The VLDB Journal (2015) 24:493–518
DOI 10.1007/s00778-015-0386-1

REGULAR PAPER

A general framework to resolve the MisMatch problem in XML keyword search

Zhifeng Bao[1] · Yong Zeng[2] · Tok Wang Ling[2] · Dongxiang Zhang[2] · Guoliang Li[3] · H. V. Jagadish[4]

Received: 26 May 2014 / Revised: 30 March 2015 / Accepted: 30 March 2015 / Published online: 18 April 2015
© Springer-Verlag Berlin Heidelberg 2015

Abstract When users issue a query to a database, they have expectations about the results. If what they search for is unavailable in the database, the system will return an empty result or, worse, erroneous mismatch results. We call this problem the *MisMatch problem*. In this paper, we solve the MisMatch problem in the context of XML keyword search. Our solution is based on two novel concepts that we introduce: *target node type* and *Distinguishability*. *Target Node Type* represents the type of node a query result intends to match, and *Distinguishability* is used to measure the importance of the query keywords. Using these concepts, we develop a low-cost post-processing algorithm on the results of query evaluation to detect the MisMatch problem and generate helpful suggestions to users. Our approach has three noteworthy features: (1) for queries with the MisMatch problem, it generates the explanation, suggested queries and their sample results as the output to users, helping users judge whether the MisMatch problem is solved without reading all query results; (2) it is portable as it can work with any lowest common ancestor-based matching semantics (for XML data without ID references) or minimal Steiner tree-based matching semantics (for XML data with ID references) which return tree structures as results. It is orthogonal to the choice of result retrieval method adopted; (3) it is lightweight in the way that it occupies a very small proportion of the whole query evaluation time. Extensive experiments on three real datasets verify the effectiveness, efficiency and scalability of our approach. A search engine called XClear has been built and is available at http://xclear.comp.nus.edu.sg.

Keywords XML · Keyword search · MisMatch problem

✉ Zhifeng Bao
zhifeng.bao@rmit.edu.au

✉ Yong Zeng
zengyong8@gmail.com

Tok Wang Ling
lingtw@comp.nus.edu.sg

Dongxiang Zhang
zhangdo@comp.nus.edu.sg

Guoliang Li
liguoliang@tsinghua.edu.cn

H. V. Jagadish
jag@umich.edu

1 School of Computer Science and Information Technology, RMIT University, Melbourne 3000, Australia

2 School of Computing, National University of Singapore, Singapore 117417, Singapore

3 Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

4 Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA

1 Introduction

When users issue a query to a database, they have expectations about the results. If what they search for is unavailable in the database, due to reasons such as product removed from shelves, clothes size unavailable, the result they seek may not be found in the database. In such a case, the system may return an empty result or, worse, return erroneous results. We call this the *MisMatch problem*.

For example, a user wants to search for a laptop. She wants the model Vaio W with color being red. If red color is unavailable for laptop Vaio W in the database, then obviously the user will not get what she wants no matter how the data is organized or what kind of query it is.

The MisMatch problem is a natural and common problem. It can happen in any form of information retrieval over data of any structure, i.e., can be either structured query or key-

word query on structured, unstructured and semi-structured data. Such a problem has attracted a lot of research effort in the context of structured queries on structured data [6,17,29–31], with descriptions such as failing queries and non-answer queries. However, no such work has been done in the context of keyword search on semi-structured data. This is an important area to address. According to our experiments conducted on XClear, an XML keyword search engine available at [41], users suffered from such a problem for 27 % of their queries.

What can we offer to help the user? Ideally, we can get the following help if we are interacting with a human:

1. Notification: "Sorry, we do not have such a product."
2. Explanation: "Because red color is unavailable for Vaio W."
3. Suggestion: "You can choose some other available colors: white, blue and pink."

When structured queries are issued over structured data (relational tables), the MisMatch problem (i.e., what users search for is unavailable in the database) leads to empty result. Detecting the problem is trivial because empty result is obvious. A message (notification part) will be given to users. Some existing works [6,17] try to explain the non-answer queries by pinpointing the constraint causing the empty result (explanation part). Some works [29–31] focus on generating some alternative constraints to come up with some suggested queries (suggestion part).

When keyword queries are issued over unstructured data (in web search), the MisMatch problem will lead to a list of mismatch results. It is even difficult to detect the problem in the first place because most likely the results being returned are not empty. It could be the case that the query keywords appearing in one document are far away from each other and not semantically related. E.g., for a keyword query 'Vaio W red', if color red is not available for laptop Vaio W, there still can be many webpages being returned, where 'Vaio W' appears in one part of the webpage while 'red' appears in another part of the webpage. It leads to mismatch results. Therefore, we need to analyze whether the keywords are 'semantically' related in the results. Such analysis is challenging because the data is unstructured. A limited solution to a part of the problem (only the suggestion generation part) is to mine some similar and popular queries from query log [19,42] and show them to users (suggestion part). But the downside is that such popular queries do not guarantee to have reasonable results.

In this work, we focus on identifying and solving the MisMatch problem in the context of keyword search over semi-structured data, i.e., XML data. Most research works on XML are done based on XML data without ID references. In this paper, we also focus on solving the MisMatch problem on XML keyword search without considering ID references. But later in Sect. 6, we will extend our MisMatch problem solution onto XML keyword search which considers ID references, where the matching semantics are different. In the rest of the paper, without specification, we are talking about XML without ID references by default. We will only talk about XML data with ID references in and after Sect. 6.

Now, let us take a look at how the MisMatch problem behaves in the context of XML keyword search.

Example 1 An XML data tree in Fig. 1 describes the item information of an online shopping mall. Suppose a user wants to buy a laptop. She prefers Sony's Vaio W with red color and wants to know how much it is. Then she may issue a query $Q = \{$'Vaio','W','red','price'$\}$ to search for a laptop. Unfortunately, no laptop can meet all her requirements. Vaio W only has three colors: white, blue and pink. Existing keyword search methods, such as LCA (lowest common ancestor) [33], SLCA [38], ELCA [12] or even the most recent variant [22] of LCA, still can find some results containing all query keywords. One of the query results is the subtree rooted at *shop*:0.0.0, where keyword 'red' matches one laptop, while the rest keywords match another laptop. Obviously, the subtree rooted at *shop* is not expected by the user, as it contains too much irrelevant information, i.e., all laptops. What is

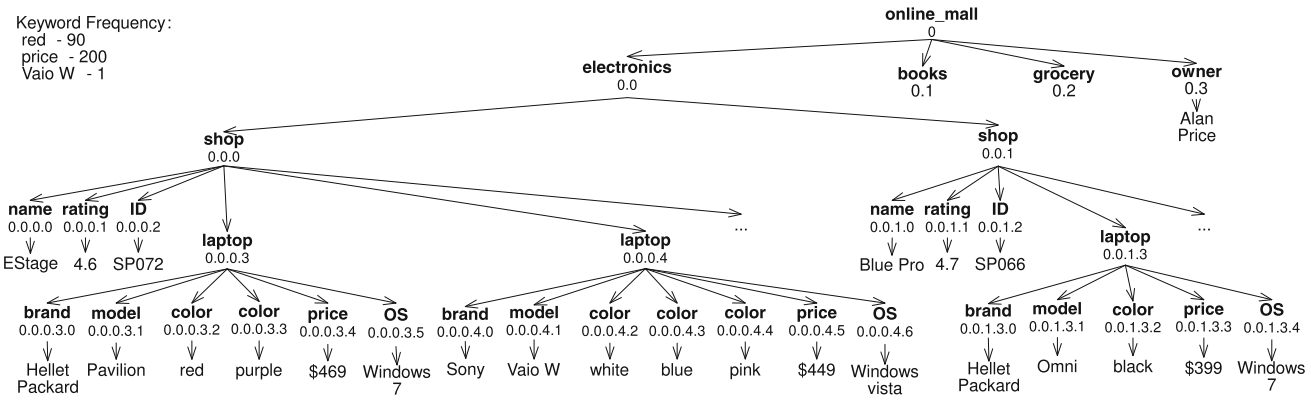

Fig. 1 Sample XML document about an online shopping mall

worse, there could be hundreds of shops selling Vaio W and therefore hundreds of mismatch results are returned. In this case, imagine if the user was interacting with a salesman, she would be informed of the unavailability of the product and suggested with some other available colors for the laptop Vaio W.

As we can see, the MisMatch problem in XML keyword search also leads to a list of mismatch results. It poses three challenges for a search engine to help users: (1) how to design a detection method to distinguish queries with the MisMatch problem from those without; (2) how to explain why the query leads to mismatch results; (3) how to find good suggestions, and what should be a good way to present them to users.

Our solution to the MisMatch problem is to run a small post-processing job at the end of the query evaluation, consisting of two components, namely *detector* and *suggester*. The former addresses the first challenge above, and the latter addresses the remaining two.

The central idea of our technique for mismatch detection is based on the notion of *Target Node Type* (see Sect. 2 for the formal definition). Intuitively, Target Node Type denotes the type of node a query result r intends to match. We calculate it at schema level. Meanwhile, the actual root of result r is calculated at data level by existing techniques. If r's root does not match its Target Node Type, we claim that r misses the target. We can perform a similar check on all results of a query Q. If all results of a query Q miss their targets, then we say that Q has the MisMatch problem.

Once a mismatch is detected, we propose a concept called *distinguishability* to find 'important' keywords in the original query, and use these to explain the reason for the mismatch and to suggest possible relaxations. Distinguishability is inspired by the *tf*idf* scoring measure proposed in IR [32] while taking the structural property of XML data into account. Then based on each query result r we try to find some 'approximate' query results, which contain these 'important' query keywords and are structurally consistent with r, while having reasonable replacement for the rest 'less-important' query keywords. Finally, the explanation and suggested queries can be inferred from the approximate results. To further improve the user experience, our suggester also generates a sample result for each suggested query Q' even without evaluating the query Q', which helps users to judge whether Q' is helpful.

Putting these together, we have our complete algorithm. The input is a (ranked) list of all results returned by search engine. For a user query that has the MisMatch problem, the output of our algorithm consists of three parts:

1. An explicit notification to user: "what you search for is not available".
2. An explanation on which keyword(s) in the query leads to mismatch results.
3. Some data-driven suggested queries, which guarantee to have reasonable results.

Note that the quality of query results is always the holy grail of keyword search problem, especially for those queries that have ambiguity problem. As no existing approach can solve the ambiguous query thoroughly [2], our solution cannot guarantee the detection of the MisMatch problem to be hundred percent correct as well, which has actually been studied in our experiments as well. That also explains why we adopt a post-processing without affecting the results of the original query, i.e., the results of their original query will be returned together with our generated suggestion.

There are many possible relaxations of a given query, and many of these may themselves also be empty (result in mismatch). It is important to ensure that the suggestions given have at least some results and are not mismatch themselves.

As discussed in the related work section below, there is a great body of work on query relaxation and on generating partial match answers. These systems, while valuable, do not address all three of the challenges we described above, and hence are not suited for our problem context. In particular, many of them generate large lists of possible partial match answers that the user has to wade through even to realize that there is a mismatch at all.

In summary, our major contributions include:

1. We identify the MisMatch problem in XML keyword search. We detect the MisMatch problem by investigating into the query results and inferring the Target Node Type for each query result. It is *portable* as it can work with any LCA-based matching semantics and is orthogonal to the choice of result retrieval method.
2. We design a *data-driven* approach to generating explanation and suggested queries by finding approximate query results, which contain important keywords in the original query Q while having consistent structure with the results of Q. We propose distinguishability, which is a structure-aware tf*idf scoring measure, to quantify the importance of keywords.
3. We propose a novel bitmap-based labeling scheme to accelerate finding approximate results. As a result, the MisMatch detector and suggester is *lightweight*: it takes only 4 % of the whole query processing time.
4. We further extend the detector and suggester to handle XML data with ID references, where the data is no longer a tree structure and the search semantics/methods are different, i.e., minimal Steiner tree-based search semantics/methods. This results in a general framework solving the MisMatch problem, which is *portable*, *data-driven* and *lightweight*.

5. We build a search engine called XClear [41] which embeds the above MisMatch problem detector and suggester and is capable of handling any XML data with/without ID references. Extensive experiments have verified the effectiveness, efficiency and scalability of our method.

We first solve the MisMatch problem over XML data without ID references. Detecting the MisMatch problem is in Sect. 2. Section 3 discusses how to find the explanations and suggested queries. Section 4 presents our labeling scheme for efficient approximate results detection. Section 5 presents indices and algorithms. Then we further extend our MisMatch solution onto XML data with ID references in Sect. 6. Experiments are in Sect. 7. Related works are in Sect. 8, where we also state the difference between this work and our previous work in [39]. Then we conclude with future work in Sect. 9.

2 Detecting the MisMatch problem over XML data without ID references

In this section, we would like to present how to detect the MisMatch problem over XML data without ID references. First, we model the XML document and the query results (Sect. 2.1). Then we will talk about how the detector will infer user's possible search target(s) based on the query results (Sect. 2.2). Finally the MisMatch problem can be easily detected by checking the inferred search target(s) and the query results (Sect. 2.3).

2.1 Preliminaries

2.1.1 Data model

We model a data-centric XML document without ID references as a rooted, labeled and ordered tree. Each node of the tree corresponds to an element of the XML data, and it has a tag name and (optionally) some value. Without loss of generality, we simply use the word "node" to mean the node in an XML tree. To accelerate the keyword query processing, all existing works adopt the dewey labeling scheme [37]. As shown in Fig. 1, for a node n, its dewey label consists of a sequence of components that implicitly contain all ancestor nodes on the path from the document root to n. E.g., from $laptop$:0.0.0.3, it is easy to find that the label of its parent is 0.0.0.

Definition 1 (*Node Type*) The type of a node n in an XML tree, denoted as $n.type$, is the tag name path from root to n.

In the rest of the paper, the tag name of n is used to represent the node type of n if no ambiguity is caused.

Definition 2 (*Keyword Match Node*) A node n is called a *keyword match node* for a keyword k if the tag name or the value part of n contains k.

Definition 3 (*Subtree-contain*) A node n is said to *subtree-contain* another node m if n equals to m or there exists a directed path from n to m. n is also said to *subtree-contain* the keywords in m's tag name or value part.

E.g., in Fig. 1, the node type of $laptop$:0.0.0.3 is $online\_mall/electronics/shop/laptop$; $color$:0.0.0.3.2 is a *keyword match node* w.r.t. keyword 'red'; $laptop$:0.0.0.3 is said to *subtree-contain* node $color$:0.0.0.3.2; $laptop$:0.0.0.3 is also said to *subtree-contain* keyword 'red' as 'red' is the value part of $color$:0.0.0.3.2.

2.1.2 Query result format

To define a general format to represent the query results, let us look at the existing matching semantics first. Existing matching semantics so far, such as SLCA [15,38], ELCA [12], entity-based SLCA [26] are all based on the concept of lowest common ancestor (LCA). Let $lca(m_1, \ldots, m_n)$ be the lowest common ancestor of nodes $m_1, \ldots, m_n$. For a given query $Q = \{k_1, \ldots, k_n\}$ and an XML document D, L_i denotes the inverted list of k_i. Then the LCAs of Q on D are defined as $LCA(Q) = \{v | v = lca(m_1, \ldots, m_n), m_i \in L_i (1 \leq i \leq n)\}$. Both SLCA and ELCA define a subset of $LCA(Q)$, and we refer readers to Sect. 8 for detailed definitions of SLCA and ELCA, and their relationships with LCA.

