ASTERIX: Ambiguity and Missing Element-Aware XML Keyword Search Engine

Ba Quan Truong
Nanyang Technological University
Singapore
bqtruong@ntu.edu.sg

Sourav S Bhowmick
Nanyang Technological University
Singapore
assourav@ntu.edu.sg

Curtis Dyreson
Utah State University
USA
curtis.dyreson@usu.edu

Hong Jing Khok
Nanyang Technological University
Singapore
hjkhok1@e.ntu.edu.sg

ABSTRACT

Despite a decade of research on XML keyword search (XKS), demonstration of a high quality XKS system has still eluded the information retrieval community. Existing XKS engines primarily suffer from two limitations. First, although the *smallest lowest common ancestor* (SLCA) algorithm (or a variant, *e.g.,*ELCA) is widely accepted as a meaningful way to identify subtrees containing the query keywords, SLCA typically performs poorly on documents with *missing elements*, *i.e.,* (sub)elements that are optional, or appear in some instances of an element type but not all. Second, since keyword search can be ambiguous with multiple possible interpretations, it is desirable for an XKS engine to automatically *expand* the original query by providing a classification of different possible interpretations of the query w.r.t. the original results. However, existing XKS systems do not support such *result-based query expansion.* We demonstrate ASTERIX, an innovative XKS engine that addresses these limitations.

1 INTRODUCTION

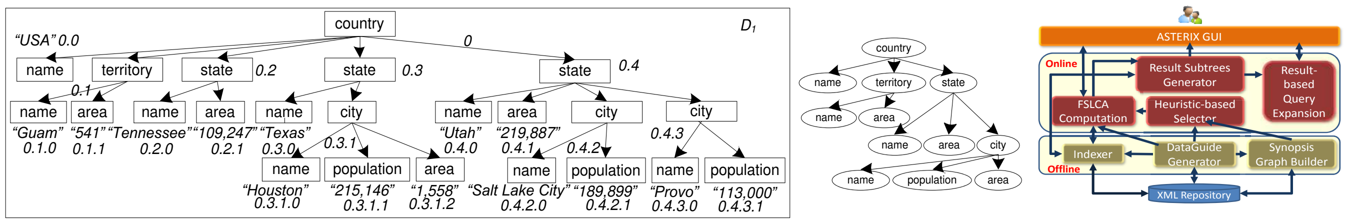
The lack of expressivity and inherent ambiguity of XML keyword search (XKS) bring in three key challenges in building a superior XKS engine. First, we need to automatically connect the nodes that match the search keywords in an intuitive, meaningful way. In this context, the notion of *smallest lowest common ancestor* (SLCA) [12] is arguably the most popular strategy to address this challenge and has become the building block of many XML keyword search approaches [2, 7, 8]. A keyword search using the SLCA semantics returns nodes in the XML tree such that each node in the result satisfies the following two conditions: (a) the subtree rooted at a node contains all of the keywords, and (b) no proper descendant of the node satisfies condition (a). The set of returned nodes is referred to as the SLCAs of the keyword search query. The second challenge deals with effective identification of the desired return information. Specifically, it focuses on filtering nodes within these matching subtrees to produce relevant and coherent results. There have been several research efforts toward addressing this challenge [2, 7, 8] such as filtering irrelevant matches under an SLCA node by returning only *contributors* [8]. The third challenge is *ranking* [2, 6] or *clustering* [5] these relevant result subtrees according to certain criteria and returning them.

The aforementioned challenges have inspired a large body of research on XKS [2, 7, 8, 12]. Several XKS systems have also been demonstrated in major venues [1, 6]. Hence, at first glance one may question the need for yet another XKS demonstration. In this paper, *we justify the need for such a demonstration by advocating that a high quality* XKS *system has still eluded the information retrieval research community after all these years*!

