XClean: Providing Valid Spelling Suggestions for XML Keyword Queries

Yifei Lu† Wei Wang† Jianxin Li ‡ Chengfei Liu‡

†*University of New South Wales*
{yifeil, weiw}@cse.unsw.edu.au

‡*Swinburne University of Technology*
{jianxinli, cliu}@swin.edu.au

***Abstract*—An important facility to aid keyword search on XML data is suggesting alternative queries when user queries contain typographical errors. Query suggestion thus can improve users' search experience by avoiding returning empty result or results of poor qualities.**

In this paper, we study the problem of effectively and efficiently providing quality query suggestions for keyword queries on an XML document. We illustrate certain biases in previous work and propose a principled and general framework, XClean, based on the state-of-the-art language model. Compared with previous methods, XClean can accommodate different error models and XML keyword query semantics without losing rigor. Algorithms have been developed that compute the top-k suggestions efficiently. We performed an extensive experiment study using two large-scale real datasets. The experiment results demonstrate the effectiveness and efficiency of the proposed methods.

I. Introduction

Keyword search on structured and semi-structured data has attracted much research interest, as it enables common users to retrieve information from such structured data sources without the need to learn sophisticated query languages (such as SQL or XQuery) or database structure [1].

The typical process for a user to issue a keyword query involves (1) formulating a keyword query that expresses her information need, and (2) typing the query into the system. Both steps are susceptible to several kinds of errors (See Example 1), which results in a dirty query. A dirty query is likely to return empty result or results with low quality. *Query cleaning* has been proposed to alleviate this problem by suggesting several alternative queries that better reflect the user's search intention. It has already been widely used in practice for text documents, e.g., Google's "Did you mean" feature, and recently studied in the context of relational databases [2].

***Example** 1:* Consider a user who wants to find publications by *hinrich schütze* on *geo-tagging* in a bibliography database. She might formulate a query `hinrich schutze geo-tagging`, or `hinrich schuetze geo-tagging` if she is not aware of or cannot input `ü`. She might also make typographical errors by inputting `himrich` rather than `hinrich`. Finally, the database contents might contain error such that `geo-tagging` was misspelt as `geo-taging`. [1] Should any of the above cases happen, it is expected that the query result will be empty.

An intelligent query cleaning system will suggest the clean query `hinrich schütze geo-tagging`, or `hinrich schütze geo-taging` if the paper title was indexed in misspelt form in the database. As a result, the user can locate the paper she wants.

Query cleaning, or alternative query suggestion, has already become an indispensable features for Web search engines due to the relatively high percentage of dirty queries issued by users. For example, it was reported that 10–15% queries submitted to a major search engine were misspelt [3]. Another demand comes from the existence of errors in the database contents, especially if the database contains heterogeneous contents, or is automatically gleaned from the Web [4].

In this paper, we consider query cleaning for XML keyword queries. Existing query cleaning methods are designed only for Web queries [3], [5], [6] or relational databases [2], [7]. In addition, they all suffer from the following weaknesses:

- *There is no guarantee of result quality for suggested queries*. Existing work does not even guarantee that the suggested alternative queries have *non-empty* results in the database. It is obvious that users will find it frustrating if the suggested query does not return any result (either). More generally, users will expect the suggest queries have high quality results in the database.
- *Candidate queries are scored by heuristics*. Many factors need to be considered when determining the best alternative query. Existing work usually relies on a heuristically-chosen function to combine these factors, which might lead to inherent biases (See an example in Sections VIII).
- *Query cleaning is performed (largely) independent of the database contents*. Most work for correcting Web search queries relies *solely* on the query log [3], [5]. Therefore, the quality of the suggestions heavily depends on the

[1]This particular error does not happen in the DBLP dataset, which is maintained almost manually. However, a real example of such nature is paper titles containing `verication` rather than the correct word `verification`. E.g., http://www.informatik.uni-trier.de/~ley/db/conf/vveis/vveis2004.html#FerrariGMRTT04.

quality and quantity of past queries. It is also inherently biased towards popular queries – it is not uncommon that a rare word in a correct query may be corrected to a similar word that appears more often in the log. For example, `TIgE serum` will be corrected to `Tigi serum` even for an immunology database.

We propose a novel XML keyword query cleaning framework, XClean, that addresses these issues. The key idea is to assign each candidate suggestion a score based on the *overall* qualities of its query results in the database. This intuitive idea immediately guarantees non-empty query results for suggested alternative query. To implement this idea in a principled manner, we derive our scoring function based on the probabilistic theory and state-of-the-art language models. We also develop efficient query processing algorithms to find top-k alternative queries. Our extensive experiments with two large-scale real datasets demonstrated the effectiveness and the efficiency of our proposed approach.

Our contributions can be summarized as:

- To the best of our knowledge, we are the first to study the query cleaning problem for XML keyword queries.
- We propose a novel probabilistic framework for query cleaning for XML documents. Our framework guarantees the result quality of suggested queries and is general enough to incorporate different query result semantics. (Section IV)
- We propose an efficient algorithm to find high quality alternative queries under our model. The algorithm makes intelligent use of inverted index merging and skipping, and also employs probabilistic pruning. (Section V)
- We have carefully designed and performed extensive experiments with two large-scale datasets (DBLP and part of Wikipedia) and compared with existing approaches as well as two major search engines. The results clearly demonstrated the merits of taking into consideration the quality of query results. (Section VII)

The rest of the paper is organized as follows: We begin with a discussion of the problems in previous work in Section II. Section III provides the preliminaries needed for our derivation. Then we formally define the XML query cleaning problem and derive our scoring method in Section IV. Efficient query processing algorithms are presented in Section V. We offer discussions in Section VI. In Section VII, we present the experimental result and our analysis. Section VIII discusses the related work. Section IX concludes the paper.

II. Scoring Bias in Previous Work

PY08 [2] is the most relevant work to ours. It tackles a similar problem in relational databases. The authors proposed a scoring function that combines the spelling error penalty and TF/IDF scores of query segments in a heuristic manner (see [2] for further technical details).

For a candidate query suggestion $C = \{ w_1, \ldots, w_l \}$, the score of C in PY08 is given by

$$score(C) = \sum_{w \in C} score_{IR}(w) \cdot f(w)$$
$$score_{IR}(w) = \max \{ tfidf(w, t) : t \in DB \}$$

where $f(w)$ is a fixed score for a given w and not concerned here, t is a tuple in the database, and $tfidf(w, t)$ is the TF/IDF score of the word w in the tuple t.

The basic form of $tfidf(w, t)$ is

$$tfidf(w, t) = \frac{count(w, t)}{|t|} \cdot \log \frac{N}{df(w)}$$

where $count(w, t)$ is the word count of w in the tuple t,$|t|$ is the number of words in tuple t, N is the total number of tuples, and $df(w)$ is the document frequency, namely the number of tuples containing w.

There are two fundamental problems in their scoring function.

- **Bias towards rare tokens**. Given other values fixed, a smaller $df(w)$ leads to a higher $score(w)$ hence a higher $score(C)$ if $w \in C$. This means their scoring function prefers rare words to popular words, which is counter-intuitive. The ultimate reason of this bias is that it is only meaningful to compare $tfidf$ scores between different tuples for a fixed keyword, but not between different keywords.
- **Ignoring Connectivity**. Since each $score_{IR}(w)$ is maximized by its own, the words of a candidate query are not required to be connected in some way. Their scoring function finds the best correction (in terms of their scores) to each query keyword individually.