Definition 4 (*Query Result Format*) For a keyword query $Q = \{k_1, \ldots, k_n\}$, we define the format of a query result r as:

$$r = \left(v_{lca}, \left\{m_1, m_2, \ldots, m_n\right\}\right)$$

where m_i is a *keyword match node* w.r.t. keyword k_i ($i \in [1, n]$), and v_{lca} is the lowest common ancestor of nodes $m_1, \ldots, m_n$, i.e., $v_{lca} = lca(m_1, \ldots, m_n)$.

Definition 4 is highly general in two aspects: (1) it is *compatible* with any existing LCA-based matching semantics adopted by search engines, because one necessary condition for a node v to be an SLCA (or ELCA) node of a query Q is: v must be a lowest common ancestor of a set of *keyword match nodes* m_i w.r.t. Q. (2) Our query result format forms the skeleton for both Path Return (returning the paths in the XML tree from each LCA node to its *keyword match nodes*) [15,24] and subtree Return (returning the subtree rooted at each LCA node) [12,38]. This observation is important in explaining the *portability* feature of our solution to detect and resolve the MisMatch problem later in Sect. 3.4.

2.2 Inferring the target node type for a query result

In this section, we are going to infer user's search target(s) based on the query results. It is obvious that user's search intention is not dependent on the availability of the data. Normally, a user issues a keyword query with her search target in mind but without knowledge of the availability of the data. So user's search intention should be inferred by analyzing the semantics of her query keywords. However, existing LCA-based result retrieval methods generate the query results purely at data level without inferring user's search intention at all. Therefore, we propose to use node types to simulate the semantics of the *keyword match nodes*, which are matched by the query keywords. By analyzing those node types, we will infer user's search intention at schema level.

Since a keyword can match different types of nodes, user's search target may be various for a certain query. E.g., keyword "price" can match an owner's name (node *owner*:0.3) or the price of a product (node *price*:0.0.0.3.4) in Fig. 1. But a certain query result r corresponds to a unique search target because each query keyword has a unique corresponding *keyword match node* in a given query result r. Therefore, we introduce a concept called *Target Node Type (TNT)* to denote the node type in which a query result r intends to match.

To infer the TNT of a result r, we propose to use node types to simulate the semantics of each *keyword match node*.

Example 2 For the query $Q = \{`Vaio', `W', `red', `price'\}$ in Example 1, if the user is interacting with a salesman, the salesman will know that the user is finding a laptop because the salesman knows the meaning of each query keyword. Here for XML keyword search, one result is $r = (0.0.0, \{0.0.0.4.1, 0.0.0.4.1, 0.0.0.3.2, 0.0.0.4.5\})$. We use node types to simulate the semantics of each *keyword match node*. The node types of each distinct *keyword match node* are ('Vaio' and 'W' match the same node):

0.0.0.4.1:
{*online_mall/electronics/shop/laptop/model*}
0.0.0.3.2:
{*online_mall/electronics/shop/laptop/color*}
0.0.0.4.5:
{*online_mall/electronics/shop/laptop/price*}.

Then we can know that the user inputs three kinds of information: laptop model, laptop color and laptop price. The user's search intention, i.e., a laptop, corresponds to the node type "online_mall/electronics/shop/laptop", which is closely related to the above three node types.

Following a similar philosophy of LCA, which finds the lowest/smallest nodes connecting all query keywords as the most relevant results, we define the lowest node type which connects to all the above node types at schema level as the Target Node Type, where the formal definition will be introduced later. It is the most relevant node type connecting to user's input information. E.g., in Example 2, {*online_mall/electronics/shop/laptop*} is the lowest node type connecting to laptop model, laptop color and laptop price at schema level even though no laptop can meet all the requirements at data level.

However, an XML document actually comes with some constraints on how many nodes of a type t_a can be subtree-contained by another node of type t_b. E.g., a laptop node (of node type *online_mall/electronics/shop/laptop*) can subtree-contain only one laptop model node (of node type *online_mall/electronics/shop/laptop/model*) while it can subtree-contain more than one laptop color node (of node type *online_mall/electronics/shop/laptop/color*). Similarly, a shop node (of node type *online_mall/electronics/shop*) can subtree-contain multiple laptop node and therefore a shop node also can subtree-contain multiple laptop model nodes.

Such constraints will affect the inferring of Target Node Type when we try to find the lowest node type connecting to user's input information at a schema level.

Example 3 Suppose a user wants to find a shop selling two laptop models, both model Pavilion and Omni produced by Hewlett Packard, she may issue a query Q = {'Hewlett', 'Packard', 'Pavilion', 'Omni'} in Fig. 1, which contains two different laptop model names. If the user is interacting with a salesman, the salesman will know the user is not finding a particular laptop but something related to two different laptops, e.g., a shop selling those two laptops. Here in terms of XML keyword search, one query result is a subtree rooted at an *electronics* node: r=(0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.1.3.1}). The node types of each distinct *keyword match node* are ('Hewlett' and 'Packard' match the same node):

0.0.0.3.0:
{*online_mall/electronics/shop/laptop/brand*}
0.0.0.3.1:
{*online_mall/electronics/shop/laptop/model*}
0.0.1.3.1:
{*online_mall/electronics/shop/laptop/model*}.

The user's keywords are describing one laptop brand and two different laptop models, i.e., the user inputs two different laptop names matching two different laptop models. In such a case, assuming one laptop can only have one laptop model name in the data, the lowest node type connecting the above three node types is no longer *online_mall/electronics/shop/laptop* because there are

two different laptop model nodes here while a laptop node can subtree-contain only one laptop model node. Instead, the lowest node type connecting the above three node types is *online_mall/electronics/shop* because a shop node can subtree-contain multiple laptop model nodes. Therefore, we can infer that the search intention is to find a shop selling two laptop models rather than finding a particular laptop model.

The containment constraints among different types of nodes can be easily inferred from the schema of the XML document. For example, DTD is a commonly used XML schema language, where operators * (zero or more occurrences), + (one or more occurrences), ? (zero or one occurrence) are used to specify the occurrence constraints of sub-elements or attributes under a particular type of node. Let $t_1.maxContain(t_2)$ be the maximum number of nodes of type t_2 which can be subtree-contained by another node of type t_1. The range is $[0, +\infty]$. E.g., in Fig. 1, if a laptop node can subtree-contain at most one laptop model node, then we have $laptop.maxContain(model) = 1$; if a shop node can subtree-contain multiple laptop nodes, we have $shop.maxContain(laptop) = +\infty$; besides, since shop node is the parent of laptop node and laptop node is the parent of model node, we can further infer $shop.maxContain(model) = +\infty$ by multiplying the above two values. Such a calculation can be done offline based on the schema. If the schema of the XML document is unavailable, we can infer such constraints approximately by scanning the XML document to summarize a DataGuide [11][1].

Now we need to count the number of occurrences of each different node type for the *keyword match nodes*. Let $\mathbb{T} = \{t_1, t_2, \ldots, t_x\}$ be a set of different node types for the *keyword match nodes*. As some of the *keyword match nodes* could be of the same node type, let $count(t_i)$ be the number of *keyword match nodes* which are of type t_i. E.g. for the query result in Example 3, there are three distinct *keyword match nodes*: two of them are of type *online_mall/electronics/shop/laptop/model* and one of them is of type *online_mall/electronics/shop/laptop/brand*. In this case, $\mathbb{T} = \{brand, model\}$, $count(brand) = 1$ and $count(model) = 2$.

We are trying to find the most relevant node type connecting to user's input as the TNT, i.e., to find the lowest node type which can connect to all the node types that are matched by the query keywords. Next we will define the Target Node Type of a result r formally.

First, TNT should be related to and connecting to each node type in $\mathbb{T}$, i.e., the TNT should be a common prefix of the node types in $\mathbb{T}$. Second, a node of the Target Node Type should be able to subtree-contain all occurrences of each node type in $\mathbb{T}$. Last, TNT should be as low as possible such that it can connect to each node type in $\mathbb{T}$ as closely as possible. So we define the extended TNT formally as follows:

Definition 5 (*Target Node Type (TNT) for a single query result*) Given a query $Q = \{k_1, k_2, \ldots, k_n\}$ and a query result $r = (v_{lca}, \{m_1, m_2, \ldots, m_n\})$ on an XML document D, let $\mathbb{T} = \{t_1, t_2, \ldots, t_x\}$[2] be the set of different node types for m_1 to m_n, the Target Node Type $TNT(r)$ for result r is defined as:

$$TNT(r) = t$$

such that t satisfies the following 3 conditions

Condition 1: $t \in commonPrefix(t_1, t_2, \ldots, t_x)$;
Condition 2: $t.maxContain(t_i) \geq count(t_i), i \in [0, x]$;
Condition 3: $\nexists t'$ such that t' is a descendent of t and t' also satisfies condition 1 and condition 2,

where $commonPrefix(t_1, t_2, \ldots, t_x)$ represents all possible common prefixes for a set of node types; $t.maxContain(t_i)$ represents the maximum number of t_i type nodes which can be subtree-contained by a t type node; $count(t_i)$ represents the number of different *keyword match nodes* in m_1 to m_n which are of node type t_i.

TNT is the lowest node type which can connect to all the node types that are matched by the query keywords. It is defined at the schema level by making use of node types, no matter whether what users search for exist in the XML document at data level or not. To calculate the TNT for a given result, we check the prefixes of each node type in $\mathbb{T}$ from the lowest one upwards, see whether it satisfies condition 2.

In the following two examples, we will infer the TNT according to the above definition for two sample queries, both of which are with the MisMatch problem, i.e., what users search for is unavailable in the data. In Example 4, all *keyword match nodes* are of different node types; in Example 5, some *keyword match nodes* are of the same node type.

Example 4 For the query $Q = \{$'*Vaio*','*W*','*red*','*price*'$\}$ in Example 1, one of the results is $r = (0.0.0, \{0.0.0.4.1, 0.0.0.4.1, 0.0.0.3.2, 0.0.0.4.5\})$, where the node types of each distinct *keyword match node* are ('Vaio' and 'W' match the same node):

0.0.0.4.1: {*online_mall/electronics/shop/laptop/model*}
(denoted as t_1)

[1] Specifically, we use strong DataGuide proposed in [11]. Our purpose of using DataGuide is only to provide a data structure to store the occurrence constraints summarized from the XML data. Many other data structures are possible.

[2] t_i may not necessarily be a one-to-one mapping to m_i because two *keyword match nodes*, say m_i and m_j, could be of the same node type.

0.0.0.3.2: {*online_mall/electronics/shop/laptop/color*} (denoted as t_2)

0.0.0.4.5: {*online_mall/electronics/shop/laptop/price*} (denoted as t_3).

The set of distinct node types $\mathbb{T} = \{t_1, t_2, t_3\}$, where $count(t_1) = 1$, $count(t_2) = 1$ and $count(t_3) = 1$.

Then we check the prefixes of all node types in $\mathbb{T}$. The lowest one is $t =$ "*online_mall/electronics/shop/laptop*". Suppose we have the following constraints (either inferred from the XML schema or by scanning the XML document): one laptop node can subtree-contain one model node, one price node and multiple color nodes, and then it will satisfy: $t.maxContain(t_1) = 1 \geq count(t_1) = 1$, $t.maxContain(t_2) = +\infty \geq count(t_2)$ and $t.maxContain(t_3) = 1 \geq count(t_3)$.

Therefore, $TNT(r) = t =$ "*online_mall/electronics/shop/laptop*" even though no laptop can meet all the user's requirements at data level. It is the lowest node type which can connect to all the node types that are matched by the query keywords.

Example 5 For the query $Q = \{$'*Hewlett*','*Packard*', '*Pavilion*', '*Omni*'$\}$ in Example 3, since there is no shop selling both of these models in Fig. 1, the results being returned are not shops as expected by the user. One of the results is $r =$ (0.0, {0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.1.3.1}), where the node types of each distinct *keyword match node* are ('Hewlett' and 'Packard' match the same node):

0.0.0.3.0 : {*online_mall/electronics/shop/laptop/brand*} (denoted as t_1)

0.0.0.3.1 : {*online_mall/electronics/shop/laptop/model*} (denoted as t_2)

0.0.1.3.1 : {*online_mall/electronics/shop/laptop/model*} (denoted as$t_2$which is the same as the previous one).

The set of distinct node types $\mathbb{T} = \{t_1, t_2\}$, where $count(t_1) = 1$ and $count(t_2) = 2$.

Then we check the prefixes of all node types in $\mathbb{T}$. The lowest one is $t =$ "*online_mall/electronics/shop/laptop*". Suppose we have the following constraints (either inferred from the XML schema or by scanning the XML document): one laptop node can subtree-contain one brand node and one model node, and then we will have: $t.maxContain(t_1) = 1 \geq count(t_1) = 1$ but $t.maxContain(t_2) = 1 \ngeq count(t_2) = 2$. As we can see, t is not the TNT as a laptop node cannot subtree-contain two model nodes.

Then we will check another prefix $t' =$ "*online_mall/electronics/shop*", which is just above t. Suppose we have the following constraints in the XML document (either inferred from the XML schema or by scanning the XML document): one shop node can subtree-contain multiple brand nodes while it can subtree-contain multiple model nodes, and then we have: $t'.maxContain(t_1) = +\infty \geq count(t_1) = 1$ and $t.maxContain(t_2) = +\infty \geq count(t_2) = 2$.

Therefore, Target Node Type of result r is $TNT(r) = t' =$ "*online_mall/electronics/shop*". It is the lowest node type which can connect to all the node types that are matched by the query keywords.

Our solution assumes there is no outer semantics provided because usually XML data exists without such information, so that we use node types to simulate semantics, where two nodes of the same type will be with the same semantics. If we do have outer semantics, like thesaurus, ontology, etc., we can further improve our approach such that we can even tell that node types "/laptop/color" and "/notebook/color" are with the same semantics while node types "/owner/name" and "/product/name" are with different semantics. This will be one of our future work.

Similar to how a salesman understands a customer's request (in Example 2), user's search intention should be inferred by analyzing the semantics of users' query keywords w.r.t. both the meta-data and the data. The intuitive idea of Target Node Type is to use node types (at meta-data level) to simulate the semantics of each keyword match node, which contains the matching query keyword(s) (at data level). By analyzing those node types, we can intuitively infer users' search intention.

2.3 Detecting the MisMatch problem based on target node type

With the Target Node Type of a query result r being inferred, the detector should figure out whether there is a mismatch between the TNT of r and the actual root of r, namely v_{lca}. If what users search for exists in the data, these two should be consistent to each other.

Definition 6 Given a query $Q = \{k_1, k_2, \ldots, k_n\}$ and a query result $r = (v_{lca}, \{m_1, m_2, \ldots, m_n\})$ on the XML data D, if v_{lca} is not of the same node type as $TNT(r)$, the query result r misses the target.

For result r in Example 4, $v_{lca}.type = shop \neq laptop = TNT(r)$, so we say r misses the target. Now, we can formally define the MisMatch problem.

Definition 7 (*MisMatch problem*) Given a query Q and its results $\mathbb{R}$ retrieved from the keyword search engine, Q has the MisMatch problem if all $r \in \mathbb{R}$ misses the target.

