implemented, (b) when batch pruning is implemented. In most cases, the processing time for the method which uses pruning has decreased. The batch pruning becomes more effective when the query length increases as shown in $q_0.3$ and $q_0.12$ on IMDB or in $q_0.4$ and $q_0.8$ on DBLP. Moreover, when the candidate queries contain some big inverted lists and we reach to the global σ^{min} early, the improvement is more considerable as in $q_0.1$ and $q_0.2$ on IMDB and in $q_0.1$ and $q_0.7$ on DBLP.

Figure 21 shows the average processing time of the test queries for BA-QP on 2 cases: (a) when no batch pruning is implemented, (b) when batch pruning is implemented. We observe from the picture that the batch pruning improved the processing time on both datasets, however, the improvement on DBLP is more considerable. The improvement is achieved because we reach to the global σ^{min} earlier by applying batch pruning, therefore it expedites the performance.

6 Related work

Failed queries When a user queries a data source, the result may be empty or otherwise below expectation. This problem known as failed queries has inspired a broad range of research in the database community (e.g. [10, 19–21, 44]). In the context of relational databases, the problem has been studied by Nambiar and Kambhampati [32], Muslea [30] as

Figure 21 Average Batch Pruning Improvement



well as Muslea and Lee [31]. Nambiar and Kambhampati [32] presented approximate functional dependencies to relax the user original query and find tuples similar to the user query. Muslea [30], as well as Muslea and Lee [31] used machine learning techniques to infer rules for generating replacement queries. Amer-Yahia, Cho and Srivastava [3], Brodianskiy and Cohen [7] as well as Cohen and Brodianskiy [11] studied query relaxation for XML data. The studies proposed to discover the constraints in the queries that prevent results from being generated and remove them so that a result can be produced. All these investigations focus on the modification of the user original query constraints on the content level rather than the semantic analysis of the original query constraints. Bao et al. [4] studied the search intention and relevance ranking problems in the XML keyword search. They proposed to exploit the statistics of underlying XML data to effectively rank the results of all possible search intentions. Truong et al. [36] studied the XML keyword search for irregular documents. They proposed a novel algorithm called MESSIAH to address the missing element problem by improving SLCA semantics to support queries involving missing elements. Hill et al. [17], used the ontology information to relax structured XML queries. Farfan et al. [14], proposed XOntoRank system to address the ontology-aware XML keyword search of electronic medical records. Unlike their work which uses SNOMED ontology for enhancing the search on medical records, we address the general no-match problem on XML data and use a general ontological knowledge base like a thesaurus or dictionary to solve the problem for general documents.

Query expansion Query expansion has widely studied in many works such as [23, 24, 34, 39]. Schenkel, Theobald, and Weikum [34] proposed XXL which combines the keyword search with structural conditions and semantic similarity to increase the quality of results. Kim and Kong [23] suggested a query expansion technique that uses an ontology algorithm to map a target DTD to ontology. This scheme is successful for expanding the queries minimally. Kim, Kong, and Jeon [24] developed a web XML document search engine that applies ontology-DTD match algorithm for remote documents. However, in all of the above works, the focus is on structured queries. In our work, we find some semantic counterparts for specific non-mapped keywords for replacement, therefore, query expansion is not useful in our case.

Recommendation systems Users are often interested in items similar to those they have visited before or to content that has been looked up by similar users. These items are presented by the recommendation systems. Akbarnejad et al. [1] and Chatzopoulou, Eirinaki and [9] proposed query recommendation based on a prediction of the items that user is interested in. Yao et al. [41] proposed to exploit structural semantics for query reformulation. Meng, Cao and Shao [28] used the semantic relationships between keywords and keyword queries to suggest a set of keyword queries from the query log. However, the semantic relationship is interpreted as the co-occurrence of the keywords and no ontological analysis is carried out. Moreover, the work focuses on extracting similar queries from the query log using data mining techniques without processing the results. Drosou and Pitoura [13] presented a database exploration framework which recommends additional items called "You May Also Like" results. However, the recommended results are compiled based on the results of the original query and there is no focus on semantic connection between the original query and recommended results.

Mismatch problem Sometimes the system shows erroneous mismatch results for a user query which is called mismatch problem. Bao et al. [5] proposed a framework to detect the keyword queries that lead to a list of irrelevant results on XML data. They detect a mismatch problem by analyzing the results of a user query and inferring the user's intended node type result based on data structure. Based on this, they are able to suggest queries with relevant results to the user. Unlike the current study, Bao et al. investigate ways of producing relevant results instead of finding results for no-match queries.