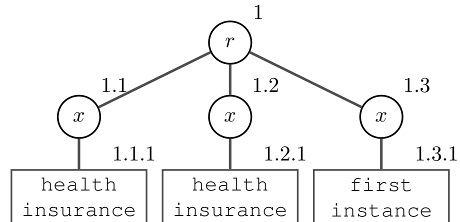


Fig. 1. Bias (Query = "`health insuance`")

We can illustrate the two problems using a simplified example in Figure 1, where the user's query is "`health insuance`", and both `insurance` and `instance` are candidate corrections for `insuance`. PY08 will incorrectly suggest "`health instance` " while the more reasonable suggestion is "`health insurance` ".

The first problem makes `instance` preferable to `insurance`, since `instance` occurs less frequently than `insurance`. The second problem allows `instance` to be selected even though it only connects with `health` through the root node.[2]

[2]Since any two nodes in an XML document can be connected through the root node, we actually do not think they are connected in a meaningful way.

Here we ignored the segmentation factor in their algorithm. However, the two problems persist even if segmentation is considered.

These problems are fully addressed in our framework.

III. Preliminaries

We model an XML document as a rooted, node labeled, ordered tree T; all the XML elements are nodes in T; there is an edge between two nodes in T if and only if they have a parent-child relationship in the XML document. For brevity, we treat attribute nodes and PCDATA as element nodes and ignore other types of nodes.[3] Only leave nodes in the tree are associated with text contents. For a collection of XML documents, we add a virtual root node that connects to the roots of all the individual XML documents to form a single tree. The depth of the tree is denoted as d, and the root node has a depth of 1.

The text contents of all XML elements are tokenized into a set of *tokens* by white spaces and punctuations. These tokens collectively form the *vocabulary* $\mathcal{V}$.

For each node n in the tree, we can describe its *label path* as the concatenation of the labels of nodes on the path between the root of the tree and the node n. The label paths can be viewed as *node types*. Two distinct nodes may have the same label path, indicating that they contain the same sort of information.

Dewey encoding assigns a unique code to each node in the tree. Each node has a distinct number among its siblings. The Dewey code of a node is the concatenation of the numbers on the path from the root to that node, separated by a separator (denoted as "`.`"). There are two kinds of partial order we can define on Dewey codes. x smaller than y, denoted by $x \prec y$, if x precedes y in the document order. $x \prec_{\text{AD}} y$ if x is an ancestor of y in the XML tree. The $\prec$ orders can be tested by comparing the Dewey codes in dictionary order. The $\prec_{\text{AD}}$ order can be determined by checking if the Dewey code of one node is a prefix of the other. The time complexity of both operations are $O(d)$.

A *keyword query* for XML documents consists of a sequence of tokens. Each token may or may not be in the vocabulary. In this paper, we focus on typographical errors and hence assume mistyped words are transformed from a valid token in the vocabulary by a few edit operations (insertion, deletion, and substitution). [4] The minimum number of edit operations needed to transform s to t is the *edit distance* of s and t, denoted by $ed(s,t)$.

IV. An XML Query Cleaning Framework

In this section, we develop a novel query cleaning framework based on statistical language modeling. We show how to accommodate various ranking factors in a rigorous manner.

[3]It is however easy to incorporate other types of nodes.

[4]This is a common setting for query corrections [3]. However, our method can easily handle other types of errors.

A. Problem Definition

Given an *observed* query Q issued to the XML document T from the user, we consider a set of possible candidate queries C_i. We use $P(C_i|Q,T)$ to denote the probability that C_i is intended while the user issues Q on the XML document T (called **Candidate Query Probability**).

Naturally, we will present to the user the k candidate queries that have the highest probabilities.[5] Hence, our XML query cleaning problem can then be formally defined as

Definition 1: Given an XML document T and a keyword query Q consisting of l query keywords $q_1, q_2, \ldots, q_l$, we return the top-k candidate queries C_i ($1 \le i \le k$) with the highest $P(C_i|Q,T)$

We choose to define the problem in a probabilistic setting (rather than, for example, resorting to heuristic-based scoring functions) because probability-based model is a proven technique in text recognition and spell correction, and has achieved good performance in those tasks.

We follow previous studies on query cleaning [3], [2] and generate the candidate queries as follows:

1) For each query keyword q_i, we first generate a list of *variants* [6], $var_\epsilon(q_i)$, each of which is a term in the vocabulary and has no more than ϵ edit errors from q_i.
2) The candidate query space (denoted as $\mathcal{C}$) is then the Cartesian product of the lists, i.e., $\mathcal{C} = var_\epsilon(q_1) \times var_\epsilon(q_2) \times \ldots \times var_\epsilon(q_l)$.

***Example** 2:* Consider the sample XML tree and query in Figure 2. We obtain three variants for each query keyword ($\epsilon = 1$). Hence the search space consists of 6 candidate queries (i.e., `tree icdt`, `tree icde`, `trees icdt`, ...).

q_i	$var_\epsilon(q_i)$
`tree`	`tree`, `trees`, `trie`
`icdt`	`icdt`, `icde`

B. Decomposing the Candidate Query Probability

Based on the Bayes Theorem, we have

$$P(C_i|Q,T) = \frac{P(Q|C_i,T) \cdot P(C_i|T)}{P(Q|T)} \quad (1)$$

$$= \kappa \cdot P(Q|C_i,T) \cdot P(C_i|T) \quad (2)$$

Since $\frac{1}{P(Q|T)}$ is fixed given the query Q and document T (denoted as κ in the last step), the probability $P(C_i|Q,T)$ is proportional to two important factors:

1) The first is an error term, $P(Q|C_i,T)$, that models the likelihood of generating the observed query Q while the intended query is actually C_i. Section IV-B1 gives more details.
2) The second term, $P(C_i|T)$, is the query generation probability given the document T. Section IV-B2 gives more details.

[5]Note that Q might be identified as one of the C_i.

[6]Also known as the *confusion set* [8].

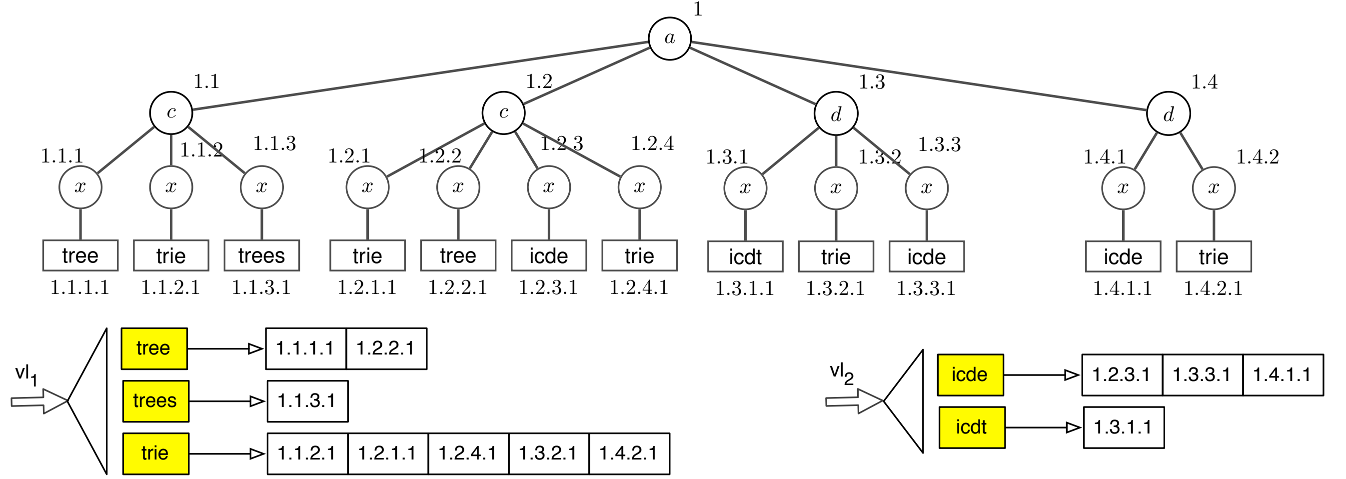


Fig. 2. An Example XML Tree (Query = `tree icdt`)

1) Modeling Typographical Errors: In Computational Linguistic literature, the *single word* error model $P(q|w)$, namely the probability of typing q when w is intended, is well studied. E.g., Mays *et al* [8] proposed an error model when allowing a single edit error (i.e., $\epsilon = 1$)

$$P(q|w) = \begin{cases} \alpha & \text{, if } q = w \\ (1-\alpha)/(|var_{\epsilon}(w)| - 1) & \text{, otherwise.} \end{cases} \quad (3)$$

That is, if the observed word q is identical to the intended word w, the $P(q|w)$ is a constant α. Otherwise, the remaining probability mass is equally distributed among the candidates in $var_{\epsilon}(q)$.