As we mentioned earlier in Sect. 2.2, different users have different search targets for a certain query. Here we choose to take a conservative approach: we only judge a query to have the MisMatch problem when there is a mismatch for all possible search intentions. Such a conclusion holds for all users with different intentions. For

example., for the result r in Example 4, we inferred that it misses the target. In a similar way, we will also calculate a TNT for each of the other results (if any). We will claim that the query has the MisMatch problem only if all the results miss their corresponding target.

Moreover, users usually investigate the retrieved results starting from the *top-ranked* ones. Therefore, without loss of generality, we can also easily extend Definition 7 by considering the top-K retrieved results of Q, such that we can adjust the aggressiveness of the approach by changing the value K. K can be as aggressive as 1 or can be as tolerant as the total number of results of query.

Time complexity of the detector is O(|$\mathbb{R}$|), which is very efficient. As discussed in Sect. 5.1 later, we store the type information of each node when building the keyword inverted list. Thereby for each $r \in \mathbb{R}$, $TNT(r)$ can be computed in O(1) time, assuming the number of keywords in a query and the depth of the XML tree are bounded by some constants.

Summary of underlying intuitions:

Note that, when user specifies a structured query like SQL, they need to specify their target in select clause and predicates in where clause. As an analogy, the node type (Definition 1) is an implicit representation of predicates specified in a structured query, while the difference is that in a keyword query you have no way to specify constraint on the structural relationship among keywords. Thereby, we try to collect the possible search predicates (in the form of the node types associated with each query keyword) in a keyword query, that comes out Definition 5. Since our primary goal is to detect the query that has MisMatch problem, we exploit the inconsistency between this "target" (at structure level) and the target (i.e. root node of result r at content level) to justify whether a MisMatch problem occurs. That results in our Definition 6 "Miss the Target". The intuitive idea of MisMatch problem (in Definition 7) is: if there is a target inconsistency between structure level and content level for ALL the results of a query, we can safely conclude what users search for is unavailable in the database.

3 Explanations and suggestions for the MisMatch problem over XML data without ID references

As discussed in Sect. 2, the main feature of the MisMatch problem is: there does not exist a single *TNT node* that *subtree-contains* all query keywords. So the query keywords have to scatter in more than one TNT node and then lead to a mismatch result. As a result, the root of the returned subtree is always an ancestor of the TNT nodes which are expected by the user. Given a user query $Q = \{k_1, k_2, \ldots, k_n\}$ and a mismatch query result $r = (v_{lca}, \{m_1, m_2, \ldots, m_n\})$, where m_i is a *keyword match node* for k_i, the basic idea to find the explanations and some promising suggested queries can be illustrated in three steps.

Step 1: Since each *keyword match node* m_i in r may contain several keywords $\mathbb{K}$ in Q, we first propose a *tf*idf*-inspired heuristic called *distinguishability* to score the importance of such $\mathbb{K}$.

Step 2: We then try to find the approximate query results, i.e., $r' = (v'_{lca}, \{m'_1, m'_2, \ldots, m'_n\})$, which are some subtrees containing the 'important' keywords (derived by Step 1). An ideal approximate result r' should satisfy the following properties: (a) the node type of r' should be the same as $TNT(r)$; (b) for each *keyword match node* m_i in original result r, there always exists a node m'_i that has the same node type as m_i ($i \in [1, n]$). By such properties, it can ensure at least the structure of r' and r are consistent with each other.

Step 3: Then, we can pinpoint which keyword(s) in the user's query lead to the mismatch results, i.e., the query keywords not contained by the approximate results. This is the explanation part. We can further infer the suggested queries by replacing those keywords with the keywords associated with the aforementioned m'_i (in approximate result) in step 2.

Step 1 is illustrated in Sect. 3.1, and the last two steps are described in Sect. 3.2. Lastly, we complement our suggester by discussing how to rank the suggested queries in Sect. 3.3.

3.1 Distinguishability

In this section, we will present a concept to measure the importance of query keywords, namely *distinguishability*. We find that the importance of query keywords is closely related to what type of nodes they match. E.g., in Fig. 1, keyword 'blue' can match either a shop name *name*:0.0.1.0 or a laptop color *color*:0.0.0.4.3. When it matches a shop name, most likely it is important since few shop names contain the keyword 'blue'; when it matches a laptop color, it may be less important since many color nodes contain the keyword 'blue'. Therefore, we propose the concept of *distinguishability*.

Distinguishability $D(\mathbb{K}, t)$ represents the importance of the query keywords $\mathbb{K}$ when $\mathbb{K}$ matches a node of type t, which also means this node of type t *subtree-contains* each keyword in $\mathbb{K}$. Large $D(\mathbb{K}, t)$ means $\mathbb{K}$ is important with respect to t.

Recall Step 1 in Sect. 3, $\mathbb{K}$ actually represents the query keywords derived from the *keyword match node*(s). To quantify $D(\mathbb{K}, t)$, we propose a scoring measure inspired by Term Frequency * Inverse Document Frequency (*tf*idf*) [32], which is widely used in information retrieval.

For *tf*, we can simply count the keyword frequency in an XML node. In this work we focus on data-centric XML data, where each XML node does not contain long text and in most cases keyword frequency is 1. The same problem is also pointed out by [13], so we follow [13] and do not consider *tf* in the formula.

For *idf*, it tells that the keywords contained by fewer documents are more important. Similar to *idf*, we have Intuition 1 in the context of XML. Let f_t be the number of nodes of type t, and $f_t^{\mathbb{K}}$ be the number of nodes which are of node type t and subtree-contain each keyword in $\mathbb{K}$.

Intuition 1 $idf(\mathbb{K}, t)$. *If few nodes of type t contain keywords* $\mathbb{K}$, $\mathbb{K}$ *should be important with respect to the node type t. Formally, the smaller the* $f_t^{\mathbb{K}}$ *is as compared to* f_t, *the larger the* $idf(\mathbb{K}, t)$ *should be.*

As there are many variants of *idf* to follow Intuition 1, we define $idf(\mathbb{K}, t) = 1 - \frac{f_t^{\mathbb{K}}}{f_t}$. In this way, $idf(\mathbb{K}, t)$ is normalized in [0,1).

The *tf*idf* works by assuming there is only one type of (flat) document, but in the context of XML data there is more than one type of node. The type of the node alone may also contribute to the importance of the keywords that match the node. Let us look at a motivating example first.

Example 6 Consider a keyword 'price' in Fig. 1. It can match both an *owner* node and all *price* nodes. When 'price' matches a price node, it may not be important as there are many price nodes and all of them contain 'price'. Accordingly, $idf(\{\text{'price'}\},\text{price})=0$ because $f_t^{\mathbb{K}}=f_t$. When it matches the owner node, it should be important as there is one and only one owner across the whole XML data. But since $f_t^{\mathbb{K}}=f_t=1$, $idf(\{\text{'price'}\},\text{owner})=0$ as well. As we can see, simply by *tf*idf*, we cannot distinguish these two cases (*idf* is 0 for both cases) because the idea of *tf*idf* assumes that there is only one type of node while we have nodes of different types, and we need to consider the weight of different node types.

So we have Intuition 2 to cater for the *node type weight (ntw)*.

Intuition 2 $ntw(t)$. *The weight of a node type t is inversely proportional to* f_t *within the XML data.*

Therefore, we define $ntw(t) = \frac{1}{f_t}$. Finally, we can define $D(\mathbb{K}, t)$ to capture the concept of distinguishability as:

$$D(\mathbb{K}, t) = idf(\mathbb{K}, t) + ntw(t) = 1 - \frac{f_t^{\mathbb{K}}}{f_t} + \frac{1}{f_t}\left(1 \le f_t^{\mathbb{K}} \le f_t\right) \quad (1)$$

It is easy to verify that the range of distinguishability is (0,1]. Note that there can be some alternative formulas which are able to simulate distinguishability above. It is one of our future work to study those possibilities and the differences between them.

3.2 Finding explanation and suggested queries

In order to find the explanation and suggested queries, we first need to find some 'important' query keywords (in terms of distinguishability) from the result r of the original query. So first of all, we need to set a threshold τ,[3] say τ=90 %. Those keywords whose distinguishability is higher than τ are considered as 'important' and must be kept. Besides, we find that those 'important' keywords $\mathbb{K}$ are indeed derived from the *keyword match node(s)* of r. Therefore, we may need to consider two independent cases at the same time:

(1) $\mathbb{K}$ is derived from a single *keyword match node* of r;
(2) $\mathbb{K}$ is derived from multiple *keyword match nodes* of r, i.e., combing the keywords from multiple *keyword match nodes* could achieve high distinguishability.

Then the remaining task is to find the approximate results, each containing the important keywords $\mathbb{K}$, from which suggested queries are inferred.

3.2.1 Phase 1: based on single keyword match node

In Phase 1, we derive important keywords from a single *keyword match node* and find the approximate results as follows:

Given a user query Q and a mismatch query result $r = (v_{lca}, \{m_1, m_2, \ldots, m_n\})$, each *keyword match node* m_i contains some keyword(s) $\mathbb{K}_i$ in Q. For each distinct m_i, we calculate the distinguishability $D(\mathbb{K}_i, m_i.type)$. If it is larger than the threshold, then we try to find a TNT node containing m_i as an approximate result. Let the path from v_{lca} to m_i be $(v_{lca}/p_1/p_2/\ldots/p_j/m_i)$, where $p_1, p_2, \ldots, p_j$ are the nodes between v_{lca} and m_i. Then we proceed to traverse each node v'_{lca} from p_1 down to m_i (i.e., $v'_{lca} \in \{p_1, p_2, \ldots, p_j, m_i\}$) and verify whether the subtree rooted at v'_{lca} can form an approximate query result $r'=(v'_{lca}, \{m'_1, m'_2, \ldots, m'_n\})$ w.r.t. r.

Definition 8 (*Approximate Result*) Given a query result $r=(v_{lca}, \{m_1, m_2, \ldots, m_n\})$ for a query Q, $r' = (v'_{lca}, \{m'_1, m'_2, \ldots, m'_n\})$ is an approximate result if r' have the following two properties:

- **P1**: $v'_{lca}.type = TNT(r)$
- **P2**: $\forall m_i$ in the original result r, $\exists m'_i$ in r', such that $m'_i.type = m_i.type$, for $i \in [1, n]$.

P1 is specified to ensure the approximate result should be consistent with the Target Node Type that we infer in Sect. 2 based on a result r. In other words, P1 is to ensure v'_{lca} of

[3] The choice of an appropriate τ will be discussed in the experimental study.

r' should have the same node type as the TNT that result r intends to match (but fail to do so). P2 is to ensure a consistency of the internal structure of r and r' in the way that, each node type appearing in the *keyword match node* of r must also appear in those of r'. Intuitively speaking, the node type of each *keyword match node* implicitly reflects the constraint that user intends to specify for the desired query result. Therefore we need to keep all of them in the approximate result. As an analogy, it is an implicit representation of predicates specified in a structured query, whereas the difference is that in a keyword query you have no way to specify constraint on the structural relationship among keywords. Note that, if possible, m'_i and m_i should be the same node, since we prefer changing as small number of keywords as possible. That is why we find the approximate results by checking the path from v_{lca} to m_i. Only the m_i that is not in the subtree rooted at v'_{lca} will be replaced by a distinct node m'_i.

The construction of the approximate result starts from the subtree rooted at v'_{lca}, followed by checking whether this subtree satisfies the aforementioned properties.

Suggested query and sample query result After the approximate query results are found, the explanation and suggested query can be inferred easily by the following way: 1) for each different *keyword match node* m_i which is not the same node as m'_i, the query keyword(s) in m_i is the reason for the mismatch results; 2) the suggested query can be generated by replacing the keywords in m_i with the associated value of m'_i, highlighted by an underline. Besides, the approximate query result will be used as a sample query result for the corresponding suggested query.

Property 1 For a given query with MisMatch problem, each suggested query produced by our MisMatch suggester guarantees to have at least one result which does not miss the target.

Proof Given a query result $r = (v_{lca}, \{m_1, m_2, \ldots, m_n\})$ and its approximate result $r' = (v'_{lca}, \{m'_1, m'_2, \ldots, m'_n\})$, let the set of different node types for m_1 to m_n be $\mathbb{T} = \{t_1, t_2, \ldots, t_x\}$ and the set of different node types for m'_1 to m'_n be $\mathbb{T}' = \{t'_1, t'_2, \ldots, t'_x\}$, we can have $t_i = t'_i$ $(i \in [1, x])$ according to P2 in Definition 8. Then according to Definition 5, we can know $TNT(r) = TNT(r')$. Because of P1 in Definition 8, we can have $v'_{lca}.type = TNT(r')$. So r' does not miss the target. □

Next, we will use two running examples to illustrate how we find the suggested queries and sample query result. The following two running examples correspond to the queries in Examples 4 and 5, respectively.

Example 7 For query Q={'Vaio','W','red','price'} in Example 4, one query result is r=(0.0.0, {0.0.0.4.1, 0.0.0.4.1, 0.0.0.3.2, 0.0.0.4.5}), where there are only three distinct *keyword match nodes*. So we calculate three distinguishability values w.r.t. the query keywords in the three *keyword match nodes*: $D(\{\text{'}Vaio\text{'}, \text{'}W\text{'}\},\text{model})=100\,\%$,$D(\{\text{'red'}\},\text{color}) = 68.2\,\%$,$D(\{\text{'price'}\}, price) = 0.5\,\%$.

Since D({'Vaio', 'W'},model)$> \tau = 90\,\%$, it is important and must be kept. Then we check the path from $shop$:0.0.0 (v_{lca}) to $model$:0.0.0.4.1 (m_i), which is ($shop$:0.0.0/$laptop$:0.0.0.4/$model$:0.0.0.4.1). In Example 4, we know $TNT(r) = laptop$, so we check the subtree rooted at $laptop$:0.0.0.4. For each *keyword match node* m_i in the original result r, within the subtree rooted at 0.0.0.4, we can always find a node m'_i with the same type. For example, for the *keyword match node* 0.0.0.3.2 in r, we can find node 0.0.0.4.2 with the same node type: $(0.0.0.4.2).type = color = (0.0.0.3.2).type$. Thus the set of m' nodes is: {0.0.0.4.1, 0.0.0.4.1, 0.0.0.4.2, 0.0.0.4.5}. Therefore, an approximate query result r' is constructed:

$$r' = (0.0.0.4, \{0.0.0.4.1, 0.0.0.4.1, 0.0.0.4.2, 0.0.0.4.5\})$$

Compared to r, *keyword match node* $color$:0.0.0.3.2 is changed to $color$:0.0.0.4.2. Node $color$:0.0.0.3.2 contains keyword 'red' and the content of $color$:0.0.0.4.2 is 'white'. So the keyword 'red' in user's query leads to the mismatch results. The suggested query can also be inferred as {'Vaio', 'W',<u>'white'</u>, 'price'} by changing 'red' to 'white', and r' is its corresponding sample result. Similarly, we can also find suggested queries by changing 'red' to 'blue' or 'pink'.

Example 8 For query Q = {'Hewlett','Packard','Pavilion', 'Omni'} in Example 5, where the user wants to search for a shop selling both the laptop model 'Pavilion' and 'Omni'. However, there is no such shop which sells both of the laptop models. One query result is a subtree rooted at an $electronics$ node: r=(0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.1.3.1}), where there are only three distinct *keyword match nodes*. So we calculate three distinguishability values w.r.t. the query keywords in the three *keyword match nodes*: $D(\{\text{'}Hewlett\text{'}, \text{'}Packard\text{'}\}, brand) = 75.5\,\%$, $D(\{\text{'}Pavilion\text{'}\}, model) = 100\,\%$, $D(\{\text{'}Omni\text{'}\}, model) = 100\,\%$.