Query cleaning Sometimes the empty result is caused by typographical errors. Pu and Yu [33] and Lu et al. [26] investigated a way of suggesting queries that have been cleaned of typing errors. Unlike our study, these authors do not tackle the problem of non-mapped keywords.

Ontology-based querying Many studies have used ontology information for searching the semantic web [12, 25, 27]. Studies by Aleman-Meza [2], Cakmak and Özsoyoglu [8] as well as Wu, Yang and Yan [37] used ontology information to find frequent patterns in graphs. Wu, Yang and Yan [37] proposed an improved subgraph querying technique by ontology information. They revised subgraph isomorphism by mapping a query to semantically related subgraphs in terms of a given ontology graph. Our work generates substitute queries for the user given keyword query by extracting the semantically related keywords from the ontological knowledge base and thereafter, produce semantically related results to the user query instead of returning an empty result set to the user.

7 Conclusion and future work

This paper investigates ways of efficiently building substitute queries against XML data sources when the user given keyword query fails to produce any result as one or more of its keywords do not exist in the data source. Our approach depends on an ontological knowledge base for a discovery of semantically related keywords to generate the substitute queries, which can be executed against the data source to produce the semantically related results for the user's original query. As the number of substitute queries can be potentially large and also, not all semantically related results are meaningful to the same degree, we propose efficient pruning techniques to reduce the number of substitute queries and return only the top-k semantically related results. We develop two query processing algorithms to evaluate the substitute queries against the data source based on our pruning techniques. We also develop a batch processing technique that exploits the shared keywords among the substitute queries to expedite the performance further. The extensive experiments with two real datasets validate the effectiveness and efficiency of our approach. There are some directions to continue this research in the future. One such direction is to solve the no-but-semantic-match problem using other popular XML keyword search semantics such as ELCA. Another interesting direction is to consider special data environment and to use Hadoop to improve the processing time in addition to the batch query processing technique.

Acknowledgments This work is supported by the Australian Research Council discovery grants DP140103499 and DP160102412.

References

1. Akbarnejad, J., Chatzopoulou, G., Eirinaki, M., Koshy, S., Mittal, S., On, D., Polyzotis, N., Varman, J.S.V.: SQL Querie recommendations. PVLDB **3**(2), 1597–1600 (2010)
2. Aleman-Meza, B., Halaschek-Wiener, C., Sahoo, S.S., Sheth, A.P., Arpinar, I.B.: Template based semantic similarity for security applications. In: ISI, pp. 621–622 (2005)
3. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree pattern relaxation. In: EDBT, pp. 496–513. Springer, London (2002)
4. Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective XML keyword search with relevance oriented ranking. In: ICDE, pp. 517–528 (2009)
5. Bao, Z., Zeng, Y., Ling, T.W., Zhang, D., Li, G., Jagadish, H.V.: A general framework to resolve the mismatch problem in XML keyword search. VLDB J. **24**(4), 493–518 (2015)
6. Bouquet, P., Kuper, G.M., Zanobini, S.: Asking and answering queries semantically. In: WOA, pp. 22–27 (2005)
7. Brodianskiy, T., Cohen, S.: Self-correcting queries for xml. In: CIKM, pp. 11–20. ACM, New York (2007)
8. Cakmak, A., Özsoyoglu, G.: Taxonomy-superimposed graph mining. In: EDBT, pp. 217–228 (2008)
9. Chatzopoulou, G., Eirinaki, M., Polyzotis, N.: Query recommendations for interactive database exploration. In: SSDBM, pp. 3–18 (2009)
10. Chu, W.W., Yang, H., Chiang, K., Minock, M., Chow, G., Larson, C.: CoBase: A scalable and extensible cooperative information system. Intell. Inf. Syst. **6**(2), 223–259 (1996)
11. Cohen, S., Brodianskiy, T.: Correcting queries for xml. Inf. Syst. **34**(8), 757–777 (2009)
12. Corby, O., Dieng-Kuntz, R., Faron-Zucker, C., Gandon, F.L.: Searching the semantic web approximate query processing based on ontologies. IEEE Intell. Syst. **21**(1), 20–27 (2006)
13. Drosou, M., Pitoura, E.: Ymaldb: exploring relational databases via result-driven recommendations. VLDB J. **22**(6), 849–874 (2013)
14. Farfan, F., Hristidis, V., Ranganathan, A., Weiner, M.: XOntoRank: Ontology-aware search of electronic medical records. In: ICDE, pp. 820–831 (2009)
15. Feng, J., Li, G., Wang, J., Zhou, L.: Finding and ranking compact connected trees for effective keyword proximity search in XML documents. Inf. Syst. **35**(2), 186–203 (2010)
16. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: Ranked Keyword Search over XML Documents. In: SIGMOD, pp. 16–27 (2003)
17. Hill, J., Thorson, J., Guo, B., Chen, Z.: Toward ontology-guided knowledge-driven XML query relaxation. In: Second International Conference on Computational Intelligence, Modeling and Simulation, pp. 448–453 (2010)
18. Hristidis, V., Koudas, N., Papakonstantinou, Y., Srivastava, D.: Keyword proximity search in XML trees. IEEE Trans. Knowl Data Eng. **18**(4), 525–539 (2006)
19. Huh, S., Moon, K.H., Ahn, J.K.: Cooperative query processing via knowledge abstraction and query relaxation. In: Advanced Topics in Database Research, pp. 1:211–228 (2002)
20. Islam, S., Liu, C., Zhou, R.: A framework for query refinement with user feedback. J Syst. Softw. **86**(6), 1580–1595 (2013)
21. Islam, S., Liu, C., Zhou, R.: FlexIQ: a flexible interactive querying framework by exploiting the skyline operator. J Syst. Softw. **97**, 97–117 (2014)
22. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Trans. Inf. Syst. **20**(4), 422–446 (2002)
23. Kim, M.S., Kong, Y.H.: Ontology-DTD matching algorithm for efficient XML query. In: Fuzzy Systems and Knowledge Discovery, pp. 3614:1093–1102 (2005)
24. Kim, M.S., Kong, Y.H., Jeon, C.W.: Remote-specific XML query mobile agents. In: Data Engineering Issues in E-Commerce and Services, pp. 4055:143–151 (2006)
25. Little, E., Sambhoos, K., Llinas, J.: Enhancing graph matching techniques with ontologies. In: FUSION, pp. 1–8 (2008)
26. Lu, Y., Wang, W., Li, J., Liu, C.: Xclean: providing valid spelling suggestions for XML keyword queries. In: ICDE, pp. 661–672 (2011)
27. Mei, J., Ma, L., Pan, Y.: Ontology query answering on databases. In: International Semantic Web Conference, pp. 445–458 (2006)
28. Meng, X., Cao, L., Shao, J.: Semantic approximate keyword query based on keyword and query coupling relationship analysis. In: CIKM, pp. 529–538 (2014)
29. Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.: Wordnet: an on-line lexical database. Int. J. Lexicogr. **3**, 235–244 (1990)
30. Muslea, I.: Machine learning for online query relaxation. In: KDD, pp. 246–255. ACM, New York (2004)

