Breaking out of the MisMatch Trap

Yong Zeng * Zhifeng Bao * Tok Wang Ling * H.V. Jagadish + Guoliang Li #

*National University of Singapore +University of Michigan #Tsinghua University

zengyong,baozhife,lingtw@comp.nus.edu.sg jag@umich.edu liguoliang@tsinghua.edu.cn

***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*. 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 LCA-based matching semantics and 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.**

I. 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 like product removed from shelves, clothes size unavailable, etc., 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 keyword 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 [8], [19], [17], [18], 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 [25], users suffered from such a problem for 27% of their queries. This is our central concern in this paper.

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: black, blue and white.”

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 [8], [3] try to explain the non-answer queries by pinpointing the constraint causing the empty result (explanation part). Some works [19], [17], [18] focus on generating some alternative constraint to come up with a suggested query (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 [10], [26] 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. Now, let us take a look at how the MisMatch problem behaves in such context.

Example 1: An XML data tree in Figure 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 [21], SLCA [24], ELCA [5] or even the most recent variant [11] of LCA, still can find some results containing all query keywords. One of

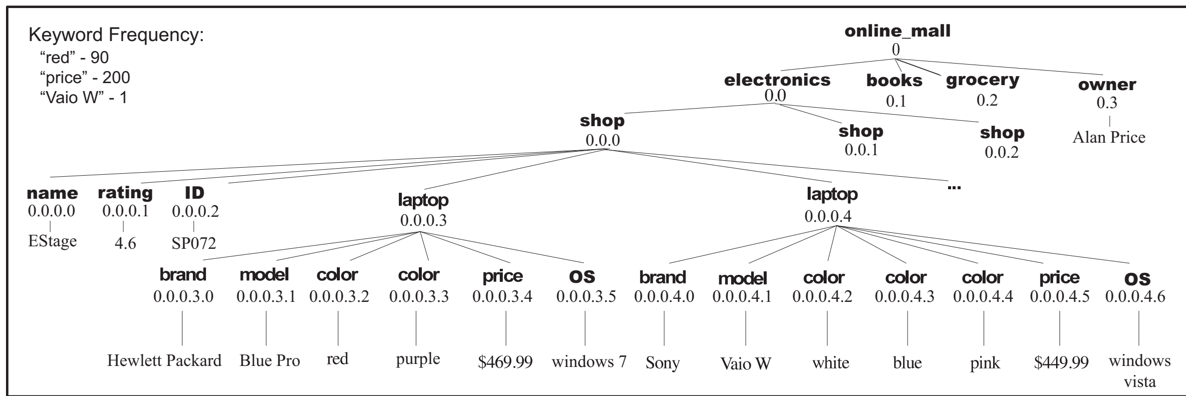


Fig. 1. Sample XML Document about an Online Shopping Mall

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 worse, there could be hundreds of shops selling Viao W and therefore hundreds of mismatch results are returned. □

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 Sec. III 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 [20] 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 of our algorithm 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 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 in this paper 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 generate 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 build a search engine called XClear [25] which embeds the MisMatch problem detector and suggester. Extensive experiments have verified the effectiveness, efficiency and scalability of our method.

We present preliminaries in Sec. II. Detecting the MisMatch problem is in Sec. III. Sec. IV discusses how to find the explanations and suggested queries. Sec. V presents our labeling scheme for efficient approximate results detection. Sec. VI presents indices and algorithms. Experiments are in Sec. VII, related works are in Sec. VIII and we conclude in Sec. IX.

II. PRELIMINARIES

A. Data Model

We model data-centric XML 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 [23]. As shown in Figure 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* a keyword k if there exists a *keyword match node* w.r.t. k within the subtree rooted at n. □

E.g., in Figure 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* keyword 'red', as its descendant $color$:0.0.0.3.2 contains 'red'.