Since both D({'Pavilion'},model) and D({'Omni'}, model) are larger than the threshold τ (90 %), both of them are important. So we will check the following two paths for finding approximate results: path from $electronics$:0.0 (v_{lca}) to $model$:0.0.0.3.1 (m_i); path from $electronics$:0.0 (v_{lca}) to $model$:0.0.1.3.1 (m_i). Here we will take the first path as an example to illustrate how to check the path, which is ($electronics$:0.0/$shop$:0.0.0/$laptop$:0.0.0.3/$model$:0.0.0.3.1). In Example 5 we know $TNT(r) = shop$, so we check the subtree rooted at $shop$:0.0.0. For each *keyword match node* m_i in the original result r, within the subtree rooted at 0.0.0, we can always find a node m'_i with the same type. For example, for the *keyword match node* 0.0.1.3.1

in r, we can find node 0.0.0.4.1 with the same node type: $(0.0.0.4.1).type = model = (0.0.1.3.1).type$. Thus the set of m' nodes is: {0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.0.4.1}. Therefore, an approximate query result r' is constructed:

$r' = (0.0.0, \{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.1, 0.0.0.4.1\})$

Compared to r, *keyword match node model*:0.0.1.3.1 is changed to *model*:0.0.0.4.1. Node *model*:0.0.1.3.1 contains keyword 'Omni' and the content of *model*:0.0.0.4.1 is 'Vaio W'. So the keyword 'Omni' in user's query leads to the mismatch results. The suggested query can also be inferred as {'Hewlett','Packard','Pavilion','Vaio','W'} by changing 'Omni' to 'Vaio W,' because shops selling these two models are available. r' is the corresponding sample result.

Note that, if we set the threshold τ to a very low value, say zero, which means all keywords are with acceptably high distinguishability, then we will examine all the TNT nodes containing at least one of the *keyword match nodes*. This can cover all possibilities but of course more time will be consumed. We will show in the experiment (Sect. 7) that most likely it is not necessary.

3.2.2 Phase 2: based on multiple keyword match nodes

When the important keywords are derived from multiple *keyword match nodes* m_i, i.e., combing the keywords from multiple *keyword match nodes* could achieve high distinguishability, we need to compute the lowest common ancestor of these m_i, denoted by v, in order to calculate distinguishability. This is the only difference as compared to Phase 1. Let $\mathbb{K}$ be the query keywords subtree-contained by v. Then the rest job is similar to *Phase 1*, where we calculate $D(\mathbb{K}, v.type)$ and if it is acceptably high, we will check the path from v_{lca} to v to find the approximate result(s). Please refer to Algorithm 1 for details on our two-phase solution.

However, it requires 2^n times of calculation to get all possible lowest common ancestors of any subset of the *n keyword match nodes*. But we find Property 2 to help fulfill it in linear time.

Property 2 Let $M = \{m_1, m_2, \ldots, m_n\}$ be the set of distinct *keyword match nodes* for a query result ($m_i \neq m_j$ if $i \neq j$), sorted by their Dewey labels. Then all possible lowest common ancestors (LCA) for any subset S of M, where $|S| \geq 2$, are in the set $\{lca(m_1, m_2), \ldots, lca(m_i, m_{i+1}), \ldots, lca(m_{n-1}, m_n)\}$.

Proof (By Induction) Step 1: For $n = 2$, this property obviously holds. Step 2: We assume that for n=$k-1$, all LCAs of any subset of M_{k-1}=$\{m_1,m_2,\ldots,m_{k-1}\}$ are in $\{lca(m_1, m_2), lca(m_2, m_3), \ldots, lca(m_{k-2},m_{k-1})\}$. We will show that for a set of k nodes M_k=$\{m_1,m_2,\ldots,m_{k-1},m_k\}$, all possible LCAs are in the set L=$\{lca(m_1,m_2),lca(m_2,m_3),\ldots,lca(m_{k-1},m_k)\}$. Suppose $Dewey(m_{k-1})$=$a_1.a_2\ldots a_j.a_{j+1}\ldots$ and $Dewey(m_k)$=$a_1.a_2\ldots a_j.a'_{j+1}\ldots$, let m' = $lca(m_{k-1}, m_k)$, then $Dewey(m')$=$a_1.a_2\ldots a_j$. As nodes are sorted by Dewey label, there does not exist another node m_i in M_k such that $lca(m_i, m_k)$ is a descendant of m'; otherwise, $Dewey(m_i)$ should be of the form $a_1.a_2\ldots a_j.a'_{j+1}\ldots$ and m_i should appear between m_{k-1} and m_k. So for any subset containing m_k, namely $\{m'_1, m'_2, \ldots, m_k\}$, their LCA must not be a descendant of m'. If the LCA node equals to m', it is in L; if the LCA node is an ancestor of m', we can get the following because finding LCA is equal to finding the longest common prefix of Dewey labels of a set of nodes: $lca(\{m'_1, m'_2, \ldots, m_k\}) = lca(\{m'_1, m'_2, \ldots, m'\})$= $lca(\{m'_1, m'_2, \ldots, m_{k-1}\})$, which is also in L according to the assumption. Besides, for subsets not containing m_k, their LCAs will also be in L according to the assumption. □

With Property 2, for a query that has MisMatch problem, we only need to conduct at most $n-1$ times of LCA computations to find all possible approximate results. We will use Example 9 to illustrate how we infer suggested queries for Phase 2.

Example 9 Suppose a user wants to find a laptop which is of brand Hewlett Packard with purple color running windows vista. She may try to issue a query Q={'Hewlett', 'Packard', 'purple', 'windows', 'vista'} in Fig. 1. One of the query results is r=(0.0.0,{0.0.0.3.0, 0.0.0.3.0, 0.0.0.3.3, 0.0.0.4.6, 0.0.0.4.6}). By Definition 6 we know that $TNT(r) = laptop$ but the result is a subtree rooted at a *shop* node. Therefore it misses the target.

Suppose Hewlett Packard only has two models with purple color. The keywords matching *brand*:0.0.0.3.0, *color*: 0.0.0.3.3 and *OS*:0.0.0.4.6 are not of high distinguishability (90 %) in Phase 1: D({'Hewlett','Packard'},brand) = 75.5 %; D({'windows', 'vista'}, OS) = 42.5 %; D({'purple'},color) = 80.7 %. Now in phase 2, by Property 2, all possible lowest common ancestors of the *keyword match nodes* are 0.0.0.3 and 0.0.0. Take 0.0.0.3 as an example, we will find that the keywords subtree-contained by *laptop*:0.0.0.3 have high distinguishability:

D({'Hewlett', 'Packard', 'purple'},laptop)=98.4 %

Note that the above three keywords are actually from two *keyword match nodes*, i.e., *brand*:0.0.0.3.0 and *color*: 0.0.0.3.3.

Then similar to Phase 1, we will try to find an approximate query result along the path from r's v_{lca} to *laptop*:0.0.0.3, i.e. (*shop*:0.0.0/*laptop*:0.0.0.3). Finally we find the approximate result rooted at *laptop*:0.0.0.3 and get a suggested query by changing keywords 'windows vista' to 'windows 7'.

3.3 Ranking the suggested queries

After all suggested queries are generated, we build a preliminary ranking model to judge the quality $score$ of a suggested query with the following factors:

1. Number of keywords (in original query) that need to be changed is denoted as cn. The larger cn is, the lower $score$ should be.
2. Distance between the approximate query result root v'_{lca} and original query result root v_{lca} is denoted as dt (dt is equal to the length difference of their Dewey labels). The larger dt is, the higher $score$ should be, because more compact subtree is preferred.
3. Sum of distinguishability of the keywords that need to be changed is denoted as $\sum D$. The larger $\sum D$ is, the lower $score$ should be, because we prefer not to replace keywords those are with high distinguishability.

To sum up the above ranking factors, we calculate the ranking score by taking a product of them:

$$score = \frac{1}{e^{cn}} \times \left(1 - \frac{1}{e^{dt}}\right) \times \frac{1}{e^{\sum D}} \tag{2}$$

3.4 Summary of features of our approach

To summarize, our MisMatch detector and suggester have the following features. First, it is *portable*: by capturing the LCA commonality among existing search semantics in defining the format of query result (Definition 4), our approach can work with any LCA-based matching semantics (recall Sect. 2.1.2); since our approach is a post-processing of the query evaluation, it is orthogonal to the result retrieval method adopted. Second, it is *result-driven*: our approach accepts the results of the original query as input, and recall Sect. 3.2 the suggester finds the important keywords (to be kept in suggested queries) from each result, to guarantee the empirical quality of suggestions. Third, it is *lightweight*: it occupies a small proportion of the whole query evaluation time, as discussed in Sect. 4 later. Thereby, for user queries that do not have mismatch problem, the user will not be annoyed much by the extra time taken by the detector. Last, the suggestions come along with the results of original user query, so for users satisfying the current results of the original query, they can simply ignore our suggestion and browse the results of their original query.

4 Efficient MisMatch problem processing by a novel labeling scheme over XML data without ID references

Recall Definition 8, to check whether a TNT node is an approximate query result, the core operation is to verify whether the two properties **P1** and **P2** hold. Checking P1 is trivial, so we aim to achieve an efficient check of P2 by designing a novel node labeling scheme and the corresponding logical operations.

4.1 Node labeling

Since our suggester needs to frequently access the type of a node along the way to finding suggested queries, we first collect all node types in XML data. By simply scanning the XML file, we can get a schema tree which contains all node types using DataGuide [11]. E.g., for the XML data in Fig. 2, we can construct a schema tree as shown in Fig. 3a, where each node in the schema tree represents a unique node type. Note that each node in Fig. 3 should be a node type represented as a path (according to Definition 1), but for simplicity we use a tag name instead because there is no ambiguity.

Then, we use a *bitmap* to denote all node types in the schema tree, where each bit in the bitmap corresponds to a specific type. We purposefully decide which bit corresponds to which type as follows:

- Flatten the schema tree level by level in a top-down manner. Suppose a node n has k children, then n will be inserted into a place between its $\lfloor \frac{k}{2} \rfloor$th and $(\lfloor \frac{k}{2} \rfloor + 1)$th children. As a result, n will maintain its position between its neighbors and neighbors' children. Figure 3a–c show such a process of flattening.
- Construct a virtual $bitmap$ as shown in Fig. 3d. Each distinct node type has a unique position number in $bitmap$. E.g., F's position number is 3.

Such a bit-to-type mapping has a nice property: *the bits of all node types that appear in a specific subtree in XML will stay together*. As we can see later, this property helps ensure the label size as compact as possible.

For a node n in the XML tree, the subtree rooted at n may contain different types of nodes. To indicate which node

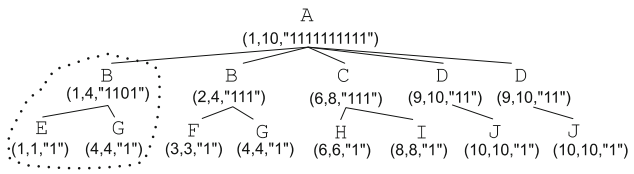

Fig. 2 An XML tree with nodes labeled by exLabels

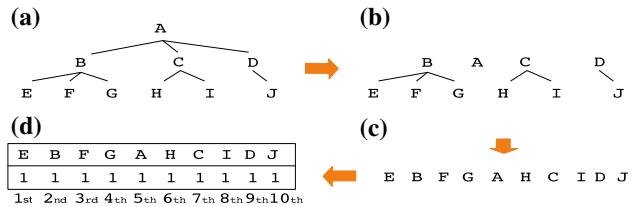

Fig. 3 Schema tree flattening and virtual bitmap construction

types appear in its subtree ST_n, we assign n a label (a, b, bm), called exLabel. Here, a is the smallest position number (in the $bitmap$) of the node type appearing within ST_n; similarly, b is the largest position number of the node type appearing within ST_n. bm is a sub-sequence of the $bitmap$ (of the schema tree) from position a to b, indicating which type of nodes can be found in the subtree rooted at n. In particular,

- $bm[i]$=1, if the node type at position $a + i - 1$ in $bitmap$ appears in the subtree rooted at n;
- $bm[i]$=0, otherwise. (i∈[1,b-a+1])

Example 10 In Fig. 2, for the subtree rooted at node B circled by the dotted line, it contains nodes of types E, B and G. According to the $bitmap$ in Fig. 3d, the position number is 1 for E, 2 for B and 4 for G. Among the four node types ranging from position 1 to 4, bm of node B indicates which of those four node types appear in B's subtree ST_B. As a result, bm=1101 as the 3rd node type F does not appear in ST_B, and B's exLabel = (1,4,'1101'). Note that the exLabel of B is *compact* because the bits representing E, B and G are staying together, which is the benefit from the aforementioned bit-to-type mapping.

4.2 Logical operation

Similar to node labeling, for a query result $r = (v_{lca}, \{m_1, m_2, \ldots, m_n\})$, we can intentionally construct an exLabel to represent its node type information even though it is not a node at all. Let a' (b') be the smallest (largest) position number of the node type for m_i, and the label for the query result is denoted as (a', b', bm').

Having a query result label (a', b', bm') and a subtree root label (a, b, bm), we can verify property **P2** by examining the following containment relation: $(a', b', bm') \subseteq (a, b, bm)$. This relationship holds only if $a \leq a' \leq b' \leq b$ and all bits that appear in bm' also appear in bm. This can be efficiently done by a logical *AND* operation on bm' and bm.

Example 11 In Fig. 2, suppose a query result $r = (v_{lca}, \{m_1, m_2\})$, where $m_1.type = B$, $m_2.type = G$. Then the exLabel for r is (2,4,'101'). If we want to check whether an approximate query result exists in the subtree rooted at the left node B in Fig. 2, whose exLabel is (1,4,'1101'), then we know the approximate query result exists because (2,4,'101') ⊆ (1,4,'1101').

5 Index construction and algorithms for MisMatch solution over XML data without ID references

5.1 Data processing and index construction

In the phase of XML document parsing, we collect all distinct node types and generate a bitmap code for each node type as discussed in Sect. 4.1. For each node n visited, we assign a Dewey label $deweyID$ [37] to n; get the node type t_n of n; construct an $exLabel$ for n. To speed up the query processing and refinement, three indexes are built.

The *first* index is called *replacement table*, which is a B+ tree storing each node with (t,$deweyID$) as its key. Such an index has the following property: by scanning rightwards of the position (t,$deweyID$), we can find all the nodes of type t under the subtree rooted at $deweyID$. Recall in Sect. 3.2, after we find an approximate query result r', we need to materialize the replacement nodes within r' in order to infer the suggested query. Since we know the type t of each replacement node and the $deweyID$ of the root node of r', with replacement table, we can easily materialize all such nodes by calling getReplacement (t,$deweyID$). The *second* index is to maintain the exLabel and type info for each node.