We follow the same spirit of this model and handle error threshold larger than 1 by distributing the probability mass inverse proportionally to the edit distance between q and w.

For each variant $w \in var_{\epsilon}(q)$, we define

$$P(w|q) = \frac{1}{z} \cdot \exp(-\beta \cdot ed(q, w)) \quad (4)$$

where β is an error penalty parameter and z is a normalization factor. Therefore

$$P(q|w) = \frac{1}{z'} \cdot \exp(-\beta \cdot ed(q, w)) \quad (5)$$

where $z' = \frac{P(q)}{P(w)z}$. The $P(q|w)$ decreases exponentially with the edit distance $ed(q, w)$ increases. β controls how fast it decreases. A large β value will heavily penalize edit errors.

In the experiments, we found that $\beta = 5$ achieves the best performance in most cases. The reported results all use this value.

To deal with multi-word queries, we make the independence assumption on error probabilities in each query word. That is,

$$P(Q|C_i) = \prod_{1 \le j \le l} P(q_j | C_i[j]) \quad (6)$$

where $C_i[j]$ is the j-th word in C_i.

Another assumption commonly adopted in the literature is that $P(Q|C_i, T) = P(Q|C_i)$, i.e., the probability of users making spelling errors is independent of the XML document. This assumption usually holds as the reasons users make spelling errors are commonly related to their keyboard layout and spelling habits, but not the database content.

2) Modeling Query Generation Probability: The second term in Equation (2), $P(C_i|T)$, is the probability of users issuing the query C_i when the XML document T is presented. This is a typical problem in Information Retrieval, where $P(Q|D)$ is used to measure the relevance of a document D to the user query Q. *Language modeling* is a proven approach to this problem [9].

The state-of-the-art language modeling approach uses the *unigram language model* and *dirichlet smoothing* as follows:

$$p(Q|D) = \prod_{w \in Q} p(w|D), \quad \text{where}$$

$$p(w|D) = \frac{count(w, D) + \mu p(w|B)}{|D| + \mu}$$

where B is the background model that represents the whole collection of documents, $p(w|B)$ is the probability that token w appears in B, and μ is a smoothing parameter to assign a non-zero probability to tokens that does not appear in D [9].

The standard language model was proposed for unstructured text documents, hence not aware of the structural information/demand in keyword queries on semi-structured XML data. XML documents provides an opportunity to identify the parts of documents that are relevant to the query in finer granularity. Next we present an approach to utilize the tree structure for better modeling of XML documents.

Much work in the database literature is based on finding *subtrees* that contain at least one instance for each query keyword. In the same spirit, we view an XML document T as a collection of disjoint subtrees, named *entities*. They are deemed the basic units of information that may interest users. Each entity is uniquely denoted by its root node (r). This model is able to immediately accommodate several XML keyword query semantics. In this paper, we focus on the

specific node type semantics [10], which defines entities by the most probable type of result nodes p_Q for each query Q. Conceptually, all the result nodes satisfying p_Q are the roots of entities for Q. Note that the entity definition depends on the query, which means we need to infer result types for each candidate query.

For a particular query C_i and a label path p, we measure how likely p is the desired result type by the following formula, as suggested by [10].

$$U(C_i, p) = \log{(1 + \prod_{w \in C_i} f_w^p)} \cdot r^{depth(p)} \tag{7}$$

where w is a keyword in C_i; f_w^p is the number of nodes whose label path is p and contains w in its subtree; r is a factor that reduce the score for deep label paths. The intuition behind this formula is that, users like the popular types of nodes that contain all query keywords, but not those too deep in the tree that contain little additional information other than the given keywords.

***Example** 3:* Consider again the sample XML tree in Figure 2. The search space of the query is illustrated in Figure 2. Consider the candidate query C = "`trie icde`" and four node types p_1 = `/a/c`, p_2 = `/a/c/x`, p_3 = `/a/d` and p_4 = `/a/d/x`. Counting the number of nodes of each type that contain the tokens in the subtree, we have $f_{trie}^{p_1} = 2, f_{trie}^{p_2} = 3, f_{trie}^{p_3} = f_{trie}^{p_4} = 2, f_{icde}^{p_1} = f_{icde}^{p_2} = 1, f_{icde}^{p_3} = f_{icde}^{p_4} = 2$. Therefore

$$\begin{aligned} U(C, p_1) &= \log{(1 + 2 \times 1)} \cdot r^2 \\ U(C, p_2) &= \log{(1 + 3 \times 1)} \cdot r^3 \\ U(C, p_3) &= \log{(1 + 2 \times 2)} \cdot r^2 \\ U(C, p_4) &= \log{(1 + 2 \times 2)} \cdot r^3 \end{aligned}$$

If $r = 0.8$, $U(C, p_3)$ will be the largest. Then the best result type is p_3 and the XML tree would be decomposed into subtrees of type `/a/d`.

Given a decomposition of tree T into a set of N entities $\{ r_j \}$, we have

$$P(C_i|T) = \sum_{j=1}^{N} P(C_i|r_j) \cdot P(r_j|T) \tag{8}$$

The second term in the above equation can be deemed as the prior probability of each entity r_j [11]. For simplicity, we consider a uniform prior here (i.e., $P(r_j|T) = 1/N$). Note that this can be easily generalized to non-uniform priors if additional data or domain knowledge is available (e.g., query logs).

In order to estimate the first term, $P(C_i|r_j)$, we form virtual documents $D(r_j)$ by concatenating the contents of all the descendant nodes under r_j into a plain document. This allows us to apply the unigram language model as:

$$P(C_i|r) = \prod_{w \in C_i} P(w|D(r_j)) \tag{9}$$

3) The Final Formula: By putting the above derivations together, the candidate query probability with a uniform prior can be calculated as:

$$P(C_i|Q,T) = \kappa \cdot P(Q|C_i) \cdot \left(\frac{1}{N} \sum_{j=1}^{N} \prod_{w \in C_i} P(w|r_j) \right) \tag{10}$$

***Example** 4:* Assume the input query is Q = "`tree icdt`". As computed in Example 3, the best result type for candidate query C = "`trie icde`" is `/a/d`. Thus there are two entities for C, namely 1.3 and 1.4. The score of C can be computed by

$$\begin{aligned} P(C|Q,T) &= \kappa \cdot P(Q|C) \cdot P(C|T) \\ P(C|T) &= \frac{1}{2} P(\texttt{trie}|D(1.3)) P(\texttt{icde}|D(1.3)) \\ &+ \frac{1}{2} P(\texttt{trie}|D(1.4)) P(\texttt{icde}|D(1.4)) \end{aligned}$$

V. Query Processing Algorithm

In this section, we discuss an efficient algorithm to compute top-k query suggestions. A naïve algorithm is to enumerate all candidate queries and score them one by one. This is obviously inefficient as there will be repeated visits of relevant inverted lists while processing candidate queries.