B. General Query Result Format

To define a general format to represent the query results, let us look at the existing matching semantics first. All existing matching semantics so far, such as SLCA [24], [7], ELCA [5], entity-based SLCA [15] are all based on the concept of lowest common ancestor (LCA). Let $lca(m_1, ..., m_n)$ be the lowest common ancestor of nodes m_1,...,m_n. For a given query Q = $\{k_1$,...,$k_n\}$ and an XML document D, L_i denotes the inverted list of k_i. Then the LCAs of Q on D are defined as $LCA(Q)$ = $\{v \mid v = lca(m_1, ..., m_n), m_i \in L_i (1 \leq i \leq n)\}$. Both SLCA and ELCA define a subset of $LCA(Q)$, and we refer readers to Sec. VIII for detailed definitions of SLCA and ELCA, and their relationships with LCA.

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

$$r = (v_{lca}, \{m_1, m_2, ..., m_n\})$$

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,..., m_n, i.e. v_{lca} = $lca(m_1, ..., m_n)$. □

Defn. 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*) [7], [13] and subtree Return (returning the subtree rooted at each LCA node) [5], [24]. This observation is important in explaining the *portability* feature of our solution to detect and resolve the MisMatch problem later in Sec. IV-D.

III. DETECTING THE MISMATCH PROBLEM

In this section, we would like to present how to detect the MisMatch problem.

First, the detector should infer user's possible search target(s) based on the query results. 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 or the price of a product in Figure 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 define a concept called *Target Node Type (TNT)* to denote the node type 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*. E.g. in Example 1, $Q = \{$'$Vaio$','W','red','$price$'$\}$, and one possible result $r = (0.0.0, \{0.0.0.4.1, 0.0.0.4.1, 0.0.0.3.2, 0.0.0.4.5\})$. The node types can simulate the semantics of each *keyword match node*:

'Vaio': $\{online_mall/electronics/shop/laptop/model\}$
'W': $\{online_mall/electronics/shop/laptop/model\}$
'red': $\{online_mall/electronics/shop/laptop/color\}$
'price': $\{online_mall/electronics/shop/laptop/price\}$.

Recall Example 1, user's search intention is a laptop, which corresponds to the node type "online_mall/electronics /shop/laptop" being closely related to these four node types.

Following a similar philosophy of LCA, which finds the lowest/smallest nodes connecting all query keywords as the most relevant and meaningful results, TNT is defined as the lowest node type which connects to all those node types at schema level:

Definition 5: **Target Node Type (TNT) for a single query result.** Given a query $Q = \{k_1, k_2, ..., k_n\}$ and a query result $r = (v_{lca}, \{m_1, m_2, ..., m_n\})$ on an XML document D, the

Target Node Type for this particular result r, namely $TNT(r)$, is defined as:

$$TNT(r) = LCP(m_1.type, m_2.type, ..., m_n.type)$$

where $m_i.type$ is the node type of m_i, and LCP represents the longest common path of a set of node types. Nodes of a specific TNT are called *TNT nodes*. □

In the laptop example above, according to Defn. 5, $TNT(r) = \{online_mall/electronics/shop/laptop\}$, even though no laptop can meet all the requirements at data level. The TNT, which is laptop, is the lowest node type connecting to laptop model, laptop color and laptop price.

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 (see Defn. 5) of r and the actual root of r, namely v_{lca}.

Definition 6: Given a query $Q = \{k_1, k_2, ..., k_n\}$ and a query result $r = (v_{lca}, \{m_1, m_2, ..., m_n\})$ 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 1, $v_{lca}.type$ = *shop*≠*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. □

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.

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.

Moreover, users usually investigate the retrieved results starting from the *top-ranked* ones. Therefore, without loss of generality, we can also easily extend Defn. 7 by considering the top-K retrieved results of Q.

Time Complexity of the detector is O($|\mathbb{R}|$), which is very efficient. As discussed in Sec. VI-A 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 constants.