To speed up the computation of distinguishability, particularly for parameter $f_t^{\mathbb{K}}$ in Formula 1, the *third* index called *inverted index* is built: For each combination of a distinct node type t and a distinct keyword k (in XML data), we build an inverted list containing all nodes of type t where each node *subtree-contains* keyword k. Those inverted lists are grouped by node type t. As a result, $f_t^{\mathbb{K}}$ can be computed by simply computing the intersection of the inverted lists for each keyword in $\mathbb{K}$ under node type t [23]. Operation getDist($deweyID$,$\mathbb{K}$) returns the distinguishability of a set of keywords $\mathbb{K}$ w.r.t. the type of the node with $deweyID$.

5.2 Algorithms

The main procedure is presented in Algorithm 1, where the input is the query Q and its retrieved results $\mathbb{R}$. For *Detector*, it checks each result of Q (line 3) and calculates its TNT (line 4). Once one of the results does not miss the target, which means what the user wants is in the retrieved results, it will terminate the process (line 5). Otherwise, it constructs an exLabel for the query result (line 8) as discussed in Sect. 4.2.

For *Suggester*, in *Phase 1* as discussed in Sect. 3.2.1, it checks each *keyword match node nd* of the query result (line 10). If the distinguishability is larger than the threshold τ (line 11), the TNT node on the path from the v_{lca} to this node will be checked in order to find an approximate query result (line 12). Whether an approximate query result exists can be easily checked by examining the containment relationship between the exLabels (line 13), where function contain() will be shown in Algorithm 3. If an approximate query result exists, the explanations and suggested queries will be inferred by calling $QuerySuggester()$ (line 14).

For *Phase 2* as discussed in Sect. 3.2.2, we sort the *keyword match nodes* (line 16) and check the LCA node of every two adjacent *keyword match nodes* (line 18) according to Property 2. Then we need to find which query keywords are *subtree-contained* by this LCA node (line 19). Afterward, we follow

Algorithm 1: MisMatchResolver(Q, $\mathbb{R}$)

```
1  suggestedQueries ← ∅;
2  {Detector}
3  foreach r ∈ ℝ do
4      if r.v_lca.type = getTNT(r) then
5          return null;
6  {Suggester}
7  foreach r ∈ ℝ do
8      rExlabel = constructExlabel(r);
9      {Phase 1}
10     foreach nd ∈ r.matchnodes do
11         if getDist(nd.dewey, nd.keywords)> τ then
12             foreach n ∈ nodes on the path from r.v_lca to nd AND
               n.type = getTNT(r) do
13                 if contain(getExLabel(n.dewey), rExlabel) then
14                     QuerySuggester(n, r, suggestedQueries);
15     {Phase 2}
16     sort(r.matchnodes);
17     for i = 1 to (r.matchnodes.length-1) do
18         Let v = getLCA(r.matchnodes[i], r.matchnodes[i + 1]);
19         kwinside = getQueryKwsInside(r, v);
20         if getDist(v, kwinside)> τ then
21             foreach n ∈ nodes on the path from r.v_lca to v AND
               n.type = getTNT(r) do
22                 if contain(getExLabel(n.dewey), rExlabel) then
23                     QuerySuggester(n, r, suggestedQueries);
24 return suggestedQueries.sort();
```

Algorithm 2: QuerySuggester(v'_{lca}, r, $sugQueries$)

```
   input  : the approximate result root v'_lca, the query result being
            changed r and the suggested queries sugQueries
   output: new suggested queries + one sample result v'_lca
1  i = 0;
2  foreach nd ∈ r.matchnodes do
3      if nd is not a descendant of v'_lca then
4          replace[i++] = getReplacement(nd.type, v'_lca.dewey);
5  foreach n_1 ∈ replace[1], ..., n_i ∈ replace[i] do
6      sugQueries = sugQueries ⋃(r.matchnodes[1] →
       n_1, ..., r.matchnodes[i] → n_i);
```

the same steps (line 20-23) as Phase 1. Finally, it returns the suggested queries (line 24) sorted by the ranking formula in Sect. 3.3, attached with one sample result for each suggested query.

Given the approximate result root and the original query result, Algorithm 2 presents how to infer the suggested queries. *Keyword match nodes* which are not in the subtree rooted at v'_{lca} will be replaced by nodes in v'_{lca} that have the same node type according to property P2 in Definition 8 (line 2–4). For a *keyword match node* that needs to be changed, there may be more than one replacement node to replace it. Such nodes can be retrieved from index by calling the function $getReplacement()$ (line 4). Note that there might be more than one *keyword match node* needed to be changed, so suggested queries will be inferred by considering all possible cases (line 6).

Algorithm 3: contain(elx, ely)

```
   input  : exLabel elx and exLabel ely
   output: a boolean indicating whether elx contains ely
1  if (elx.a ⩽ ely.a and ely.b ⩽ elx.b)==false then
2      return false;
3  bmTemp = subset of elx.bm from position ely.a to ely.b;
4  if (bmTemp & ely.bm)==ely.bm then
5      return true;
6  return false;
```

Algorithm 3 presents the function $contain()$ to examine the containment relationship between two exLabels, i.e., the first contains the second. As discussed in Sect. 4.2, one condition for the relationship to be held is that the range of the second label should be contained by the first (line 1-2). After that, we need to make sure every bit that appears in the second label also appears in the first. Since the bitmap length of the two may not be the same, we shrink the first bitmap as the same length as the second (line 3). Then bit checking can be done by only doing a logical AND operation on two bitmaps (line 4). Then a boolean result indicating the containment relationship will be returned accordingly (line 5 and line 6).

6 Resolving the MisMatch problem over XML data with ID references

In the previous sections, we have discussed how to detect and resolve the mismatch problem over the XML data without ID references. In a more general scenario, XML data may contain ID nodes and IDREF nodes to represent the reference relationships among the data. E.g., Fig. 4 is an XML document with ID references describing an online shopping mall, where the containment edges and reference edges are presented by solid line (→) and dashed line (⇢) respectively. Each shop sells some laptops. Each laptop node can have some IDREF nodes, i.e., laptopRef nodes, pointing to the laptop information, which can be reused by different shops to avoid duplication. An XML document with ID references is usually modeled as a digraph rather than a tree, which we will call *XML IDREF digraph* in this paper. Both search semantics and search methods on the digraph model differ from that of the tree model, which brings more challenges to extend the detector and suggester to be able to work on such a general scenario.

A literature study shows that the problem of keyword search on an XML IDREF digraph can be reduced to the problem of finding minimal Steiner tree (MST) or its variants in a digraph [8,14,16,20,40]: given a digraph $G = (V, E)$, where V is a set of nodes and E is a set of edges, a keyword query result is defined as a minimal directed subtree T in G such that the leaves or the root of T contain all

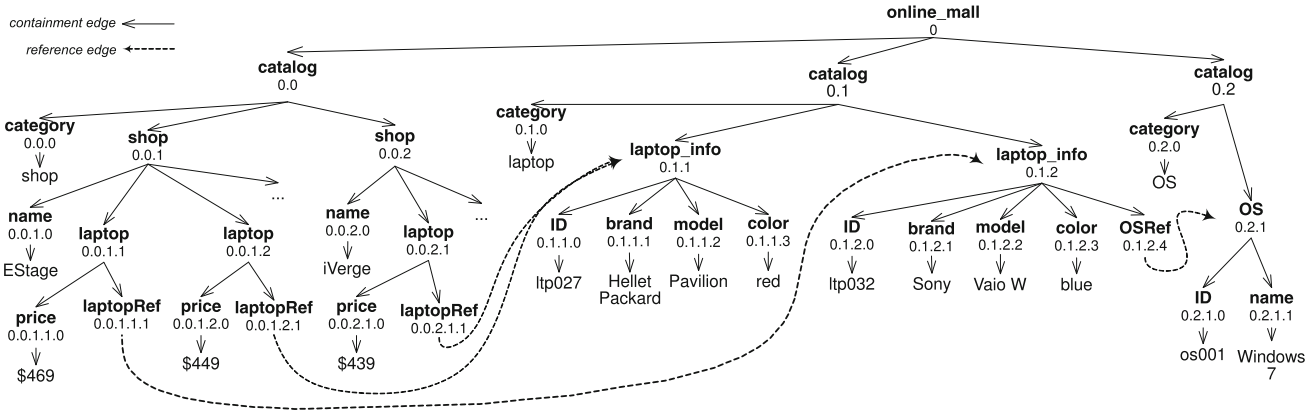

Fig. 4 Sample XML document with ID references

keywords in the query. Although the matching semantics on XML IDREF digraph is different from the LCA-based semantics on XML tree, MisMatch problem could still exist. This is because MisMatch problem exists in any form of information retrieval over data of any structure, as discussed in Sect. 1.

Example 12 Suppose a user wants to find the price of a laptop with model being Vaio W and red color, she may issue a query $Q = \{`Vaio', `W', `red', `price'\}$ over the data in Fig. 4. But red color is unavailable for model Vaio W. Therefore, what will be returned is a list of mismatch results. One of the results by minimal Steiner tree (MST) [9] is a tree rooted at *shop*:0.0.1, with three *keyword match nodes*:

0.1.2.2for keyword'Vaio′and'W′,
(following a path 0.0.1 → 0.0.1.1 → 0.0.1.1.1 ⇢ 0.1.2 → 0.1.2.2)0.1.1.3for keyword 'red′,
(following a path 0.0.1 → 0.0.1.2 → 0.0.1.2.1 ⇢ 0.1.1 → 0.1.1.3)
0.0.1.1.0for keyword 'price′.
(following a path 0.0.1 → 0.0.1.1 → 0.0.1.1.0)

As we can see, shop is returned because there is no laptop that can meet all the requirements. So MisMatch problem exists, and mismatch results are returned.

The results being returned (minimal Steiner tree) for XML IDREF digraph are also tree-structured. As our MisMatch solution is a post-processing job working on tree-structured results and orthogonal to the result retrieval method, such an observation offers a good opportunity for us to extend our post-processing MisMatch solution onto XML IDREF digraph.

In this section, we will discuss the necessary extension we need to make to extend our MisMatch solution onto XML IDREF digraph. Recall Sects. 2, 3 and 4, the solution to the MisMatch problem is based on two novel concepts we proposed: *Target Node Type* (TNT) and *distinguishability*. The framework includes three main steps: 1) Detect the MisMatch problem by calculating the Target Node Type; 2) Measure the keywords importance based on the distinguishability; 3) Efficiently discover approximate results, from which the suggested queries can be inferred.

Next we will talk about how to extend our MisMatch solution to the context of XML IDREF digraph, resulting in a general framework.

6.1 Target node type for detecting the MisMatch problem

Given a result, the central idea to calculate the Target Node Type is getting the node type of each *keyword match node* and count their occurrences. But in an XML IDREF digraph, there could be more than one paths from one node to another. Therefore, there could have more than one node types for a given node. This is different from XML tree. E.g., for the node $laptop\_info$:0.1.1 in Fig. 4, there are two possible paths from the document root node to it. One is through ID reference, and the other one is not through ID reference. So there are two different node types for that node, i.e., node type "online_mall/ catalog/laptop_info" and "online_mall/catalog/ shop/laptop/laptopRef/laptop_info". Therefore, given a query result, we need to know which path it goes through from the answer root to each *keyword match node*.

First of all, we will define the format of search result for XML IDREF digraph, which is slightly different from the format for XML tree in Definition 4:

Definition 9 (*Query Result Format for XML IDREF Digraph*) For a keyword query $Q=\{k_1, \ldots, k_n\}$ issued on the XML data with ID references, we define the format of a query result r as:

$r = \left(v_{MSTRoot}, \left\{m_1(path_1), m_2(path_2), \ldots, m_n(path_n)\right\}\right)$

where m_i is a *keyword match node* w.r.t. keyword k_i ($i \in [1, n]$); $v_{MSTRoot}$ is the root node of the minimal Steiner tree connecting m_1 to m_n; $path_i$ is the path from $v_{MSTRoot}$ to m_i.

Comparing to Definition 4, the only difference is that we need to specify the path from the result root to each *keyword match node*, as there could be more than one path from one node to another in an XML IDREF digraph. For example, in Fig. 4, there is multiple paths from node $online\_mall$:0 to node $laptop\_info$:0.1.1, either through the reference edges or not through the reference edges.

Secondly, to calculate the Target Node Type (TNT) of a result r, we need to get the node type of each *keyword match node* and count their occurrences. Given a result $r = (v_{MSTRoot}, \{m_1(path_1), m_2(path_2), \ldots, m_n(path_n)\})$, the node type of m_i consists of two parts: (1) path from document root to $v_{MSTRoot}$; (2) path from $v_{MSTRoot}$ to m_i, i.e. $path_i$. We can just combine these two parts to get the node type of m_i.

After the node type for each *keyword match node* is ready, we can now calculate the TNT of a given result and detect the MisMatch problem in the same way in Sect. 2.

Example 13 For a query $Q = \{$'$Vaio$','W','red','$price$'$\}$ issued in Fig. 4, one of the results is

$r = (0.0.1, \{0.1.2.2(0.0.1 \rightarrow 0.0.1.1 \rightarrow 0.0.1.1.1$
$\dashrightarrow 0.1.2 \rightarrow 0.1.2.2),$
$0.1.2.2(0.0.1 \rightarrow 0.0.1.1 \rightarrow 0.0.1.1.1 \dashrightarrow 0.1.2 \rightarrow 0.1.2.2),$
$0.1.1.3(0.0.1 \rightarrow 0.0.1.2 \rightarrow 0.0.1.2.1 \dashrightarrow 0.1.1 \rightarrow 0.1.1.3),$
$0.0.1.1.0(0.0.1 \rightarrow 0.0.1.1 \rightarrow 0.0.1.1.0)\}).$

The node types of these *keyword match nodes* are ('Vaio' and 'W' match the same node):

0.1.2.2 : $\{online\_mall/catalog/shop/laptop/laptopRef/laptop\_info/model\}$(denoted as t_1)
0.1.1.3 : $\{online\_mall/catalog/shop/laptop/laptopRef/laptop\_info/color\}$(denoted as t_2)
0.0.1.1.0 : $\{online\_mall/catalog/shop/laptop/price\}$ (denoted a st_3).

The set of distinct node types $\mathbb{T} = \{t_1, t_2, t_3\}$, where $count(t_1) = 1$, $count(t_2) = 1$ and $count(t_3) = 1$.

Then we check the prefixes of all node types in $\mathbb{T}$. The lowest one is $t =$ "$online\_mall/catalog/shop/laptop$". Suppose we have the following constraints (either by examining the XML schema or scanning the XML document): $t.maxContain(t_1) = 1 \geq count(t_1) = 1$, $t.maxContain(t_2) = 1 \geq count(t_2)$ and $t.maxContain(t_3) = 1 \geq count(t_3)$.

Therefore, by Definition 5, $TNT(r) = t =$ "$online\_mall/catalog/shop/laptop$" even though no laptop can meet all the user's requirements at data level.

6.2 Distinguishability for measuring keywords' importance

Recall Sect. 2 that we have proposed the concept of *distinguishability* to measure the importance of the query keywords contained in a certain *keyword match node* in XML tree model. However, in an XML IDREF digraph, the reference edges can make the structure a bit complex. There could be sequential references (a node a references a node b, and then a descendent of b also references a third node c) and cyclic references (containment edges and references edge form a cycle). Then there can be exponentially many node types. This poses challenges for directly adopting the distinguishability formula (i.e., Eq. 1). But we also notice that many node types in an XML IDREF digraph are actually representing the same type of information. Therefore, in computing the distinguishability in XML digraph, we propose to exploit the node types to replace each other when they are representing the same type of information.