Instead, we first design an efficient algorithm that simultaneously compute the scores of all candidate queries by *one pass* over the relevant inverted lists, hence achieving optimal I/O complexity in the worst case. Our algorithm also employs several skipping techniques, which effectively reduce the I/O cost in many practical situations (as well as CPU costs). Then we discuss a probabilistic pruning method to control the number of candidates evaluated in order to reduce the time and space complexities.

A. Generating Variants for Query Keywords

The first step in the query processing is to efficiently generate $var_\epsilon(q_i)$. Since most of the tokens are pretty short, we employ a partitioned version of the FastSS method [12], [13], which is one of the fastest approximate string matching method under edit distance constraints. The method builds a special index with a space complexity of $O(\min(l^\epsilon, \epsilon^2) \cdot |\mathcal{V}|)$ (where l is the average length of tokens in $\mathcal{V}$), and it can compute $var_\epsilon(q_i)$ typically in $O(\min(|q_i|^\epsilon, \epsilon^2) + |var_\epsilon(q_i)|)$ time.

The basic idea is that if two tokens are similar, after deleting a few characters from each of them, they will be the same. Specifically, given a word, we can generate a set of tokens called *deletion neighborhood* by deleting up to ϵ characters from that word. Two words are within edit distance of ϵ only if their ϵ-deletion neighborhoods have non-empty intersection. Partitioning can be employed to handle long tokens without incurring exponential growth of the ϵ-deletion neighborhood.

We build an index for the deletion neighborhoods of all tokens in the vocabulary $\mathcal{V}$ offline. In the query time, for each query keyword $q_i \in Q$, we generate tokens in its ϵ-deletion neighborhood and probe the index to find a set of candidate

words that share at least one deletion neighbor with q_i. Edit distance computation is applied to each candidate to obtain the true ϵ-variants of q_i.

The space complexity of the index is $O(\min(l^\epsilon, \epsilon^2 l_p) \cdot |\mathcal{V}|)$, where l_p is a parameter that can be tuned to strike a good balance between space and time costs. The query processing time is $O(\min(l^\epsilon, l_p\epsilon^2) \cdot size_{cand})$, where $size_{cand}$ is the average length of the inverted lists in the index, which was shown to be $O(1)$ for several real datasets.

B. Finding Result Types for Candidate Queries

In order to efficiently find the best result type for each candidate query, we use an index that, for each keyword w, returns a list P_w of types and their f_w^p values. Then the best result type for query C can be obtained by intersecting the lists $\bigcap_{w \in C} P_w$ and applying the formula (7). We denote this algorithm as $\mathsf{FindResultType}(C)$, whose input is a candidate query and output is the best result type.

Since typically the number of different types is small, this process can be very efficient. However, considering the large number of candidate queries, we cannot afford the result type computation for all candidate queries. A key observation is that some tokens may only be connected through the root node, thus are unlikely to be an interesting query (note that every token is connected to the root). We utilize this fact by delaying the result type computation until we have evidence that a candidate query is promising. Specifically, we define a *minimal depth threshold* d. We do not consider the types of depth smaller than d when computing the best result types. Moreover, we do not start the result type computation for a candidate query until we find it has at least one lowest common ancestor (LCA) node at the depth of d or deeper. In the experiments, we found that a large portion of the candidate query space is unpromising, and $d = 2$ is usually enough to prune them without affecting the suggestion quality.

C. The XClean Algorithm

Next we present the main algorithm XClean as well as the data structures it uses.

In order to efficiently locate the entities and compute the score of each candidate query, we build inverted index that maps each token to a list of tree nodes (sorted in the document order) that contain the token; each entry n is represented as a tuple of $\langle \mathsf{dewey}, \mathsf{lp}, \mathsf{tf} \rangle$, namely the Dewey code of the tree node, its label path and the frequency of the token in that node. For brevity we use n.w to denote the token associated with the entry.

Given a list of variants for a query keyword, we use an abstract data structure named MergedList to organize the inverted lists of all the variants as if they have been physically merged as a single sorted inverted list. MergedList also supports several functions. The cur_pos() function returns the *head* (i.e. the first node) of the merged list, the next() operation returns the head then remove it from the list. The skip_to(dewey) operation discards all the nodes whose Dewey code is smaller than dewey, and returns the first node that is equal to or larger than dewey. We implement the MergedList by a min heap of the heads of its member inverted lists. The root of the heap is the head of the merged list. Each time next() is invoked, the root of the heap is popped out and the next node of the corresponding inverted list is put into the heap unless no more node is available. The skip_to(dewey) function is implemented by performing a skipping via *binary search* or *exponential search* in each member inverted list; then, it rebuilds the min heap and proceeds as next().

Intuitively, the main algorithm works by first collecting the nodes in each subtree identified by the minimal depth d, then enumerating candidate queries in that subtree. For each candidate query, XClean finds its best result type as discussed in the previous section. Then the entity nodes defined by the best result type can be identified and the probability scores be computed and accumulated.

The detailed algorithm is shown in Algorithm 1. The input consists of a query Q and a minimal depth d specifying the minimal depth of subtree roots that can be return nodes. The algorithm first generates the variants of each query keyword according to Section V-A, then initialize the MergedList data structures as discussed above (Lines 2–3). Next, the algorithm repeats finding an *anchor* node, that is, the node with the largest dewey code among the current heads of each MergedList(Lines 4, 5, 16), until all nodes are exhausted. The anchor node is denoted by t_z, and $z \in \{1 \ldots l\}$ is the index of the MergedList from which t_z is selected. If any of the vl_i.cur_pos() returns **nil**, t_z is **nil**.

The dewey code of an anchor node is truncated to the depth of d, and denoted by g.dewey, which acts like the target to which all the inverted lists are aligned (Line 7–8). The occurrences of all variants in the subtree of g.dewey is collected (Line 9–11). Then we enumerate all the candidate queries that consist of the variants observed in g.dewey. For each candidate query, we can apply the FindResultType algorithm to determine the best result type p_C (Line 13). Once this is done, the XML document is essentially decomposed into entities of type p_C and we can calculate the part of candidate query score in the subtree of g according to formula (9) (Line 14–15). Scores are accumulated in the global hash table S.

***Example** 5:* Consider the example XML tree and query in Figure 2. The variants, their inverted lists, and their MergedLists are shown in Figure 2. Let the minimal depth $d = 2$ and reduction rate $r = 0.8$.

In the first iteration, the anchor t_z is node 1.2.3.1. After truncating it to depth d, we have a target of 1.2. Then the inverted lists of `tree`, `trees` and `trie` will be skipped to nodes no smaller than 1.2, i.e. 1.2.2.1, **nil** and 1.2.1.1 respectively. The leaf nodes in the subtree of 1.1 are all skipped.

At the subtree of 1.2, we find tokens `trie`, `tree` and `icde`. They can form two candidate queries C_1 = " `trie icde`" and C_2 = "`tree icde`". For C_1, its best result type is /a/d, while for C_2 it is /a/c. Therefore the node 1.2 is an entity for C_2 but not for C_1. The scores of C_2 in this entity

Algorithm 1: XClean (Q, d)

```
Data: Q is the input query; d is the minimal depth
      threshold. S is a hash table that maintains the
      scores of all candidate queries; P is a hash table
      that records the best result type for each candidate
      query.
1  for each q_i ∈ Q do
2  |  var_ε(q_i) ← makeVariants(q_i);
3  |  vl_i ← MergedList(var_ε(q_i));
4  end for
5  t_z ← max{ vl_i.cur_pos() | i ∈ {1...l} };
6  while t_z ≠ nil do
7  |  g ← truncate t_z.dewey by depth d;
8  |  for each i ∈ {1...l} do
   |  |  /* discard all the nodes smaller
   |  |     than g                              */
9  |  |  t_i ← vl_i.skip_to(g);
   |  |  /* collect the descendant nodes of
   |  |     g to a hash table h                 */
10 |  |  while t_i ≠ nil & g ≺_AD t_i.dewey do
11 |  |  |  h[t_i.w] ← h[t_i.w] ∪ t_i;
12 |  |  |  t_i ← vl_i.next();
13 |  |  end while
14 |  end for
15 |  for each candidate query C enumerated from the
   |  occurrences of variants in h do
16 |  |  if P[C] = nil then
17 |  |  |  P[C] ← FindResultType(C);
18 |  |  end if
19 |  |  for each node r of type P[C], g ≺_AD r and
   |  |  contains at least one instance of each
   |  |  keyword w ∈ C do
20 |  |  |  S[C] ← S[C] + calc_score(C);
21 |  |  end for
22 |  end for
23 |  t_z ← max{ t_i | i ∈ {1...l} };
24 end while
25 return the top-k alternative queries in S
```

is calculated and accumulated.