Despite the admirable efforts of state-of-the-art XKS techniques, they suffer from two key drawbacks. First, as shown in our previous work [11], XKS techniques based on SLCA and its variants (*e.g.,* ELCA) perform poorly in the presence of the *missing element phenomenon.* Due to the "relaxed" structure of XML data, a subelement may appear in one nested substructure of an XML document but be missing in another "similar" substructure. Note that in many real-world XML documents more than 40% of the element labels are *missing labels* [11]. Hence, it is highly possible for users' searches to involve missing elements. For example, the `area` element in the XML document D_1 in Figure 1(a) appears in the first `city` substructure but is missing in the last two substructures. A keyword query that contains the label of a missing element lowers the quality of SLCA nodes. For example, consider the query Q_1`(Provo area)` on D_1. The node with ID 0.4 (for brevity, we will use n_{id} to denote a node with ID *id*) is selected as the SLCA node by [12]. However, $n_{0.4.1}$ is not a relevant match as it is not `Provo`'s area.

Second, a keyword query can be ambiguous with multiple possible interpretations or be exploratory in nature where the user does not have a specific search target but would like to navigate among possible relevant answers. For example, the results of a query Q_2`(alaska)` may contain subtrees having multiple interpretations such as the country *Alaska* and the *Alaskan range* mountains. Hence, an XKS engine that can automatically *expand* the original query (*e.g.,* Q_2) by providing a classification of different possible interpretation of the query w.r.t the original result set is desirable. For the above example, Q_2 can be expanded to Q_{2a}`(alaska,`

SIGIR '17, August 7–11, 2017, Shinjuku, Tokyo, Japan
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5022-8/17/08…$15.00.
http://dx.doi.org/10.1145/3077136.3084137



(a) XML document (b) DataGuide (c) ASTERIX architecture

Figure 1: (a) Sample XML documents D_1, (b) its DataGuide, and (c) the architecture of ASTERIX.

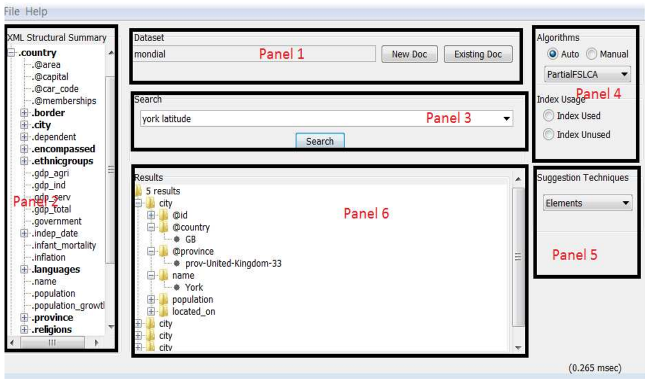


Figure 2: The ASTERIX GUI.

country) and Q_{2b}(alaska, mountain) that classifies the original result set into two key clusters containing information related to Alaska and the Alaskan range, respectively. Observe that such *result-based* query expansion can guide users to focus on the relevant subset of the original query results when a specific expanded query is chosen. Unfortunately, existing XKS systems do not provide effective techniques for such result-based query expansion.

In this demonstration, we present a novel XKS system called ASTERIX (**A**mbiguity and mi**S**sing elemen**T**-aware k**E**yword sea**R**ch **I**n **X**ML) to address the above limitations. ASTERIX has two novel features. First, it produces high-quality SLCA nodes, which are identical to SLCA nodes produced by any other SLCA-based XKS technique when the query does *not* contain missing elements or labels but unlike these existing techniques, avoids irrelevant results when missing elements are involved. Second, it supports result-based query expansion that lets users navigate within multiple possible interpretations of the result set.