Recall Eq. 1, distinguishability $D(\mathbb{K}, t)$ measures the importance of the query keywords $\mathbb{K}$ when $\mathbb{K}$ match a node of type t. It is defined based on two variables f_t and $f_t^{\mathbb{K}}$. f_t is the number of nodes of type t; $f_t^{\mathbb{K}}$ is the number of nodes which are of node type t and subtree-contain each keyword in $\mathbb{K}$.

As discussed in Sect. 5, we store a f_t value for each distinct node type t; to calculate $f_t^{\mathbb{K}}$, we build the following inverted index: for each combination of a distinct node type t and a distinct keyword k (in the XML data), we build an inverted list L_t^k, where each node in the list is of type t and subtree-contains keyword k. As a result, $f_t^{\mathbb{K}}$ can be computed by simply computing the intersection of the inverted lists L_t^k for each $k \in \mathbb{K}$.

Here for the XML IDREF digraph, it will be good if we could build the same index. However, there could be exponentially many node types in the XML IDREF digraph. So it is not feasible to store a f_t value for each distinct node type t and build an inverted index for each combination of a distinct node type t and a distinct keyword k.

Comparing the node types in an XML tree to those in an XML IDREF digraph, we notice that many node types in an XML IDREF digraph are actually representing the same type of information. For example in Fig. 4, the node type $t_a =$ "online_mall/catalog/laptop_info" and $t_b =$"online_mall/catalog/shop/laptop/laptopRef/laptop_info" are actually representing the same type of information, i.e., laptop_info. If we extract the schema graph of the XML document, as shown

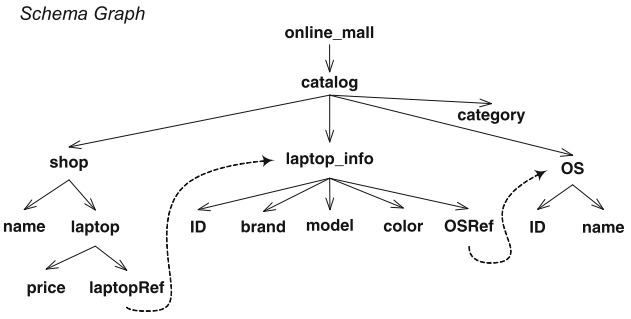

Fig. 5 Schema graph of Fig. 4

in Fig. 5, it will be clearer that these two node types actually represent the same type of information, i.e., they correspond to the same schema node in the schema graph. Node type t_a contains no ID reference edge in its path while t_b contains ID reference edge in its path.

Definition 10 (*Solid and Virtual Node Type*) We call a node type which does not contain ID reference edges in its path as solid node type; a node type which contains ID reference edges in its path as virtual node type.

Since *solid node types* do not include ID reference edges, the number of *solid node types* equals to the number of schema nodes in the schema graph. For example in Fig. 5, the number of *solid node types* is 17 as there are 17 schema nodes in the schema graph, while the number of *virtual node types* can be exponentially many. But it is easy to know from the schema graph that, every *virtual node type* corresponds to a *solid node type*, i.e., they correspond to the same schema node in the schema graph.

Therefore, to calculate the distinguishability, a feasible solution is to use the distinguishability for a *solid node type* to simulate the distinguishability for a *virtual node type*, if they are representing the same type of information. Let $solid(t)$ be the corresponding *solid node type* for a *virtual node type* t, then we can define the distinguishability as follows:

$$D(\mathbb{K}, t) = \begin{cases} 1 - \frac{f_t^{\mathbb{K}}}{f_t} + \frac{1}{f_t}, & \text{if } t \text{ is a } solid\ node\ type. \\ D(\mathbb{K}, solid(t)), & \text{if } t \text{ is a } virtual\ node\ type. \end{cases} \quad (3)$$

If a node type t is a *solid node type*, we define it the same way as Eq. 1; if a node type t is a *virtual node type*, we use the distinguishability for $solid(t)$ to simulate its distinguishability value.

So now we can store a f_t value for each distinct *solid node type* and build an inverted index for each combination of a distinct *solid node type* t and a distinct keyword k. Then distinguishability can calculated based on such indexes in the same way as discussed in Sect. 5.

6.3 exLabel for efficient approximate results detection

When we adopt the exLabel for the XML IDREF digraph, we encounter the same problem as we do in calculating the distinguishability: there could be exponentially many node types in an XML IDREF digraph (including *solid node types* and *virtual node types*). The exLabel is bitmap-based: every node n's exLabel records what types of nodes are subtree-contained by n; each bit of the exLabel corresponds to a particular node type and the value of the bit indicates whether some nodes of this type are subtree-contained by n.

In an XML IDREF digraph, there could be exponentially many node types. Every node n in the data could have exponentially many node types appearing in the nodes subtree-contained by n. Therefore, it is not a feasible solution to record all node types in n's exLabel. We adopt a similar solution as discussed in Sect. 6.2: using a *solid node type* to replace a *virtual node type* in an exLabel if they are representing the same type of information.

Actually the purpose of exLabel is: we want to check the exLabel of a node n to see whether n can be an approximate result. If the exLabel shows that node n subtree-contains the replacement for a particular type of node, then it could be an approximate result. In XML IDREF digraph, every *virtual node type* corresponds to a *solid node type* representing the same type of information, e.g., "online_mall/catalog/laptop_info" and "online_mall/catalog/shop/laptop/laptopRef/laptop_info" in Fig. 4 represent the same type of information. Also, there are relatively small number of *solid node types* in an XML IDREF digraph.

So a feasible solution is to use the *solid node types* to represent the *virtual node types* in the exLabel (which is actually a bitmap). For example, if a node n subtree-contains some nodes of *virtual node type* t, then we can set the bit for node type $solid(t)$ in the exLabel because $solid(t)$ represents the same type of information as t. As a result, the maximum size of an exLabel (number of bits) equals to the number of distinct *solid node types* in the XML IDREF digraph. Then the approximate results can be efficiently found based on the exLabel, as discussed in Sect.5. Each approximate result will subtree-contain the replacement nodes for the *keyword match nodes* containing the keywords of less importance, where such replacement nodes and the *keyword match nodes* represent the same type of information. So the suggested queries can be inferred by replacing those less important keywords with the keywords in the replacement nodes. Note that to make sure that each suggested query does not have MisMatch problem anymore, we need to check whether the TNT of the approximate result is same as its root's node type before we return the suggested query to users.

Table 1 10 of the sample queries on IMDB

IMDB			
#	Query	Suggested queries	Best-3 suggested queries (format: explanation → suggested options)
Q_1	Gladiator Spanish	5	(language): Spanish → English/Japanese/French
Q_2	Spielberg DiCaprio action movie	6	(genres): Action → Biography/Crime/Drama
Q_3	Neo hacker phonebooth	3061	(keyword): Phonebooth → computer/software/programmer
Q_4	Warner Bros. movie	0	None
Q_5	Italy Betty Fisher	12	(country): Italy → France/Canada/USA
Q_6	Spielberg Schwarzenegger	58	(name): Schwarzenegger→ Meredith Brooks/Jim Conroy/Dean Spunt
Q_7	Terminator 3 cast Sarah	19	(name): Sarah → Nick Stahl/Claire Danes/Kristanna Loken
Q_8	Panic Room 2001	11	(year): 2001 → 2002 (title): Panic Room → Promised Land / Nowhere Road
Q_9	Ettore The Man movie	1189	(director): Ettore → Ethan Coen/Salvatore Maira/Massimo Sani
Q_{10}	Boy death ghost love	992	(keyword): Love → orphanage/bully/bomb

7 Experiments

We have conducted extensive experiments to verify the effectiveness, efficiency and scalability of our approach. For expository convenience, we refer to our MisMatch Detector & Suggester as *MisMatch Module* and refer to our extended MisMatch solution (which works on XML data with ID references) as *Extended MisMatch Module*.

7.1 Experimental settings

All experiments are conducted on a 2.83GHz Core 2 Quad machine with 3GB RAM running 32-bit windows 7. All codes are implemented in Java. Berkeley DB Java Edition [1] is used to store all indexes for our algorithms.

Data set

- Three real datasets are tested for our *MisMatch Module*: (1)IMDB[4] 90MB, where around 200,000 movies of recent years are selected in our dataset. Each movie contains information like title, rating, director, etc. (2) DBLP 520MB, which contains publications since 1990. (3) IEEE Publication 90MB from INEX[5].
- One real dataset with ID references, ACMDL[6] 45MB, is tested for the *Extended MisMatch Module*. The dataset we use contains 38K publications and 253K citation (as ID references) among the publications.

Query set Our query set contains 18 queries for each of the four datasets, all of which are collected from the real-world user log data of our system. 10 sample queries for IMDB and ACMDL with their best-3 suggested queries (if any) are shown in Tables 1 and 2. Besides, 1000 random queries are generated for each dataset as well (see Sect. 7.5.2), where the max (average) number of results is 2691 (169).

Ground truth For each dataset, we employ 15 assessors to pick up the queries with the MisMatch problem, and their judgements are based on both the queries given and their respective results. We obtained the ground truth by judging a query to have the MisMatch problem if at least 8 of the 15 assessors agree on that. Eventually, 9 (10, 10, 9) out of the 18 queries for IMDB (DBLP, IEEE, ACMDL) have the MisMatch problem.

Keyword Search Method For XML data without ID references, we choose SLCA [38], which is one of the most efficient ones so far. Since no SLCA-based search method proposed so far has result ranking component, for the experiment we adopt the result ranking scheme of XRank [12].

For XML data with ID references, we choose BLINKS [14], which is one of the most efficient graph search method by building a bi-level index.

7.2 Evaluating the MisMatch problem

7.2.1 Frequency of the MisMatch problem

We have done a survey among 15 human assessors. Each assessor is required to issue 30 queries in XClear [41], an XML keyword search engine, to find some movies they want

[4] http://www.imdb.com/interfaces.

[5] https://inex.mmci.uni-saarland.de/.

[6] Thanks to Craig Rodkin at ACM Headquarters for providing the ACM Digital Library dataset.

Table 2 10 of the sample queries on ACMDL

ACMDL (with ID references)			
#	Query	Suggested queries	Best-3 suggested queries (format: explanation → suggested options)
Q_{A1}	Jeffrey Ullman INFOCOMM	163	(proceeding): INFOCOMM → PODS/SIGMOD/KDD
Q_{A2}	Ling Tok Wang KDD 1993	87	(year): 1993 → 2000
			(proceeding): KDD → SAC/ACM transaction on database systems
Q_{A3}	Michael Stonebraker PODS	255	(proceeding): PODS → CHI/OOPSLA/SIGMOD
Q_{A4}	Victor Vianu PODS 1999	81	(year): 1999 → 2000/1998/1997
Q_{A5}	Hanspeter Pfister database	671	(title): database→ integrated volume compression and visualization /
			The VolumePro real-time ray-casting system /
			VolVis a diversified volume visualization system
Q_{A6}	Michael Franklin 2000	0	None
Q_{A7}	Tan Kian-lee robot	229	(title): robot → A framework for modeling buffer replacement strategies /
			Sampling from databases using B+-trees /
			Rule-assisted prefetching in Web-server caching
Q_{A8}	SIGIR England 1985	992	(country): England → Canada
			(year): 1985 → 1984 / 1980
Q_{A9}	David Dewitt skyline	34	(title): skyline → The 007 Benchmark /
			A status report on the OO7 OODBMS benchmarking effort /
			Crash recovery in client-server EXODUS
Q_{A10}	Jagadish SIGMOD graph	2474	(title): graph → incorporating hierarchy in a relational model of data /
			A retrieval technique for similar shapes/
			Linear clustering of objects with multiple attributes

to watch in the IMDB dataset. Each assessor is asked to judge whether her queries have the MisMatch problem according to the query results. The same experiments are also conducted on DBLP, IEEE and ACMDL datasets. We find that averagely users suffered from such a problem for 27 % of their queries.

7.2.2 Sensitivity of the MisMatch detector

With the ground truth obtained, we study the precision and recall of our MisMatch detector. Let A be the set of queries that do have MisMatch problem. Let B be the set of queries that our detector claims to have MisMatch problem. Then the precision=$|A \cap B|/|B|$, while recall=$|A \cap B|/|A|$. The result for queries on each dataset is shown in Table 3. We find:

(1) Our detector achieves a perfect recall, i.e., we do not miss any query that does have MisMatch problem. This is because the detector checks *all* the results of Q before deciding whether Q has MisMatch problem (by Definition 7).

Table 3 Sensitivity of the MisMatch detector

Data set	Precision (%)	Recall (%)
Without ID reference		
IMDB	90	100
DBLP	91	100
IEEE	100	100
With ID references		
ACMDL	90	100

(2) A non-perfect precision tells that we may accidentally identify some queries without MisMatch problem as problematic. For example, for a query 'Joel Ethan' issued on IMDB, no person in database has such a name. For such a query, it is ambiguous that whether the user intends to find a movie related to two persons, or to find a person with that name which does not exist. In this case, our approach infers *movie* as the TNT, but some users may think it is to find one person but with the name wrongly input. Note that in fact no existing approach can solve the ambiguous query thoroughly [2].

(3) For the datasets with and without ID references, the recall and precision are almost the same. This is because our detector is working on the tree-structured query results regardless of the fact whether the results are returned by SLCA or BLINKS.

Further, we evaluated the sensitivity of our detector in a more general setting of MisMatch problem (Definition 7), in order to simulate various degrees of aggressiveness by different users, where we decide a query to have mismatch problem when its top-K results miss the target (in Sect. 2.3). In particular, we vary K from 1 to 10. The results are same as Table 3 in terms of precision and recall. The reason is as below. Since the input of our detector is the top-K results returned by XRank [12], whose ranking scheme gives higher scores to more compact results, and those non-mismatched results are more compact than the mismatched results, the non-mismatched results will be prioritized in the ranked list. Therefore, there are two cases to consider. Case 1: if a query Q has at least one result that does not miss the target (by Definition 6), the top-1 result will always not miss the target, so by Definition 7 Q will not have mismatch problem. Case 2: if each result of a query Q misses the target, then we can certainly derive that each of its top-K results misses the target as well (because top-K results is a subset of all results).

7.3 Effectiveness

We first have a glance at how explanations and suggestions look like for real-world queries in Table 1. For Q_8, 'Panic Room' ('2001') is associated with the node of type *title* (*year*), but no single movie contains all keywords. Naturally, one suggestion is to find a movie with the same title but different year (e.g., '2001'→'2002'), or to find a movie with the same year but different title (e.g., 'Panic Room'→'Promised Land'). Note that we do not replace the keyword(s) directly, instead we first replace the *keyword match node*, then derive the keywords as replacement. The term inside the parenthesis in Table 1 indicates *the type of the node* in which the replacement is involved. The left hand side of the arrow is the keyword(s) which lead to the mismatch problem (explanation part). Q_3 has 3061 suggestions, because Q_3 has a large number of results, and our suggester works by checking each result to generate suggestions (if any).

For ACMDL dataset (with ID reference), some of the suggested queries are found involving ID references while some of them are found without involving ID references. For example, for Q_{A1}, according to the dataset, Jeffrey Ullman did not publish any paper at INFOCOMM or reference any INFOCOMM paper in his paper. The results being returned are all mismatch results. The suggested options PODS and SIGMOD are found without involving ID references, which are the conferences Jeffrey Ullman has published papers at. Another suggested option KDD is found involving ID references. KDD is suggested because some KDD papers are referenced in Jeffrey's papers and such paper reference relationship is expressed by XML reference edges. However, such information is not reflected simply by the suggested queries. This is why our MisMatch Module also returns a sample result for each suggested query to help users understand our suggestion.