The next anchor would be 1.3.2.1, which is truncated by depth d to 1.3. The tokens found in the subtree of 1.3 are `icdt`, `trie` and `icde`. Again we can form two queries C_1 and C_3 = "`trie icdt`". Since the best result types of C_1 and C_3 are exactly `/a/d`, we can now calculate their scores based on the frequencies counted in the subtree.

Complexity Analysis. Let $L_\Sigma = \sum_{i=1}^{l} |vl_i|$, $L_{\max} = \max_{i=1}^{l} |vl_i|$, $L_{\min} = \min_{i=1}^{l} |vl_i|$, $m = \max_{i=1}^{l} |var_\epsilon(q_i)|$ and $\mathcal{C}_{\text{eff}}$ be the set of effective candidate queries encountered in the algorithm.

Since the algorithm reads every node at most once from the inverted index, it achieves the optimal I/O complexity of $O(L_\Sigma)$. The CPU cost can be shown as $O(L_{\min} \cdot (l \cdot m \cdot \log L_{\max} + |\mathcal{C}_{\text{eff}}|))$.

In practice, the runtime efficiency of the algorithm is much better than the above worst-case analysis due to two reasons: (1) the use of anchor nodes and the skippings avoids accessing many non-promising nodes in the inverted lists. (2) $|\mathcal{C}_{\text{eff}}|$ is much smaller than the worst case number $O(m^l)$ as many of the variant combinations do not connect at levels deeper than the minimal depth threshold. As we'll discuss shortly, in practice, we can limit $|\mathcal{C}_{\text{eff}}|$ to a constant without affecting the quality of the suggestion.

D. Probabilistic Candidate Query Pruning

When the size of the dataset grows, the number of effective candidate queries will increase too. This not only slows down the algorithm, but also incurs high memory usage. Previous approaches alleviate this problem by using heuristics to aggressively prune candidates (even before the computation) or making additional assumptions (to reduce the search space). Here, we present another pruning method based on pruning candidate queries on-the-fly and in a probabilistic manner. The idea is that when we have to prune candidates, we shall select candidates whose final aggregate score is unlikely to be among the top-k choices. We can perform the estimate based on their score accumulated so far.

More specifically, given an alternative query C_i, we use X_i^j to represent the score of $r_j \in T$ $(1 \leq j \leq N)$. Assume that we have processed n r_js at the time of pruning. Since we are using a uniform prior for $P(r|T)$, we are essentially calculating the average of $P(C_i|r_j)$ $(1 \leq j \leq N)$. We can use the sample mean to estimate actual mean value. Formally, let V be the true mean value and $\hat{V} = \frac{1}{n}\sum_{1 \leq j \leq n} X_i^j$ be the mean based on n samples. Hoeffding's inequality states that

$$P(|\hat{V} - V| \leq \varepsilon) \geq 1 - 2e^{-2n\varepsilon^2}$$

Therefore, we can employ the following probabilistic candidate query pruning method. We can set a parameter γ as the maximum number of candidate queries whose scores are accumulated in the memory. When we have new candidate queries yet all the γ memory accumulators are in use, we will have to choose a victim. We pick the candidate query whose estimated score (where $P(C_i|T)$ part is calculated using currently observed samples) is the lowest.

VI. Discussions

A. Extension of Our Framework

There is another kind of typographical errors that is due to the insertion or deletion of spaces or hyphens (e.g., `power point`). This kind of errors will change the number of keywords in the query. To include these errors in our framework, we assume there are at most τ changes of spaces (either insertion or deletion of spaces). We enumerate all the possibilities of space changes in the input query, and validate them by ensuring the tokens resulted from the space changes are in the vocabulary. The candidate query space is then expanded by including these new tokens. Since typically the query length and τ are not large, and most of the space

changes result in invalid tokens, the extra time cost will still be acceptable.

Besides typographical errors, our framework can be easily extended to include cognitive errors by properly define the variant set $var(q)$ and the probability $P(q|w)$ (e.g., soundex, WordNet semantic distance [7], and synonyms by using a thesaurus or an ontology). It is also possible to combine different types of errors by learning their relative weight from query logs. This is left for future work.

B. Our Framework with the SLCA Semantics

Another feature of our framework is that it is able to accommodate different XML keyword query semantics by using different ways to decompose the XML document tree into entities. In the interest of space, we focus on specific return node type semantics [10]. In the longer version of this paper, we also consider other semantics. Here, we briefly introduce how to integrate the SLCA semantics [14] in our framework.

To adopt the SLCA semantics, we consider each candidate query individually. We deem the SLCA nodes of a candidate query as its roots of entities. Thus each candidate query corresponds to its own set of entities. We then use these entities as r_is in Equation (8) to score each candidate query.

We have also designed an efficient algorithm to find top-k query suggestions in SLCA semantics. The algorithm is adapted from the multi-way SLCA algorithm [15] by new skipping criteria suitable for our problem. Similar to the XClean algorithm, it only needs to access the relevant inverted lists once.

We have also experimented with the SLCA semantics. It works equally well on the DBLP dataset (which is data-centric), but less well on the INEX dataset (which is document-centric).

VII. Experiments

We present and analyze our experimental results in this section.

A. Datasets and Queries

We use two large-scale real datasets in our experiments:

INEX is the 2008 Wikipedia document collection from INEX [16]. It contains about $660,000$ XML files with a total size of 5.8GB. We form a single XML document by adding a virtual root.

DBLP is a snapshot of the DBLP bibliography database in May 2009.[8] It contains a single XML file of 526MB.

Note that these two datasets have quite different characteristics and were chosen to represent typical document-centric (INEX) and data centric (DBLP) XML data, respectively.

We build indices for the two datasets as described in Section V-B and V-C. Stop words, numbers and short tokens (less than three characters) are not indexed. The index sizes of the INEX and DBLP datasets are 1.8GB and 400MB, respectively. Some statistics about the datasets are shown in Table I.

Dataset	#size (MB)	#node	max depth	avg depth
INEX	5,878	52M	50	5.58
DBLP	526	12M	7	3.8

TABLE I
Dataset Statistics

It is acknowledged that finding the ground truth for query cleaning research is quite difficult [3]. We carefully designed the following three-step procedure to obtain "dirty queries" and the ground truth. We first design a set of *initial queries*, and then use two methods to obtain *dirty queries* from them; finally we employ human assessors to manually mark the *ground truth*.

INEX-QUERY We downloaded the official topics of INEX 2008[9]. We extracted the contents from the "`title`" element of the topics as our keyword queries. There are in total 285 queries with query length ranging from 1 to 7. The average query length is 2.5.

DBLP-QUERY We hand-picked 49 queries for the DBLP dataset. We pick queries by looking at the descriptions of all the ACM Fellows elected in 2008[10], and compose a query consisting of (1) the last name of the person, and (2) some keywords mentioned in his or her contribution areas. For example, the query we formulated for *Jonathan S. Rose* is `rose architecture fpga`.