2 SYSTEM ARCHITECTURE

Figure 1(c) shows the system architecture of ASTERIX. We model an XML document D as an ordered and node-labeled tree. Each node $n \in D$ is assigned a Dewey number as its identifier (*e.g.,* Figure 1(a)) and is associated with a label (*i.e.,* tag) and text value (if any). Each node n has a *type* defined by its prefix path. The XML *Repository* stores the XML files in the disk. The *Indexer* module traverses D to generate an inverted list of the keywords and a *path index* to support efficient evaluation of keyword search. The *DataGuide Generator* module constructs the DataGuide [4] of D. The *Synopsis Graph Builder* module uses the DataGuide to build a *synopsis graph* of D. Note that these three modules are executed offline as the outputs remain invariant unless the document is modified. Given a keyword query Q on D, the FSLCA *Computation* module implements the two variants of the MESSIAH algorithm, namely C-MESSIAH and P-MESSIAH [11], to find *full* SLCA (FSLCA) nodes in D. Specifically, these nodes enable an XKS engine to handle missing elements. The *Heuristic-based Selector* module leverages the synopsis graph to facilitate automatic selection of the correct variant of MESSIAH for processing Q without any user intervention. The *Result Subtrees Generator* module leverages these FSLCA nodes to extract a set of result subtrees in D that match the query and ranks them. The *Result-based Query Expansion* module takes the result subtrees as input and generates the top-k *expanded* queries to provide multiple interpretations of the original query (if any). We now elaborate on these modules.

The GUI Module. Figure 2 depicts the screenshot of ASTERIX GUI. A user begins formulating a query by choosing an XML document as the query target. Panel 1 allows her to upload a new XML document or retrieve an existing XML document. The left panel (Panel 2) displays the DataGuide of the target document. Panel 3 depicts the area for formulating keyword queries and to view the expanded queries based on the original result set. Panel 4 allows a user to choose different variants of MESSIAH. By default, ASTERIX automatically chooses the correct variant of MESSIAH by invoking the *Heuristic-based Selector* module. However, for demonstration purpose we also provide an option to switch to "manual" mode (by selecting the manual radio button) and choose either the P-MESSIAH or C-MESSIAH variant from the dropdown list before executing the query. Note that Panel 4 is disabled in "live" applications as the correct variant is chosen automatically. Panel 5 allows us to choose the query expansion strategy *i.e.,* whether the keywords added to the original query is selected only from element labels in the result set or from both element labels and text values. Panel 6 displays the query results.

The DataGuide Generator Module. Given D, this module extracts its *DataGuide* in linear time by employing [4]. A *DataGuide* $\mathbb{S}$ is a prefix tree representing all unique paths in D *i.e.,* each unique path p in D is represented in $\mathbb{S}$ by a node (referred to as *schema node*) whose root-to-node path is p. Hence, each schema node in $\mathbb{S}$ corresponds to a *type* and the hierarchical relationship among schema nodes represents *type relations* (*e.g.,* descendant or child type). For example, Figure 1(b) depicts the DataGuide of document D_1 in which t_{city} is a descendant (or child) type of t_{state}.

The Indexer Module. This module generates two types of indexes on an XML document D. (a) An *Inverted List* where each keyword is mapped to a list of matches sorted by document order. The list of matches is also partitioned into two sublists corresponding to *value matches* and *type matches* of the keyword. (b) A *path index* where each root-to-node path p in the DataGuide tree is mapped to a list of nodes of path p in D. Each index is a B-tree. In addition, it generates several statistics of D (*e.g.,* keyword frequency).

The Synopsis Graph Builder Module. This module generates a *synopsis graph* of D, which shall be exploited by the *Heuristic-based Selector* module (discussed later). The *synopsis graph* G is a lightweight version of the *XSketch*-index [10], which is a directed acyclic graph synopsis where each *synopsis node* g represents a set of data nodes with the same labels and each edge (g_p, g_c) signifies the parent-child relationship between the data nodes of g_p and g_c. Each internal (resp. leaf) synopsis node stores the structural (resp. value) distribution of the child labels (resp. value tokens) among its data nodes. Specifically, G is stored with the DataGuide $\mathbb{S}$ where each synopsis node corresponds to exactly one schema node in $\mathbb{S}$.

The FSLCA Computation Module. This module generates the same SLCA nodes as any state-of-the-art approach when the query does not involve missing elements but avoids irrelevant results when missing elements are involved. To this end, it implements two variants of a novel algorithm called MESSIAH, namely C-MESSIAH and P-MESSIAH. The reader may refer to [11] for detailed description and performance results of these algorithms. Here, we briefly describe the key idea.