IV. Finding Explanations and Suggested Queries

As discussed in Sec. III, 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=$\{k_1, k_2, ..., k_n\}$ and a mismatch query result r=(v_{lca},$\{m_1, m_2, ..., m_n\}$), 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},$\{m'_1, m'_2, ..., m'_n\}$), 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'_j that has the same node type as m_i ($i, j \in [1, n]$). 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'_j (in approximate result) in step 2.

Step 1 is illustrated in Sec. IV-A, and the last two steps are described in Sec. IV-B. Lastly, we complement our suggester by discussing how to rank the suggested queries in Sec. IV-C.

A. Distinguishability

In this section, we will present a concept to measure the importance of the 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 Figure 1, keyword 'blue' can match both $model$:0.0.0.3.1 and $color$:0.0.0.4.3. When it matches a model name, most likely it is important since few model names contain the keyword 'blue'; when it matches a 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 Sec. IV, $\mathbb{K}$ 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*) [20], 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 documents, 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 [6], so we follow [6] 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 2: Consider a keyword 'price' in Figure 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({'price'},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({'price'},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 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:

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

It is easy to verify that the range of distinguishability is (0,1].

B. 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[1], 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. Then the remaining task is to find the approximate results, each containing the important keywords $\mathbb{K}$, from which suggested queries are inferred.

We derive important keywords from the *keyword match nodes* and find the approximate results as follows:
Given a user query Q and a mismatch query result r=(v_{lca},$\{m_1, m_2, ..., m_n\}$), 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 $(v_{lca}/p_1/p_2/.../p_j/m_i)$, where p_1,p_2,....,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. $v'_{lca} \in \{p_1, p_2, ..., p_j, m_i\}$), and verify whether the subtree rooted at v'_{lca} can form an approximate query result r'=(v'_{lca},$\{m'_1, m'_2, ..., m'_n\}$) w.r.t. r.

We find that an ideal approximate result r' should at least have the following two properties:

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

If we find a result r' that satisfies the above two properties, then we say

$$r' = (v'_{lca}, \{m'_1, m'_2, ..., m'_n\})$$

is an **approximate query result of** Q.

Recall Sec. III, the reason that leads to the MisMatch problem is: there does not exist a single *TNT node* that can *subtree-contain* all query keywords. Therefore, P1 is specified to ensure the approximate result should be consistent with one of the TNTs that user intends to search for. In other words, P1 is to ensure v'_{lca} of 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, i.e. 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.

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.

Example 3: For query Q ={'Vaio','W','red','price'} issued on Figure 1's XML data tree, one query result is r=(0.0.0,

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

{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*: $D(\{`Vaio', `W'\}, model) = 100\%$, $D(\{`red'\}, color) = 68.2\%$, $D(\{`price'\}, price) = 0.5\%$.

Since D({'Vaio', 'W'},model)>τ=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). By Defn. 5 $TNT(r) = laptop$, 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. E.g. 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: $(0.0.0.4.2).type = color = (0.0.0.3.2).type$. As a result, 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 mismach 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. □

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 (Sec. VII) that most likely it is not necessary.

C. 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, 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}, denoted as dt (dt is equal to the length difference of their Dewey labels). The larger dt is, the higher $score$ should be. Because a more compact subtree is preferred.
3) Sum of distinguishability of the keywords that need to be changed, 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:

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

D. 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 (Defn. 4), our approach can work with any LCA-based matching semantics (recall Sec. II-B); 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 Sec. IV-B 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 Sec. V later.

V. Efficient Approximate Results Detection

Recall Sec. IV-B, 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.