In order to obtain dirty queries from the above initial query sets, we design two methods to simulate possible spelling errors.

RAND We apply random edit operations (i.e., insertion, deletion, and substitution) to each keyword in an initial query to obtain the corresponding dirty query. There are two technical subtleties: (1) we make sure that the resulting tokens after injecting random edit operations do *not* belong to the vocabulary. This minimizes the risk that the edited query is still a reasonable and clean query. (2) we do not introduce random edit operations to very short tokens (in our experiment, those whose length is no larger than 4). This is to ensure enough information is preserved such that it is still feasible for an intelligent query cleaning algorithm to recover the clean query.

RULE We make use of the list of common misspellings occurring at the Wikipedia site.[11]. The list is edited by humans and is also used by the spell checker Aspell [12]. Given an initial query, we look up each token in the list and replace the token with one of its misspelt forms.

[7]http://wordnet.princeton.edu/
[8]http://www.informatik.uni-trier.de/~ley/db/
[9]http://www.inex.otago.ac.nz/data/documentcollection.asp
[10]http://fellows.acm.org/
[11]http://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines
[12]http://aspell.net/test/common-all/

Note that some of these misspellings are distant from the correct form, hence we need to explore a larger space of variants thereby candidate queries than the RAND ones.

To obtain the ground truth, we employ two human assessors to choose the best suggestions independently. We then take the *union* of their judgement as the golden standard, i.e., if two assessors chose different suggestions, we will take both as the correct suggestions. The two assessor agree on 50% of the queries.

Astute readers might wonder why we do not use the initial queries as the ground truth. The reason is that manual labeling helps us to find other valid alternative queries. Sometimes, it even finds alternative queries that we think are superior than the initial queries. For example, we found two such instances in the INEX query set: `european cities skycrapers higher meters` and `famous bouddhist places`, which both contain misspellings in the initial query.

Finally, by applying the above two methods to the two initial query sets, we can obtain four dirty (also called *negative*) query sets. We also include the initial query sets as *positive* query sets (named as **INEX-Clean** and **DBLP-Clean** respectively). Some sample queries are shown in Table II.

Query Set	Sample Queries

INEX-CLEAN
INEX-RAND
INEX-RULE
	`great barrier reef`
	`reat barier reef`
	`gerat barrier reef`

DBLP-CLEAN
DBLP-RAND
DBLP-RULE
	`rose architecture fpga`
	`rose amrchitecture fpga`
	`rised archetecture fpga`

TABLE II
QUERY SETS AND THEIR SAMPLE QUERIES

B. Algorithms and Measures

Since there is no existing work on XML keyword query cleaning, we adapt the query cleaning algorithm in PY08 [2] to our settings and use it as the baseline of comparisions. This algorithm treats each relational tuple as an independent document, and essentially flattens the relational database into a collection of documents. In the same spirit, we adapt the algorithm to work on XML data by treating each XML element as a document.

Besides, we also tested two major search engines, denoted by SE1 and SE2. We send dirty queries to them using the `site:` feature, so that the search scope is limited to the respective domain. For example, a query `reat barier reef` will be input as `reat barier reef site:en.wikipedia.org`. Although strictly speaking we cannot directly compare our results with the search engines, since we are not aware of their settings, which may be quite different from ours, we can still get a reasonable understanding of the effectiveness of XClean system.

We take the following measurements in our experiments:

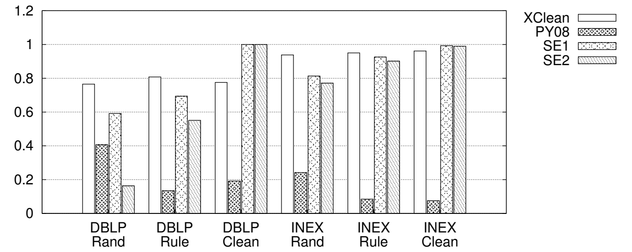


Fig. 3. MRR ($\beta = 5, \gamma = 1000$)

MRR The quality of the suggestions can be measured by its Mean Reciprocal Rank (MRR) values, which is commonly used in Information Retrieval to evaluate the effectiveness of retrieval systems that returns a ranked result. Specifically, for a dirty query Q, its reciprocal rank is $1/\text{rank}(Q_g)$, where $\text{rank}(Q_g)$ is the rank of the ground truth Q_g in the top-k list. Then for a set of queries $\mathcal{Q}$, the MRR is defined as $\text{MRR} = \frac{1}{|\mathcal{Q}|}\sum_{Q\in\mathcal{Q}}\frac{1}{\text{rank}(Q_g)}$. MRR values are within $[0, 1]$ and a larger value indicates better retrieval performance.

Precision@N It is defined as $precision@N = \frac{A}{|\mathcal{Q}|}$, where A is the number of queries whose top-N suggestions contain the ground truth. This measure gives the percentage of queries that users will be satisfied if they were presented with at most N suggestions for each query. For search engines, since they only return at most one suggestion, we can only measure precision@1 for them.

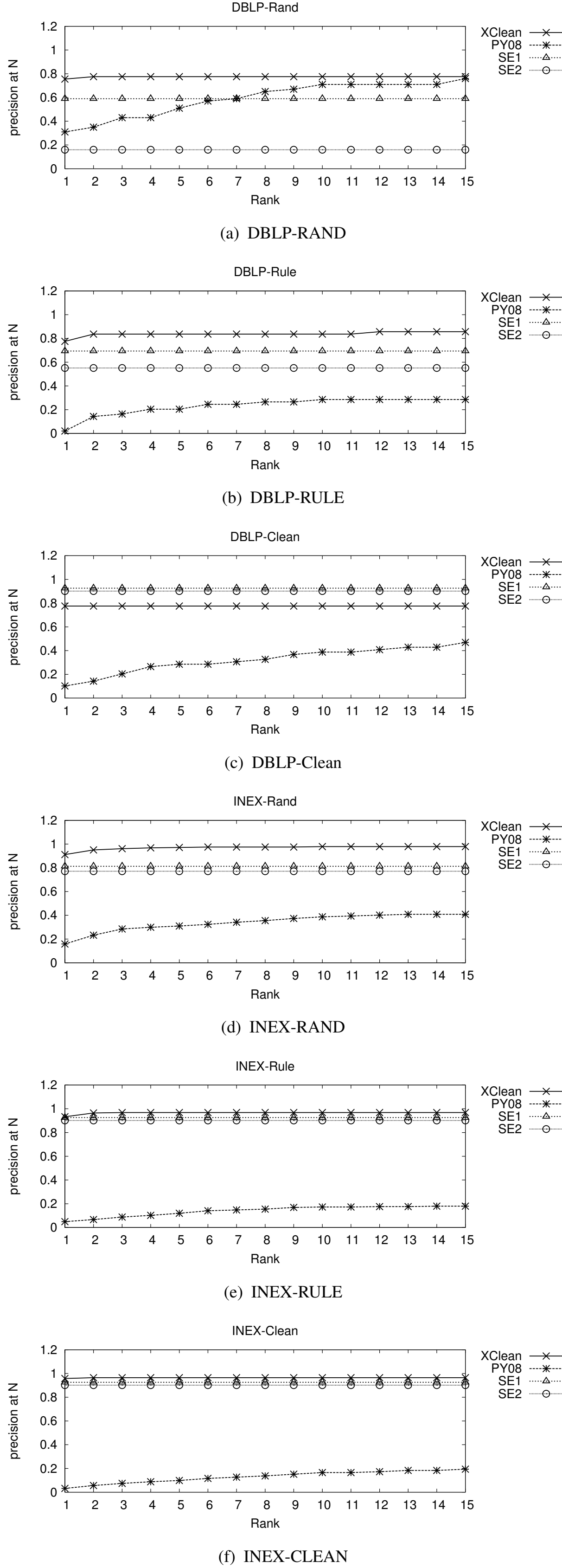
Time We record the running time of XClean and PY08 algorithms.