7.3.1 Evaluation method

We select the queries with the MisMatch problem for each dataset to conduct a user study.

To conduct a fair evaluation, we are aware of two things. *First*, we invite both experts and novices to participate the task of scoring the suggested query. For DBLP, IEEE and ACMDL, we ask three CS research students and three undergraduates in other faculties; for IMDB, we ask three movie fans and three non-fans. The assessors are shown the matching results of each query, the best-5 suggested queries together with the corresponding sample query results. *Second*, the assessors are asked to score the quality of each suggested query by using the Cumulated Gain-based evaluation (CG) metric [18] (from 0 to 5 points, 5 means best while 0 means worst). In contrast to traditional metrics like precision and recall which adopt a binary judgement (yes or no), CG is aware of the fact that all results are not of equal relevance to user.

7.3.2 Evaluation of overall quality

The average scores for best-3 and best-5 suggestions are shown in Fig. 6 [7]. We can find for queries with the MisMatch problem, our approach is able to find reasonable suggested queries for them, and subsequently it leads to more meaningful results; the scores for best-3 suggestions are always higher than those of best-5, which also shows the effect of our query ranking scheme.

Although our suggested queries can lead to better query results, some are still given low scores by some assessors because new keywords and old keywords are not semantically similar, such as the replacement for Q_{10} in Table 1. But considering lexical semantics is out of the scope of this paper.

Most likely, the best-3 suggested queries will be viewed by the struggling users. So in the rest of the paper, when we talk about the quality of the suggested queries, we mean the average score of the best-3 suggested queries.

[7] Here by default we adopt $\tau = 0.9$. Experiment on effects of threshold setting is discussed in Sect. 7.3.4.

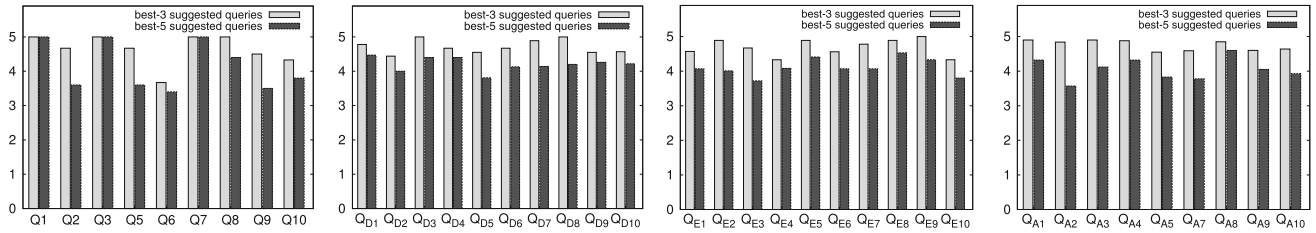

Fig. 6 Average quality measure of suggested queries (for the testing queries with MisMatch problem). **a** IMDB. **b** DBLP. **c** IEEE. **d** ACMDL (with ID references)

Table 4 Suggestion quality w.r.t. different τ and ranking factors

τ	All ranking factors	No *cn*	No *dt*	No $\sum D$
IMDB				
0.9	4.63	4.30	4.37	4.13
0.6	4.63	4.30	4.37	4.13
0.3	4.63	4.30	4.37	4.13
0.0	4.63	4.30	4.37	4.13
DBLP				
0.9	4.71	4.39	4.39	4.13
0.6	4.71	4.36	4.42	4.18
0.3	4.71	4.36	4.42	4.18
0.0	4.71	4.36	4.42	4.18
IEEE				
0.9	4.68	4.34	4.41	4.18
0.6	4.68	4.34	4.42	4.19
0.3	4.68	4.34	4.42	4.19
0.0	4.68	4.34	4.42	4.19
ACMDL				
0.9	4.61	4.32	4.35	4.10
0.6	4.61	4.32	4.35	4.10
0.3	4.61	4.32	4.35	4.10
0.0	4.61	4.32	4.35	4.10

7.3.3 Study of the query ranking scheme

We further study how the proposed ranking factors for ranking suggested query affect the overall quality of suggested queries. The ranking factors include *cn*, *dt* and $\sum D$, as discussed in Sect. 3.3. The scores for the suggested queries of each case are shown in Table 4. Please ignore the choice of τ for the time being. By comparing the scores in a columnwise way, we find:

(1) The model taking all ranking factors always outperforms any models that miss one of the three ranking factors.
(2) Without considering the distinguishability of the keywords to be replaced (i.e., $\sum D$), the suggested query quality decreases more than the case without any of the other two factors. It shows that distinguishability plays an important role.

7.3.4 Study of the distinguishability threshold

Besides the query ranking scheme, recall Sect. 3.2, the choice of the distinguishability threshold τ will determine what 'important' keywords to keep in suggestions, thereby may lead to different candidates for suggested queries Q's, which in turn may affect the overall quality of Q's. Therefore, we adopt 4 choices of τ, from strong (0.9) to weak (0), as shown in Table 4.

By comparing the scores in a rowwise way, we can see that the best suggested queries usually do not change even when we set a smaller threshold τ. It is because we have already found the best suggested queries when we set a high τ like 0.9, since preserving the keywords with high distinguishability is more reasonable as discussed in Sect. 3. Later we will also study the impact of τ on the efficiency of our approach in Sect. 7.5.1.

7.3.5 System comparison: XClear versus XRank

To further verify the importance of the *MisMatch Module*, we compare XClear that incorporates our *MisMatch Module* with a well-known LCA-based search engine XRank [12], which only works on XML data without ID references. Therefore, we only test three of the datasets which are without ID references, i.e., IMDB, DBLP and IEEE. For queries with MisMatch problem, XRank may still return a ranked list of query results while XClear returns a ranked list of suggested queries. Therefore, for a fair comparison, we retrieve the results for each suggested query produced by XClear, rank them using XRank's result ranking scheme, and then pick the top-5 results to compare with the top-5 results of XRank. A result is regarded as *relevant* if 8 of the 15 assessors agree that the result does not miss the target; otherwise it is regarded as *irrelevant*. Figure 7 shows the precision of top-5 results of queries on our three datasets, which is calculated as $(number\ of\ relevant\ results\ in\ top\text{-}5\ results)/5$. We find

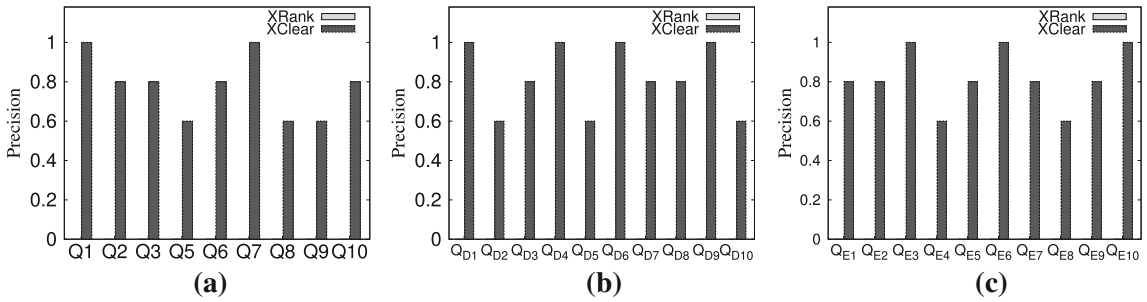

Fig. 7 Top-5 results of XClear for suggested queries versus Top-5 results of XRANK for the original query (XRANK's precision = 0 for all queries). **a** IMDB. **b** DBLP. **c** IEEE

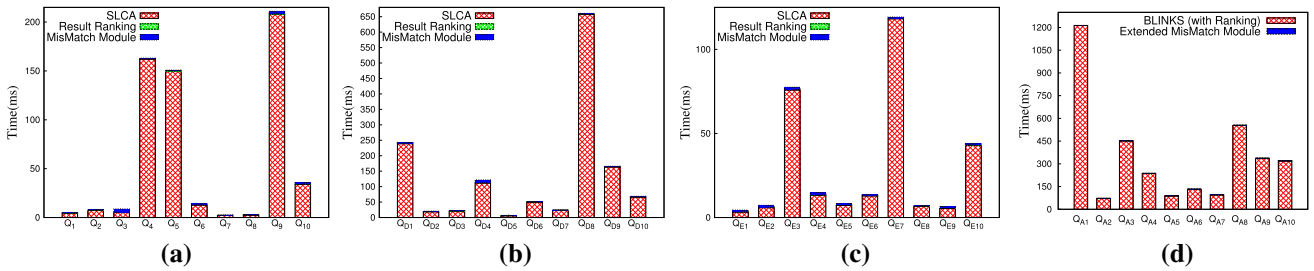

Fig. 8 Processing time for some sample queries (the result ranking time is too small to display). **a** IMDB. **b** DBLP. **c** IEEE. **d** ACMDL (with ID references)

for queries with MisMatch problem, XRank cannot find any relevant result, leading to a precision of zero because XRank is not aware of the fact that what user searches for may not exist, but return the full matches as 'perfect results', which are usually the whole XML data tree.

Another thing to take note here is that, we are measuring the quality of top-5 results in this set of experiment. Even though our suggested queries guarantee to have some results which do not miss the target by Property 1, they should still be ranked higher than those mismatched results and be visible to users. Since the result ranking formula of XRank [12] gives higher scores to more compact results, and those non-mismatched results are more compact than the mismatched results, so the non-mismatched results will be prioritized in the ranked result list.

7.4 Efficiency

For each query in Table 1, we run our algorithm 10 times and collect the average processing time on hot cache, as shown in Fig. 8a. The query result ranking time is too small to display. Moreover, we record the time used by the *MisMatch Module*. We have three observations from Fig. 8a:

(1) The *MisMatch Module* only takes a small portion of the whole query processing time. On average, it is around 4 % for our query set. For the queries on which MisMatch Module spends less than 1ms, it is too small to display in Fig. 8a. Besides, on average the detector spends about 1/40 time of the suggester because it only needs to check the node type of the results as discussed in Sect. 4.

(2) When more suggested queries are generated, the processing time of MisMatch Module is relatively longer. For example, as we can see in Table 1, Q_3 generates more suggested queries than the other queries, so MisMatch Module consumes more time.

(3) For the query that has no MisMatch problem, MisMatch Module introduces a negligibly small time as compared to the query evaluation time because it will terminate once it finds a query result without the MisMatch problem. For example, for Q_4 which intends to find the movie by company Warner Bros, since there exist such kind of movies, Q_4 does not have the MisMatch problem, and our MisMatch Module takes only 0.05 ms.

Figure 8b, c shows the processing time for 10 (out of the total 18) queries on DBLP and IEEE, where we can get similar observations. For ACMDL, the keyword evaluation time by BLINKS dominates the whole query processing time, as shown in Fig. 8d. The processing time of the Extended MisMatch Module for all ten queries is less than 10 ms. So it could be too small to display in Fig. 8d.

7.5 Scalability

Recall that our detector checks all results of a query before concluding the existence of the MisMatch problem, and for each query result, our suggester tries to derive suggested

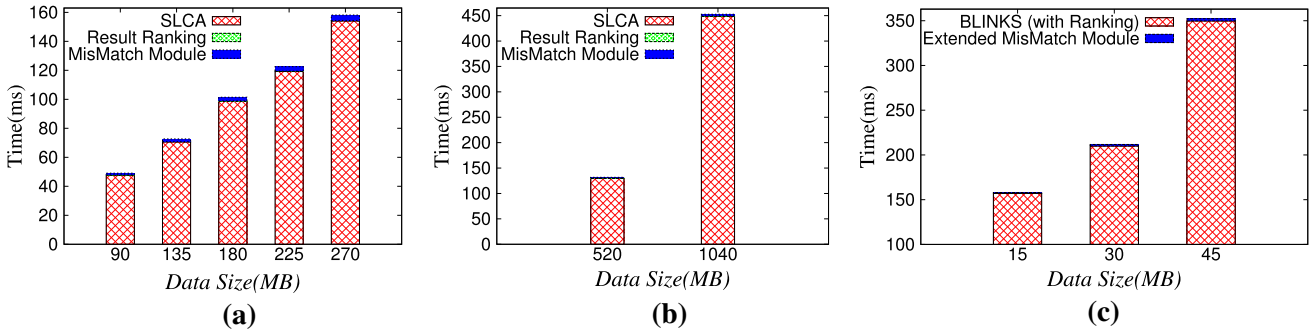

Fig. 9 Impact of data size. **a** IMDB. **b** DBLP. **c** ACMDL (with ID references)

query. Therefore, the processing time of the MisMatch Module should be dependent on the number of suggested queries found, which in turn depends on

- the size of the XML data being queried, and
- the choice of the distinguishability threshold τ, and
- the number of results investigated by MisMatch Module

7.5.1 Sample queries

Firstly, we conduct our scalability test by studying the impact of increasing data size on the MisMatch Module. We run the queries on IMDB and DBLP with different sizes. Since BLINKS is an in-memory approach, we find that it throws out-of-memory errors in our machine if the dataset is larger than 45MB. A recent survey [7] also has a similar conclusion. Therefore, we have to downgrade the size of ACMDL dataset for this experiment. Figure 9 shows the average processing time of one query on the datasets, where we have two observations.

(1) The processing time of the MisMatch Module increases linearly w.r.t. the data size because larger data size leads to possibly larger number of results, and our MisMatch Module needs to check all results to decide the MisMatch existence and find suggestions based on each result.
(2) As the query processing time increases w.r.t. the data size as well, the MisMatch Module only takes around 4 % of the whole query processing time regardless of the data size.

Secondly, we study the impact of the distinguishability threshold τ on the processing time of our MisMatch Module. Figure 10 shows the average number of suggested queries generated for one query w.r.t. different distinguishability threshold τ and the corresponding processing time, where the choice of τ is same as that of the query quality study (in Sect. 7.3.4). As we can see, more suggested queries will be generated when τ is set to be smaller. Meanwhile, it will take longer to process because when threshold τ is set lower,

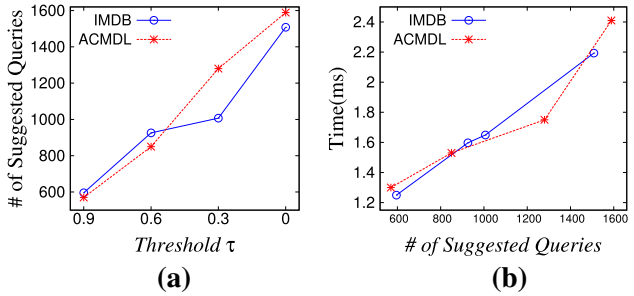

Fig. 10 Impact of distinguishability threshold τ. **a** Impact on suggested queries. **b** Impact on processing time

more keywords will be considered as with acceptably high distinguishability, and we will check more TNT nodes and therefore find out more suggested queries. As discussed in Sect. 7.3, most likely, setting τ to 0.9 can find the same best suggested queries as setting τ to 0.6, 0.3 and even 0.0. So we set τ to 0.9 as a balance between efficiency and effectiveness. To summarize, *MisMatch Module* takes a very small portion of the keyword query processing time, while can come up with some helpful suggested queries to users for possible MisMatch problem.