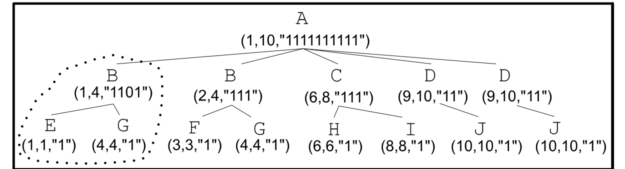


Fig. 2. An XML Tree with Nodes Labeled by exLabels

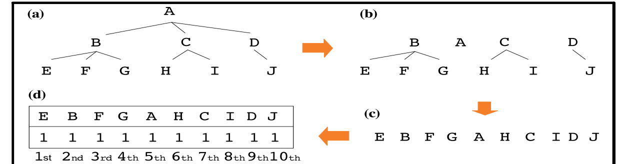


Fig. 3. Schema Tree Flattening and Virtual Bitmap Construction

A. 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 [4]. E.g., for the XML data in Figure 2, we can construct a schema tree as shown in Figure 3(a), where each node in the schema tree represents a unique node type. Note that each node in Figure 3 should be a node type represented as a path (according to Defn. 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 3(a), (b) and (c) show such a process of flattening.
- Construct a virtual $bitmap$ as shown in Figure 3(d). 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 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 4: In Figure 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 Figure 3(d), 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. □

B. Logical Operation

Similar to node labeling, for a query result $r = (v_{lca}, \{m_1, m_2, ..., m_n\})$, 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 5: In Figure 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 Figure 2, whose exLabel is (1,4,'1101'), then we know the approximate query result exists because (2,4,'101') ⊆ (1,4,'1101'). □

VI. Index Construction and Algorithms

A. 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 Sec. V-A. For each node n visited, we assign a Dewey label $deweyID$ [23] 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 Sec. IV-B, 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 [12]. Operation getDist($deweyID$,$\mathbb{K}$) returns the distinguishability of a set of keywords $\mathbb{K}$ w.r.t. the type of the node with $deweyID$.