All experiments were carried out on a PC with Intel Xeon X3220@2.40GHz CPU and 4GB RAM. The operating system is Debian 4.1.1-21. All algorithms were implemented using JDK 1.6.

C. Effectiveness

We plot the MRR values for all systems in Figure VII-B. Note that since SEs only return at most one suggestion, their MRR values should be interpreted as lower bounds. We can make several observations based on the results.

- XClean is significantly better than PY08 on all query sets. We will give a detailed analysis shortly.
- The two search engines have better performance than XClean on the two clean query sets, which means they are highly successful in *not* suggesting any alternative queries for clean queries. Besides, they work better for dirty queries generated by rules (common misspellings by humans) than those by random edit operations. We conjecture that the above observations are likely due to their use of query logs. For example, query logs were used in several published work [3], [5], [6]. XClean achieved comparable or better results without the knowledge of query logs.
- XClean didn't achieve a close-to-1 MRR value for the DBLP query sets. We found that, in many case, the



(a) DBLP-RAND

(b) DBLP-RULE

(c) DBLP-Clean

(d) INEX-RAND

(e) INEX-RULE

(f) INEX-CLEAN

Fig. 4. Precision@N ($\beta = 5, \gamma = 1000$)

alternative queries it suggested are pretty reasonable. For example, consider the query `rose architecture fpga`. XClean suggests `based architecture fpga` first and then the original query. We believe this is a valid suggestion as many papers in DBLP contain "fpga based architecture" in their titles.

Analysis of PY08's Results. We analyze the low performance of PY08 on both datasets. An example query and suggestions are given in Table III. It is obvious that PY08 tends to suggest rare tokens (`rävel`). In addition, it does not consider if the suggested query has any result in the database (their suggested query `adventuresome rävel dairy` does not return any meaningful result).

These observations confirm our analysis in the inherent biases in PY08's scoring function (See Sections VIII).

Query	`adventurescome ravel diiry`
XClean	`adventuresome travel diary`
PY08	`adventuresome rävel dairy`
SE1/2	`adventures come travel diary` / no result/suggestion

TABLE III
SAMPLE QUERY CLEANING RESULTS (UNDERLINED WORDS IN THE QUERY CONTAIN ERRORS)

Figures 4(a)–4(f) show the Precision@N results.

- On all negative query sets, XClean achieves the highest precisions, and the curve barely improves with the increase of N. This means most of the correct suggestions are found at the top of the suggestion list.
- The curve for PY08 improves gradually with N. This is because the correct suggestion is down in the suggestion list and is only likely to be returned when N is large.

Effects of error penalty β.

XClean					
Query Set	$\beta = 0$	5	10	15	100
DBLP-Rand	**0.77**	**0.77**	**0.77**	**0.77**	**0.77**
DBLP-Rule	0.54	**0.81**	**0.81**	**0.81**	**0.81**
DBLP-Clean	0.70	**0.78**	**0.78**	**0.78**	**0.78**
INEX-Rand	0.87	**0.92**	0.91	0.91	0.91
INEX-Rule	0.48	**0.93**	0.92	0.92	0.93
INEX-Clean	0.72	0.95	**0.96**	**0.96**	**0.96**

TABLE IV
MRR VS. β ($\gamma = 1000$)

We experimented with different spelling error penalty parameter β (See Equation (5)).The results when $\gamma = 1000$ are listed in Table IV. We use bold font to highlight the best results.

As we can see from the table, $\beta = 5$ achieves the best results for almost every query set. The MRRs improve quickly from $\beta = 0$ to $\beta = 5$, then almost plateau for higher β values. In

the INEX datasets, we can even observe minor decrease in MRR values for $\beta > 5$.

Our explanation is that when β is small, we are more lenient towards spelling errors. If a misspelt word has a distant yet frequently occurring variant, there is a possibility that our algorithms will rank it higher.

Effects of In-Memory Accumulators (γ). We also evaluated the impact of the number of accumulators (γ) on the effectiveness of XClean (See Table V). The γ reflects the memory requirement of the XClean algorithm.

We can observe that the quality of the suggestions improves with the increase of γ and such improvement is more noticeable when the candidate query space is larger. We also found that 1000 accumulators is almost sufficient to achieve the best effectiveness. Note that this echoes the finding in previous studies that 50 candidates is sufficient in practice [3].

For PY08, we abuse the notation γ to denote the number of top segments that are computed for each partial query. Around $\gamma = 100$ it achieves the best effectiveness overall.

Algorithm	Query	$\gamma = 10$	100	1000	10000
XClean	**DBLP-RAND**	**0.76**	**0.76**	**0.76**	**0.76**
	DBLP-RULE	0.72	**0.81**	**0.81**	**0.81**
	DBLP-CLEAN	**0.78**	**0.78**	**0.78**	**0.78**
	INEX-RAND	0.92	**0.94**	**0.94**	**0.94**
	INEX-RULE	0.72	0.87	0.93	**0.95**
	INEX-CLEAN	0.92	0.95	**0.96**	**0.96**
PY08	**DBLP-RAND**	0.38	**0.41**	**0.41**	**0.41**
	DBLP-RULE	0.11	0.11	**0.13**	0.12
	DBLP-CLEAN	0.17	**0.19**	**0.19**	**0.19**
	INEX-RAND	**0.24**	**0.24**	**0.24**	**0.24**
	INEX-RULE	0.08	**0.09**	0.08	0.08
	INEX-CLEAN	0.07	**0.08**	0.07	0.07

TABLE V
MRR vs. γ ($\beta = 5$)

D. Efficiency

Query	XClean	PY08
DBLP-RAND	0.01	0.17
DBLP-RULE	0.53	5.11
DBLP-CLEAN	0.01	0.16
INEX-RAND	0.11	0.77
INEX-RULE	12.24	59.15
INEX-CLEAN	0.13	0.75

TABLE VI
AVERAGE RUNNING TIME IN SECONDS ($\gamma = 1000$)

We report the average query time in Table VI. We can observe that:

- XClean is about 5-10 times faster than PY08, as PY08 requires multiple passes of inverted lists when combining segments while XClean only requires a single pass.
- The query processing time is higher on INEX than on DBLP. This is because (1) the INEX data is almost ten times as large as the DBLP data. (2) the vocabulary of INEX is also six times as large as that of DBLP. Hence, on average, the inverted lists and the size of variants for a query is much larger for the INEX dataset.
- The query processing time for RULE perturbation is higher than RAND and CLEAN. This is mainly because the common misspellings tends to have a larger edit distance than the other two sets. With much more variants to consider, both algorithms slow down noticeably.

VIII. RELATED WORK

Spellchecking. Spellchecking is an old problem. It aims to find words in a document that are likely to be misspelt. It has been widely used in word processing software and the Internet. Existing approaches can be roughly classified into dictionary-based ones and context-sensitive ones. We refer readers to the surveys [17], [18] and recent work for Web queries in [3], [5], [6].

Query correction on structured databases differs from traditional and Web-based spell checking and correction in that (1) the query does not necessarily form a valid English sentence (e.g., `join dewitt vldb`), and (2) we need to take into consideration the structures within the data source to provide *valid* and *high-quality* suggestions.

Keyword Query Cleaning. PY08 [2] is the most relevant work to ours. We analyzed their algorithm in details in Section II.

Pu presented a subsequent work in [7], in which the query generation process is modeled based on a Hidden Markov Model (HMM). There are two weaknesses in their model. First, their model is built on a huge state space, since each tuple approximately matching a query keyword is considered as a state. This may become infeasible even for moderate-sized databases, since the complexity of the inference algorithm on HMM is $O(|U||Q|^2)$ where U is the state space. Model parameters also become harder to learn when the state space is large due to the data sparsity problem. Aggressive states pruning was employed to alleviate the problem but it may hurt the result quality. Second, HMM is a sequential model. The query generation process is assumed to be sequentially traveling in the database while emitting query keywords. This may not be the case in general, as one may combine several concepts that are not sequentially related and make a query. Since only the preliminary result is reported, it is not clear whether this model performs well in general cases.