7.5.2 Random queries

Besides the real-world sample queries, we further study the performance of our MisMatch Module over random queries. Keywords in IMDB and ACMDL datasets are randomly picked to form queries of length 2–5 and those with MisMatch problem will be kept. We record the first 1000 of such queries and count the suggested queries output by our MisMatch Module. The distribution of these queries with different ranges for the number of suggestions is shown in Fig. 11a, from which we find most queries will result in suggested queries no larger than 500. Besides, comparing to IMDB, there are slightly more queries over ACMDL which lead to more than 500 suggested queries. This is because the ID references in ACMDL make the data more linked to each other, such that more replacement can be found. Similar to

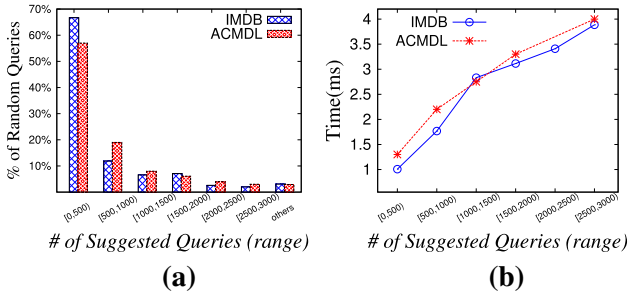

Fig. 11 Scalability test of random queries. **a** Distribution. **b** MisMatch processing time

our findings on sample queries, Fig. 11b reports the linear relationship between the MisMatch Module processing time and the number of suggested queries on random queries.

In this section, we have verified in experiments that our post-processing approach can detect the MisMatch problem with high precision and recall, and subsequently generate useful suggestion to help users. Meanwhile, it is also scalable and light-weight, which only takes around 4 % of whole query processing time.

8 Related work

To the best of our knowledge, so far there is no work on the MisMatch problem in XML keyword search. In the related work, we will look at (1) how the MisMatch problem is handled in other forms of information retrieval; (2) keyword search over XML data without ID references; (3) keyword search over XML data with ID references.

8.1 MisMatch problem in information retrieval

Structured data When structured queries are issued over structured data (relational tables), the MisMatch problem (i.e., what users search for is unavailable in the database) leads to empty result. It has attracted a lot of research efforts such as [17,29–31], where the problem is also known as failing queries, non-answer queries. [31] proposed a method to remove some constraints of the query with the help of approximate functional dependencies and then execute the new queries to finally find some alternative queries. [29,30] proposed another method which adopts the machine learning way to learn some rules from the database. For each failed query, it will find the most similar rule for generating the alternative queries. Recently, [17] also studied how to explain non-answer queries by pinpointing the constraint causing the empty result. Meanwhile, even for queries with meaningful results, some work [10] studied how to recommend some related data in the database which is not part of the result.

Unstructured data When keyword queries are issued over unstructured data (in web search), the MisMatch problem will lead to a list of mismatch results. As discussed in Sect. 1, detecting the MisMatch problem may be challenging. One way to alleviate the problem is to mine some similar and popular queries from query log. [19] tried to modify the query by some pre-computed queries and phrases based on user query log and similarity, which is given by a machine learning model. Later, [42] proposed methods to improve query substitution by selecting a better training set for the machine learning model.

Since results of XML keyword search are very different, which are some subtrees with structure, none of the above techniques consider tree structure and can be used to detect MisMatch problem in XML keyword search. Our solution makes use of the unique tree structure information in XML, i.e., node types, to find some useful suggestion for users.

Compared with our previous work in [39], this article includes the following additional materials: (1) We have improved the definition of Target Node Type in Sect. 2.2 to cover a more general case, where users' query keywords could match the same type of nodes multiple times. For example, users giving two different laptop model names in the query are with different intention with those who give one laptop model name in the query. (2) For the process of finding explanation and suggested queries in Sect. 3.2, we further consider the situation where the important keywords are derived from multiple *keyword match nodes*. (3) We have extended the MisMatch solution to resolve the MisMatch problem over XML data with ID references in Sect. 6, resulting in a general framework of solving the MisMatch problem. (4) We have conducted extensive experiments in Sect. 7 to evaluate the effectiveness, efficiency and scalability of our proposed extension in addressing the MisMatch problem over XML IDRef Digraph. In particular, real dataset ACM Digital Library (ACMDL) is used and the complete XClear system (with general solution to address both XML data tree and XML digraph) is re-evaluated for queries and XML data without IDRef in our earlier work [39].

8.2 Keyword search on XML data without ID references

An XML document without ID references is usually modeled as a tree. The *first* part of the research efforts is the definition of matching semantics. LCA (lowest common ancestor) semantics is first proposed in [12] to find XML nodes, each of which contains all query keywords within its subtree. For a given query $Q = \{k_1, \ldots, k_n\}$ and an XML document D, L_i denotes the inverted list of k_i. Then the LCAs of Q on D are defined as $LCA(Q)=\{v|v = lca(m_1, \ldots, m_n), v_i \in L_i (1 \leq i \leq n)\}$. Subsequently, SLCA (smallest LCA [15,38]) is proposed, which is indeed a subset of $LCA(Q)$, of which no LCA in the subset is the ancestor of any other LCA. ELCA

[12], which is also a widely adopted subset of $LCA(Q)$, is defined as: a node v is an ELCA node of Q if the subtree T_v rooted at v contains at least one occurrence of all query keywords, after excluding the occurrences of keywords in each subtree $T_{v'}$ rooted at v's descendant node v' and already contains all query keywords. [24] proposed Valuable LCA (VLCA) by eliminating redundant LCAs that should not contribute to the answer, but also retrieves the false negatives filtered out wrongly by SLCA. Recently, structural consistency [22] is proposed to further constrain LCA s.t. no query result has an ancestor-descendant relationship at the schema level with any other query results. The *second* part is the proposals of efficient result retrieval methods based on a certain matching semantics: [26,38] for computing SLCA nodes and [12,35] for computing ELCA nodes.

Moreover, improving user experience is studied in various ways [2–4,24,25,28], but none of them is aware of the MisMatch problem. [2,3] proposed a statistical way to identify the search target candidates. [35] studied the problem of finding the nearest node containing a specific keyword to a given node. [36] proposed a ranking approach for keyword queries based on an extension of the concepts of data dependencies and mutual information. Sun et al. [34] generalize SLCA to support keyword search involving combinations of AND and OR boolean operators. XSeek [26] generates the return nodes which can be explicitly inferred by keyword match pattern and the concept of entities in XML data. [27] further proposed an axiomatic way to decide whether a result is relevant to a keyword query in term of the monotonicity and consistency properties w.r.t the XML data and query. [28] studied how to differentiate the search results of an XML keyword query, aiming to save user efforts in investigating and comparing potentially large results.

However, all the keyword search methods above are trying to find sub-structures containing all query keywords, and the MisMatch problem may exist in all of them. Since our technique to solve the MisMatch problem is orthogonal to keyword search methods, our technique can be easily deployed to all existing keyword search methods.

8.3 Keyword search on XML data with ID references

An XML document considering ID references is usually modeled as a directed graph (digraph) [16]. Keyword search on a digraph is usually reduced to the Steiner tree problem or its variants: given a digraph $G = (V, E)$, where V is a set of nodes and E is a set of edges, a keyword query result is defined as a minimal directed subtree T in G such that the leaves or the root of T contain all keywords in the query. The Steiner tree problem is NP-complete [9], and many works are interested in finding the "best" answers of all possible Steiner trees, i.e., finding top-K results according to some criteria, like subtree size (sum of length of all edges in the subtree), diameter (maximum distance between any two nodes in the subtree), etc. Backward expanding strategy is used by BANKS [5] to search for Steiner trees in a digraph. It starts the searching from the nodes which directly contain the query keywords. Then it concurrently runs multiple threads to traverse from those nodes until they find some common nodes which connect to all query keywords. To improve the efficiency, BANKS-II [20] proposed a bidirectional search strategy to reduce the search space, which searches as small portion of digraph as possible. It starts a backward searching from the nodes directly containing the keywords. Meanwhile, it also conducts a forward searching starting from the nodes which have been visited during backward searching. Later [8] designed a dynamic programming approach (DPBF) to identify the top-k Steiner trees containing all query keywords. With some slightly modification on DPBF, a variant of DPBF to output the top-k results in increasing weight order is also proposed in the work. BLINKS [14] proposes a bi-level index and a partition-based method to prune and accelerate searching for top-k results in a digraph. It first divides the XML nodes into several blocks. Then it builds intra-block index and inter-block index for all the nodes. With the index which conveys the connectivity information among and within the blocks, it can prune some unnecessary search space. XKeyword [16] presented a method to optimized the query evaluation by making use of the schema of the XML document. It infers the possible schema structure of the potential results such that it can avoid some search space which will not lead to any results complied with that structure. But it is still orders of magnitude slower than BLINKS as it involves a lot of table joins [40]. Recently STAR [21] proposed an approximate approach which returns near-optimal Steiner trees as results. The results are returned quickly and then iteratively refined.

9 Conclusion and future work

In this paper, we first identified and defined the MisMatch problem, in which what user intends to search for does not exist in the XML data. Then we proposed a practical way to detect the MisMatch problem and generate helpful suggestions to users based on two novel concepts that we introduce: *Target Node Type* and *distinguishability*. Our approach can be viewed as a post-processing job of query evaluation and has four main features: (1) both detector and suggester are result-driven; (2) it adopts explanations, suggested queries and their sample results as output, helping users judge whether the MisMatch problem is solved without reading all query results; (3) it is portable as it is orthogonal to the choice of result retrieval method, which can work with any LCA-based matching semantics (for XML without ID references) or MST-based matching semantics (for XML with ID ref-

erences); (4) it is lightweight as it occupies a very small proportion of the whole query evaluation time.

Acknowledgments H.V. Jagadish is supported in part by NSF IIS-1250880 and IIS-1017296.

References

1. Berkeley, D.B.: http://www.sleepycat.com
2. Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective xml keyword search with relevance oriented ranking. In: ICDE (2009)
3. Bao, Z., Lu, J., Ling, T.W., Chen, B.: Towards an effective xml keyword search. IEEE Trans. Knowl. Data Eng. **22**(8), 1077–1092 (2010)
4. Bao, Z., Lu, J., Ling, T.W., Xu, L., Wu, H.: An effective object-level xml keyword search. In: DASFAA (2010)
5. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using banks. In: ICDE (2002)
6. Chapman, A., Jagadish, H.V.: Why not? In: SIGMOD (2009)
7. Coffman, J., Weaver, A.C.: An empirical performance evaluation of relational keyword search techniques. IEEE Trans. Knowl. Data Eng. **26**(1), 30–42 (2014)
8. Ding, B., Yu, J.X., Wang, S., Qin, L., Zhang, X., Lin, X.: Finding top-k min-cost connected trees in databases. In: ICDE (2007)
9. Dreyfus, S.E., Wagner, R.A.: The Steiner problem in graphs. In: Networks (1971)
10. Drosou, M., Pitoura, E.: Ymaldb: exploring relational databases via result-driven recommendations. VLDB J. **22**(6), 849–874 (2013)
11. Goldman, R., Widom, J.: Dataguides: enabling query formulation and optimization in semistructured databases. In: VLDB (1997)
12. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: Xrank: ranked keyword search over xml documents. In: SIGMOD (2003)
13. Hadjieleftheriou, M., Chandel, A., Koudas, N., Srivastava, D.: Fast indexes and algorithms for set similarity selection queries. In: ICDE (2008)
14. He, H., Wang, H., Yang, J., Yu, P.S.: Blinks: ranked keyword searches on graphs. In: SIGMOD (2007)
15. Hristidis, V., Koudas, N., Papakonstantinou, Y., Srivastava, D.: Keyword proximity search in xml trees. IEEE Trans. Knowl. Data Eng. **18**(4), 525–539 (2006)
16. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword proximity search on xml graphs. In: ICDE (2003)
17. Huang, J., Chen, T., Doan, A., Naughton, J.F.: On the provenance of non-answers to queries over extracted data. PVLDB **1**(1), 736–747 (2008)
18. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of ir techniques. ACM Trans. Inf. Syst. **20**(4), 422–446 (2002)
19. Jones, R., Rey, B., Madani, O., Greiner, W.: Generating query substitutions. In: WWW (2006)
20. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB (2005)
21. Kasneci, G., Ramanath, M., Sozio, M., Suchanek, F.M., Weikum, G.: Star: Steiner-tree approximation in relationship graphs. In: ICDE (2009)
22. Lee, K.H., Whang, K.Y., Han, W.S., Kim, M.S.: Structural consistency: enabling xml keyword search to eliminate spurious results consistently. VLDB J. **19**(4), 503–529 (2010)
23. Lemire, D., Kaser, O., Aouiche, K.: Sorting improves word-aligned bitmap indexes. Data Knowl. Eng. **69**(1), 3–28 (2010)
24. Li, G., Feng, J., Wang, J., Zhou, L.: Effective keyword search for valuable lcas over xml documents. In: CIKM (2007)
25. Li, G., Li, C., Feng, J., Zhou, L.: Sail: structure-aware indexing for effective and progressive top-k keyword search over xml documents. Inf. Sci. **179**(21), 3745–3762 (2009)
26. Liu, Z., Chen, Y.: Identifying meaningful return information for xml keyword search. In: SIGMOD (2007)
27. Liu, Z., Chen, Y.: Reasoning and identifying relevant matches for xml keyword search. PVLDB **1**(1), 921–932 (2008)
28. Liu, Z., Sun, P., Chen, Y.: Structured search result differentiation. PVLDB **2**(1), 313–324 (2009)
29. Muslea, I.: Machine learning for online query relaxation. In: KDD (2004)
30. Muslea, I., Lee, T.J.: Online query relaxation via bayesian causal structures discovery. In: AAAI (2005)
31. Nambiar, U., Kambhampati, S.: Answering imprecise queries over autonomous web databases. In: ICDE (2006)
32. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill Inc., New York (1986)
33. Schmidt, A., Kersten, M.L., Windhouwer, M.: Querying xml documents made easy: nearest concept queries. In: ICDE (2001)
34. Sun, C., Chan, C.Y., Goenka, A.K.: Multiway slca-based keyword search in xml data. In: WWW (2007)
35. Tao, Y., Papadopoulos, S., Sheng, C., Stefanidis, K., Stefanidis, K.: Nearest keyword search in xml documents. In: SIGMOD (2011)
36. Termehchy, A., Winslett, M.: Using structural information in xml keyword search effectively. ACM Trans. Database Syst. **36**(1), 4 (2011)
37. Vesper, V.: http://www.mtsu.edu/vvesper/dewey.html
38. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest lcas in xml databases. In: SIGMOD (2005)
39. Zeng, Y., Bao, Z., Ling, T.W., Jagadish, H.V., Li, G.: Breaking out of the mismatch trap. In: ICDE (2014)
40. Zeng, Y., Bao, Z., Ling, T.W., Li, G.: Efficient xml keyword search: from graph model to tree model. In: DEXA (2013)
41. Zeng, Y., Bao, Z., Ling, T.W., Li, G.: Removing the mismatch headache in xml keyword search. In: SIGIR (2013, demo paper. http://xclear.comp.nus.edu.sg)
42. Zhang, W.V., He, X., Rey, B., Jones, R.: Query rewriting using active learning for sponsored search. In: SIGIR (2007)