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      foreach nd ∈ r.matchnodes do
10         if getDist(nd.dewey, nd.keywords)>τ then
11             foreach n ∈ nodes on the path from r.v_lca to nd
               AND n.type = getTNT(r) do
12                 if contain(getExLabel(n.dewey), rExlabel) then
13                     QuerySuggester(n, r, suggestedQueries);
14 return suggestedQueries.sort();
```

B. Solving the MisMatch problem

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

TABLE I
10 OF THE SAMPLE QUERIES ON IMDB

IMDB:90MB			
#	Query	suggested queries	best-3 suggested queries (Format: explanation → suggested options)
Q1	Gladiator Spanish	5	(language): Spanish → English / Japanese / French
Q2	Spielberg DiCaprio Action movie	6	(genres): Action → Biography / Crime / Drama
Q3	Neo hacker phonebooth	3061	(keyword): phonebooth → computer / software / programmer
Q4	Warner Bros. movie	0	None
Q5	Italy Betty Fisher	12	(country): Italy → France / Canada / USA
Q6	Spielberg Schwarzenegger	58	(name): Schwarzenegger→ Meredith Brooks / Jim Conroy / Dean Spunt
Q7	Terminator 3 cast Sarah	19	(name): Sarah → Nick Stahl / Claire Danes / Kristanna Loken
Q8	Panic Room 2001	11	(year): 2001 → 2002 (title): Panic Room → Promised Land / Nowhere Road
Q9	Ettore The Man movie	1189	(director): Ettore → Ethan Coen / Salvatore Maira / Massimo Sani
Q10	boy death ghost love	992	(keyword): love → orphanage / bully / bomb

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 Sec. V-B.

For *Suggester*, it checks each *keyword match node* nd of the query result (line 9). If the distinguishability is larger than the threshold τ (line 10), 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 11). Whether an approximate query result exists can be easily checked by examining the containment relationship between the exLabels (line 12), 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 13).

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** $nd \in r.matchnodes$ **do**
3   **if** nd *is not a descendant of* v'_{lca} **then**
4   replace[i++] = getReplacement($nd.type$, $v'_{lca}.dewey$);
5 **foreach** $n_1 \in replace[1]$,...,$n_i \in replace[i]$ **do**
6   $sugQueries$ = $sugQueries \bigcup$ ($r.matchnodes[1] \rightarrow n_1$,..., $r.matchnodes[i] \rightarrow n_i$);

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 Sec. IV-B (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 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 presents the function $contain()$ to examine the containment relationship between two exLabels, i.e., the first contains the second. As discussed in Section V-B, 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).

Algorithm 3: contain(elx, ely)

input : exLabel elx and exLabel ely
output : a boolean indicating whether elx contains ely

1 **if** ($elx.a \leqslant ely.a$ *and* $ely.b \leqslant elx.b$)==*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 $false$;

VII. 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 D&S*.

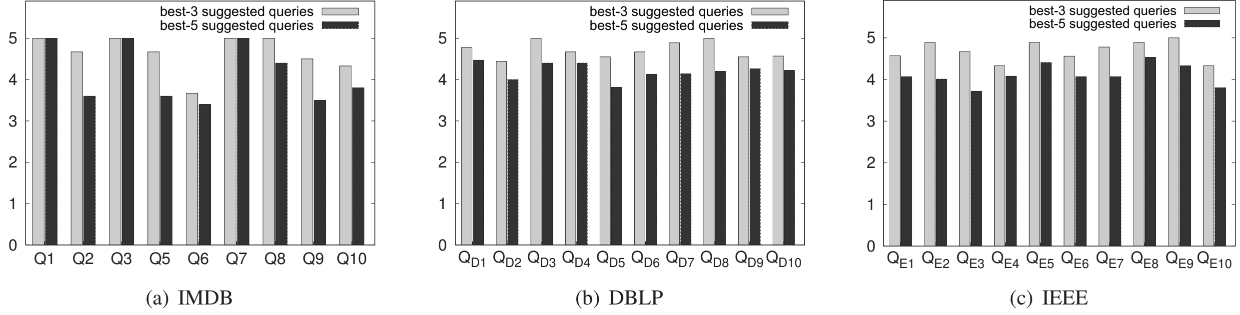
A. Experimental Settings

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: (1)IMDB[2] 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[3].

Query Set. Our query set contains 18 queries for each of the datasets, all of which are collected from the real-world user log data of our system. 10 sample queries for IMDB and their best-3 suggested queries (if any) are shown in Table I. Besides, 1000 random queries are generated for each dataset

[2]http://www.imdb.com/interfaces
[3]https://inex.mmci.uni-saarland.de/



(a) IMDB (b) DBLP (c) IEEE

Fig. 4. Average Quality Measure of Suggested Queries (for the Testing Queries with MisMatch Problem)

as well (see Sec. VII-F.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) out of the 18 queries for IMDB (DBLP, IEEE) have the MisMatch problem.

Keyword Search Method. Here we choose SLCA [24], 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 [5].

B. Frequency of the MisMatch Problem