Query Relaxation for Structured XML Queries. Query relaxation for structured queries has also been studied in [19] for relational databases and [20], [21], [22] for XML data. In their problem settings, certain structural constraints need to be relaxed or removed in order to avoid returning empty result or insufficient number of results. That is a different problem from ours as we are not concerned with structured queries.

XML Keyword Search. Keyword search in XML databases has been studied extensively in recent years, including

early work in incorporating keyword into XML query languages [23], and recent ones on free-style keyword search. Different structures of search results have been explored, such as ELCAs [24], [25], SLCAs [26], [27], MLCAs [14], and interconnection semantics [28], [29]. Two recent works XSeek [30] and XReal [10] utilized the statistics of underlying XML database to provide a scoring mechanism that is sensitive to the data; we share the same spirit as them. Another line of work [31], [15] focused on developing efficient algorithms for finding matches with LCA-based semantics. Chen *et al* studied the top-k processing of XML keyword queries and proposed an algorithm with special index structure for finding top-k ELCAs [32].

Our work is orthogonal to these studies in the sense that we measure the quality of a query rather than produce search results that conform to a specific semantics.

In this paper we mainly assume the specific node types semantics and briefly discuss how to incorporate SLCA semantics. However, our framework is general enough to accommodate other semantics as well.

IX. Conclusions and Future Work

In this paper, we proposed a novel and sound framework for XML keyword query cleaning. The central idea is to use query results to score the alternative queries. We derived the final scoring function based on the probabilistic theory and the state-of-the-art language model, and take into consideration (1) the error model that models the typing errors; (2) the query generation model that estimates the probablity of users being interested in a particular query; and (3) XML tree structure and keyword query semantics, which help to identify the units of information in the XML data and measure the quality of query in a finer granularity.

We also proposed an algorithm, XClean, that computes the top-k alternative query efficiently. We have conducted an extensive experimental evaluation with previous methods and major search engines. The results demonstrate both the effectiveness and efficiency of the proposed approach.

We did not take into consideration that concatenation/splitting of words in this paper. This kind of transformation is another common source of misspellings in real life queries. Extending the XClean system to include other kinds of misspellings is our future work.

Acknowledgement

Jianxin Li and Chengfei Liu are supported by ARC Discovery Projects DP0878405 and DP110102407. Wei Wang is supported by ARC Discovery Projects DP0987273, DP0881779 and DP0878405.

References

[1] Y. Chen, W. Wang, Z. Liu, and X. Lin, “Keyword search on structured and semi-structured data,” in *SIGMOD Conference*, 2009, pp. 1005–1010.

[2] K. Q. Pu and X. Yu, “Keyword query cleaning,” *PVLDB*, vol. 1, no. 1, pp. 909–920, 2008.

[3] S. Cucerzan and E. Brill, “Spelling correction as an iterative process that exploits the collective knowledge of web users,” in *EMNLP*, 2004, pp. 293–300.

[4] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang, “WebTables: exploring the power of tables on the web,” *PVLDB*, vol. 1, no. 1, pp. 538–549, 2008.

[5] M. Li, M. Zhu, Y. Zhang, and M. Zhou, “Exploring distributional similarity based models for query spelling correction,” in *COLING-ACL*, 2006, pp. 1025–1032.

[6] Q. Chen, M. Li, and M. Zhou, “Improving query spelling correction using web search results,” in *EMNLP*, 2007, pp. 181–189.

[7] K. Q. Pu, “Keyword query cleaning using hidden markov models,” in *KEYS*, 2009, pp. 27–32.

[8] E. Mays, F. J. Damerau, and R. L. Mercer, “Context based spelling correction,” *Inf. Process. Manage.*, vol. 27, no. 5, pp. 517–522, 1991.

[9] C. Zhai and J. D. Lafferty, “Two-stage language models for information retrieval,” in *SIGIR*, 2002, pp. 49–56.

[10] Z. Bao, T. W. Ling, B. Chen, and J. Lu, “Effective XML keyword search with relevance oriented ranking,” in *ICDE*, 2009, pp. 517–528.

[11] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum, “Probabilistic information retrieval approach for ranking of database query results,” *ACM Trans. Database Syst.*, vol. 31, no. 3, pp. 1134–1168, 2006.

[12] B. S. T. Bocek, E. Hunt, “Fast Similarity Search in Large Dictionaries,” Department of Informatics, University of Zurich, Tech. Rep. ifi-2007.02, April 2007.

[13] W. Wang, C. Xiao, X. Lin, and C. Zhang, “Efficient approximate entity extraction with edit distance constraints,” in *SIGMOD Conference*, 2009, pp. 759–770.

[14] Y. Li, C. Yu, and H. V. Jagadish, “Schema-free XQuery,” in *VLDB*, 2004, pp. 72–83.

[15] C. Sun, C. Y. Chan, and A. K. Goenka, “Multiway SLCA-based keyword search in XML data,” in *WWW*, 2007, pp. 1043–1052.

[16] L. Denoyer and P. Gallinari, “The Wikipedia XML Corpus,” *SIGIR Forum*, 2006.

[17] K. Kukich, “Techniques for automatically correcting words in text,” *ACM Comput. Surv.*, vol. 24, no. 4, pp. 377–439, 1992.

[18] *English Spelling and the Computer*. Longman Group, 1996.

[19] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica, “Relaxing join and selection queries,” in *VLDB*, 2006, pp. 199–210.

[20] S. Amer-Yahia, S. Cho, and D. Srivastava, “Tree pattern relaxation,” in *EDBT*, 2002, pp. 496–513.

[21] T. Brodianskiy and S. Cohen, “Self-correcting queries for XML,” in *CIKM*, 2007, pp. 11–20.

[22] S. Cohen and T. Brodianskiy, “Correcting queries for XML,” *Inf. Syst.*, vol. 34, no. 8, pp. 690–710, 2009.

[23] D. Florescu, D. Kossmann, and I. Manolescu, “Integrating keyword search into XML query processing,” *Computer Networks*, vol. 33, no. 1-6, pp. 119–135, 2000.

[24] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, “XRANK: Ranked keyword search over XML documents,” in *SIGMOD Conference*, 2003, pp. 16–27.

[25] L. Kong, R. Gilleron, and A. Lemay, “Retrieving meaningful relaxed tightest fragments for XML keyword search,” in *EDBT*, 2009, pp. 815–826.

[26] Y. Xu and Y. Papakonstantinou, “Efficient keyword search for smallest LCAs in XML databases,” in *SIGMOD Conference*, 2005, pp. 537–538.

[27] Z. Liu and Y. Chen, “Reasoning and identifying relevant matches for XML keyword search,” *PVLDB*, vol. 1, no. 1, pp. 921–932, 2008.

[28] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, “XSEarch: A semantic search engine for XML,” in *VLDB*, 2003, pp. 45–56.

[29] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv, “Interconnection semantics for keyword search in XML,” in *CIKM*, 2005, pp. 389–396.

[30] Z. Liu and Y. Chen, “Identifying meaningful return information for XML keyword search,” in *SIGMOD Conference*, 2007, pp. 329–340.

[31] Y. Xu and Y. Papakonstantinou, “Efficient LCA based keyword search in XML data,” in *EDBT*, 2008, pp. 535–546.

[32] L. J. Chen and Y. Papakonstantinou, “Supporting top-k keyword search in XML databases,” in *ICDE*, 2010, pp. 689–700.