31. Muslea, I., Lee, T.J.: Online query relaxation via Bayesian causal structures discovery. In: AAAI, pp. 831–836. AAAI Press (2005)
32. Nambiar, U., Kambhampati, S.: Answering imprecise queries over autonomous web databases. In: ICDE, p. 45 (2006)
33. Pu, K.Q., Yu, X.: Keyword query cleaning. PVLDB **1**(1), 909–920 (2008)
34. Schenkel, R., Theobald, A., Weikum, G.: Semantic similarity search on semistructured data with the XXL search engine. Inf. Retr. **8**(4), 521–545 (2005)
35. Sun, C., Chan, C.Y., Goenka, A.K.: Multiway Slca-Based Keyword Search in XML Data. In: WWW, pp. 1043–1052 (2007)
36. Truong, B.Q., Bhowmick, Dyreson, C., Sun, A.: MESSIAH: missing element-conscious SLCA nodes search in XML data. In: SIGMOD, pp. 37–48 (2013)
37. Wu, Y., Yang, S., Yan, X.: Ontology-based subgraph querying. In: ICDE, pp. 697–708 (2013)
38. Wu, Z., Palmer, M.: Verbs semantics and lexical selection. In: ACL, pp. 133–138, Stroudsburg, PA, USA (1994)
39. Xu, J., Croft, W.B.: Query expansion using local and global document analysis. In: SIGIR, pp. 4–11 (1996)
40. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest Lcas in XML databases. In: SIGMOD, pp. 537–538 (2005)
41. Yao, J., Cui, B., Hua, L., Huang, Y.: Keyword query reformulation on structured data. In: ICDE, pp. 953–964 (2012)
42. Yao, L., Liu, C., Li, J., Zhou, R.: Efficient computation of multiple XML keyword queries. In: WISE, pp. 368–381 (2013)
43. Zhou, J., Bao, Z., Wang, W., Zhao, J., Meng, X.: Efficient query processing for xml keyword queries based on the IDList index. VLDB J. **23**(1), 25–50 (2014)
44. Zhou, X., Guagaz, J., Balke, W.T., Nejdl, W.: Query relaxation using malleable schemas. In: SIGMOD, pp. 1:545–556 (2007)