We have done a survey among 15 participants. Each participant is required to issue 30 queries in XClear [25], an XML keyword search engine, to find some movies they want to watch in the IMDB dataset. Each participant is asked to judge whether her queries have the MisMatch problem according to the query results. The same experiments are also conducted on DBLP and IEEE datasets. We find that, averagely users suffered from such a problem for 27% of their queries.

TABLE II
SENSITIVITY OF THE MISMATCH DETECTOR

	IMDB dataset	DBLP dataset	IEEE dataset
Precision	90%	91%	100%
Recall	100%	100%	100%

C. 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 II. 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).
(2) A non-perfect precision tells that we may accidentally identify some queries without MisMatch problem as problematic. E.g. for a query 'Joel Ethan' issued on IMDB, no person in database has such a name. For such an ambiguous query, it is not easy to know 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 may infer *director* as the TNT, but users intending to find a movie related to two persons will not be affected, because they can simply ignore our suggestion. Note that in fact no existing approach can solve the ambiguous query thoroughly [2].

D. Quality of the Suggested Queries

We first have a glance at how explanations and suggestions look like for real-world queries in Table I. For Q8, '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 I 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). Q3 has 3061 suggestions, because Q3 has a large number of results, and our suggester works by checking each result to generate suggestions (if any).

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 and IEEE, we ask three CS research students and three undergraduates in other faculties; for IMDB, we ask three movie fans and three non-fans. The participants are shown the matching results of each query, the best-5 suggested queries together with the corresponding sample query results. *Second*, the participants are asked to score the quality of each suggested query by using the Cumulated Gain-based evaluation (CG) metric [9] (from 0 to 5 points, 5 means best while 0 means worst). In contrast to traditional metrics like precision and recall which adopt a

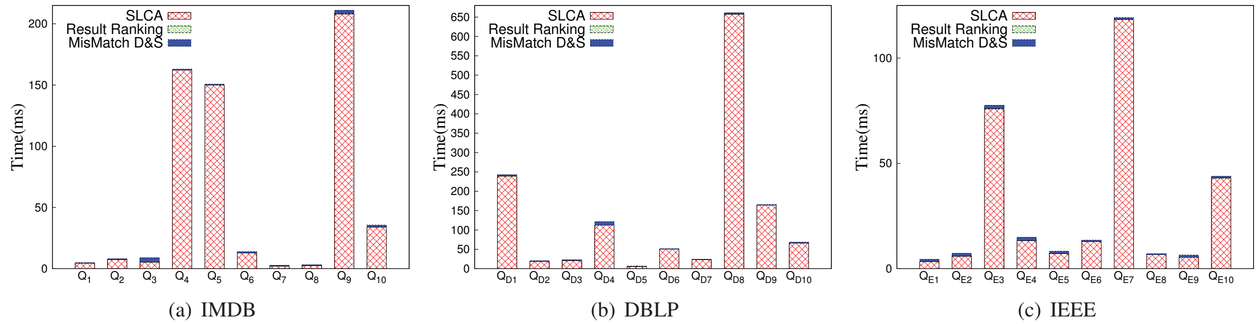


Fig. 5. Processing Time for some Sample Queries (The result ranking time is too small to display.)

binary judgement (yes or no), CG is aware of the fact that all results are not of equal relevance to user.

2) Evaluation of Overall Quality: The average scores for best-3 and best-5 suggestions are shown in Figure 4 [4]. 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 participants because new keywords and old keywords are not semantically similar, such as the replacement for Q10 in Table I. 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.

TABLE III

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

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 Sec. IV-C. The scores for the suggested queries of each case are shown in Table III. Please ignore the choice of τ for the time being. By comparing the scores in a columnwise way, we find:

[4]Here by default we adopt τ = 0.9. Experiment on effects of threshold setting is discussed in Sec. VII-D.4.

(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.

4) Study of distinguishability threshold: Besides the query ranking scheme, recall Sec. IV-B, 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 III.

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 Sec. IV. Later we will also study the impact of τ on the efficiency of our approach in Sec. VII-F.1.

E. Sample Query Processing Time

For each query in Table I, we run our algorithm 10 times and collect the average processing time on hot cache, as shown in Figure 5(a). The query result ranking time is too small to display. Moreover, we record the time used by the *MisMatch D&S* part. We have three observations from Figure 5(a):
(1) The *MisMatch D&S* 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 D&S spends less than 1ms, it is too small to display in Figure 5(a). 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 Sec. V.
(2) When more suggested queries are generated, the processing time of MisMatch D&S is relatively longer. E.g., as we can see in Table I, Q3 generates more suggested queries than the other queries, so MisMatch D&S consumes more time.
(3) For the query that has no MisMatch problem, MisMatch D&S 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. E.g. for Q4 which intends to find the movie by company Warner Bros, since there exist such kind of movies, Q4 does not have the MisMatch problem, and our MisMatch D&S takes only 0.05ms.

Figure 5(b) and 5(c) show the processing time for 10 (out of the total 18) queries on DBLP and IEEE, where we can get similar observations.

F. Scalability Test

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

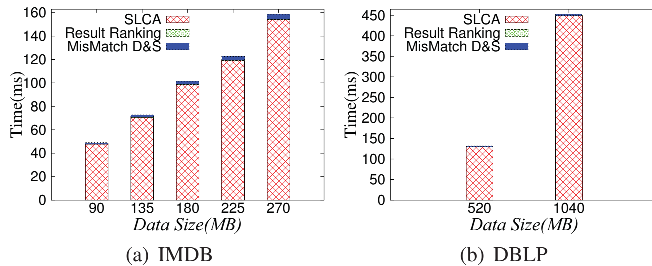
- the size of the XML data being queried, and
- the choice of the distinguishability threshold τ, and
- the number of results investigated by MisMatch D&S

1) Sample Queries:

Firstly, we conduct our scalability test by studying the impact of increasing data size on the MisMatch D&S. We run the queries on IMDB and DBLP with different sizes. Figure 6 shows the average processing time of one query on the datasets, where we have two observations.

(1) The processing time of the MisMatch D&S increases linearly w.r.t. the data size. Because larger data size leads to possibly larger number of results, and our D&S needs to check all results to decide the MisMatch existence and find suggestions based on each result.

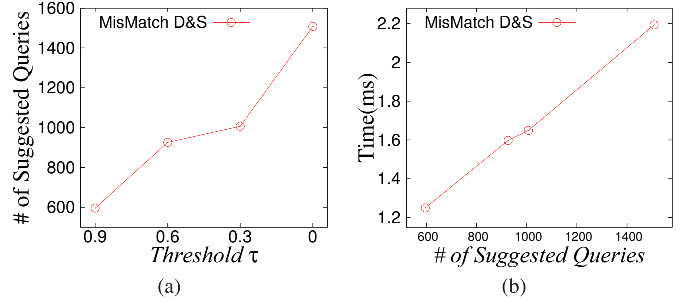
(2) As the query processing time increases w.r.t. the data size as well, the MisMatch D&S only takes around 4% of the whole query processing time regardless of the data size.



(a) IMDB (b) DBLP

Fig. 6. Impact of Data Size.

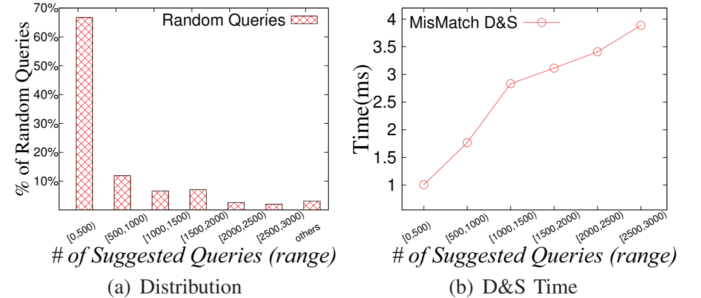
Secondly, we study the impact of the distinguishability threshold τ on the processing time of our MisMatch D&S. Figure 7 shows the average number of suggested queries generated for one query w.r.t. different distinguishability threshold τ and the corresponding processing time, where the choice of τ is same as that of the query quality study (in Sec. VII-D.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, 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 Sec. VII-D, 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 D&S* 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.



(a) (b)

Fig. 7. Impact of Distinguishability Threshold τ

2) Random Queries: Besides the real-world sample queries, we further study the performance of our D&S over random queries. Keywords in IMDB dataset are randomly picked to form queries of length 2$\sim$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 D&S. The distribution of these queries with different ranges for the number of suggestions is shown in Figure 8(a), from which we find most queries will result in suggested queries no larger than 500. Similar to our findings on sample queries, Figure 8(b) reports the linear relationship between the D&S time and the number of suggested queries on random queries.



(a) Distribution (b) D&S Time

Fig. 8. Scalability Test of Random Queries

VIII. 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) related work in XML keyword search.

MisMatch Problem in Information Retrieval.

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 [8], [19], [17],

[18], where the problem is also known as failing queries, non-answer queries. [19] 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. [17], [18] 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, [19] also studied how to explain non-answer queries by pinpointing the constraint causing the empty result.

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 Sec. I, detecting the MisMatch problem may be challenging. One way to alleviate the problem is to mine some similar and popular queries from query log. [10] 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, [26] proposed methods to improve query substitution by selecting a better training set for the machine learning model.

Since the 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.

XML Keyword Search.

The *first* part of the research efforts is the definition of matching semantics. LCA (lowest common ancestor) semantics is first proposed in [5] to find XML nodes, each of which contains all query keywords within its subtree. For a given query Q = $\{k_1,...,k_n\}$ and an XML document D, L_i denotes the inverted list of k_i. Then the LCAs of Q on D are defined as $LCA(Q)$=$\{v \mid v = lca(m_1, ..., m_n), v_i \in L_i(1 \leq i \leq n)\}$. Subsequently, SLCA (smallest LCA [24], [7]) 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 [5], 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. Recently, structural consistency [11] 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: [24], [15] for computing SLCA nodes and [5], [22] for computing ELCA nodes. Moreover, improving user experience is studied in various ways [16], [13], [14], [2], but none of them is aware of the MisMatch problem.

IX. Conclusions

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. 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 the output to users, helping users judge whether the MisMatch problem is solved without reading all query results; (3) it is portable as it can work with any LCA-based matching semantics and orthogonal to the choice of result retrieval method; (4) it is lightweight as it occupies a very small proportion of the whole query evaluation time.

Acknowledgement. Guoliang Li is partially supported by "NExT Research Center" funded by MDA, Singapore, under Grant No. WBS:R-252-300-001-490.

References

[1] *Berkeley DB*. http://www.sleepycat.com.
[2] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, 2009.
[3] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, 2009.
[4] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.
[5] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over xml documents. In *SIGMOD*, 2003.
[6] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.
[7] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in xml trees. In *TKDE*, 2006.
[8] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 2008.
[9] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.*, 2002.
[10] R. Jones, B. Rey, O. Madani, and W. Greiner. Generating query substitutions. In *WWW*, 2006.
[11] K.-H. Lee, K.-Y. Whang, W.-S. Han, and M.-S. Kim. Structural consistency: enabling xml keyword search to eliminate spurious results consistently. *VLDB J.*, 19(4), 2010.
[12] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *CoRR*, 2009.
[13] G. Li, J. Feng, J. Wang, and L. Zhou. Effective keyword search for valuable lcas over xml documents. In *CIKM*, 2007.
[14] G. Li, C. Li, J. Feng, and L. Zhou. Sail: Structure-aware indexing for effective and progressive top-k keyword search over xml documents. *Inf. Sci.*, 179(21), 2009.
[15] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD*, 2007.
[16] Z. Liu, P. Sun, and Y. Chen. Structured search result differentiation. *PVLDB*, 2009.
[17] I. Muslea. Machine learning for online query relaxation. In *KDD*, 2004.
[18] I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, 2005.
[19] U. Nambiar and S. Kambhampati. Answering imprecise queries over autonomous web databases. In *ICDE*, 2006.
[20] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.
[21] A. Schmidt, M. L. Kersten, and M. Windhouwer. Querying xml documents made easy: Nearest concept queries. In *ICDE*, 2001.
[22] Y. Tao, S. Papadopoulos, C. Sheng, and K. Stefanidis. Nearest keyword search in xml documents. In *SIGMOD*, 2011.
[23] V. Vesper. http://www.mtsu.edu/vvesper/dewey.html.
[24] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, 2005.
[25] Y. Zeng, Z. Bao, T. W. Ling, and G. Li. Removing the mismatch headache in xml keyword search. In *SIGIR*, pages 1109–1110, 2013. http://xclear.comp.nus.edu.sg.
[26] W. V. Zhang, X. He, B. Rey, and R. Jones. Query rewriting using active learning for sponsored search. In *SIGIR*, 2007.