Since the missing element phenomenon does not occur in an XML document without missing elements (called a *full document*), existing SLCA-based techniques work fine on it. Hence, MESSIAH *logically* transforms D to a *minimal full document* $F(D)$ (*i.e.,* it does not physically add missing elements), where all missing elements are represented as empty elements, and then employs efficient strategies to identify *full* SLCA (FSLCA) nodes from it. For example, if we compute the SLCA nodes on $F(D_1)$ (*i.e.,* minimal full document of D_1) for Q_1, it would produce $n_{0.4.3}$ due to the existence of the empty element `area` as its child. Hence, a *full* SLCA (FSLCA) node of a query Q in a document D is an SLCA node of Q on $F(D)$.

Since a full document $F(D)$ may contain empty nodes that do not exist in the original document D, each FSLCA node can be categorized as *complete* (C-FSLCA) or *partial* (P-FSLCA). In the case of the former, both the FSLCA node and its supporting matches are in the original document D whereas for the latter the FSLCA node is in D but some of its supporting matches may not be. For example, consider the query Q_3(`area`, `city`) on D_1. Then the FSLCA node $n_{0.3.1}$ is a C-FSLCA as its subtree includes matches for both `city` and `area`. On the other hand, $n_{0.4.2}$ is a P-SLCA node as it does not have any `area` element as descendant in D_1. This module implements two algorithms called C-MESSIAH and P-MESSIAH to efficiently identify these two categories of FSLCA nodes, respectively.

Both variants of the algorithm retrieve multiple document-order-sorted streams of candidates to find FSLCA nodes. P-MESSIAH retrieves two streams of candidates, namely L_1 and L_2, for P-FSLCA nodes without value matches and with at least one value match, respectively. First, in the L_1 stream it locates the label matches in the DataGuide of D and finds the SLCAs of these matches (*e.g.,* `city` node in Figure 1(b) for Q_3). Next, it retrieves the instances in D corresponding to these DataGuide nodes (*e.g.,* $n_{0.3.1}$, $n_{0.4.2}$, and $n_{0.4.3}$). Second, in the L_2 stream, for each value match v (anchor node), it computes the LCA a_1 between v and the last and next value match of each keyword (*e.g.,* $n_{0.4.2}$ for the value match $n_{0.4.2.0}$ in Figure 1(a)). Then it computes the SLCA level ℓ between v's path and paths of each label keyword using DataGuide ($\ell = 3$ for Q_3) and finds ancestor a_2 of v at level ℓ (*e.g.,* $n_{0.4.2}$). The P-FSLCA candidate of v is the descendant between a_1 and a_2 (*e.g.,* $n_{0.4.2}$). Hence, the final P-FSLCA nodes are $n_{0.3.1}$, $n_{0.4.2}$, and $n_{0.4.3}$.

The C-MESSIAH, on the other hand, retrieves three streams of candidates L_s, L_1, and L_2, where L_1 and L_2 are the same as in P-MESSIAH and L_s is the stream of SLCA candidate nodes. The results of C-MESSIAH are generated only from L_s by filtering nodes using candidates from L_1 and L_2. For instance, for the query Q_3, the L_s stream contains the SLCA candidates $n_{0.3.1}$ and $n_{0.4}$ in D_1. The L_1 stream is used to filter any candidate nodes whose paths are prefixes of the path `country/state/city` (*e.g.,* $n_{0.4}$). Using the L_2 stream, for each value match v, it compute the P-FSLCA candidate anchored at v ($n_{0.4.2}$) and filters candidate ancestors of the P-FSLCA node. Hence, the final C-FSLCA node is $n_{0.3.1}$.

The Heuristic-based Selector Module. Observe that C-MESSIAH ignores result nodes containing missing elements (returns C-FSLCA) whereas P-MESSIAH returns all complete FSLCA nodes of C-MESSIAH as well as additional results containing missing elements where these elements are indicated as empty nodes (returns P-FSLCA). Since a user may not have sufficient knowledge to manually choose a variant for a query Q, it is important to automatically deduce which variant of MESSIAH needs to be executed for Q. This module implements a heuristic-based mechanism to achieve it.

Intuitively, the selection choice is influenced by the usefulness of the additional results generated by P-MESSIAH. We advocate that it depends on the number of complete FSLCAs as well as the number of results (denoted by N) desired by a user. So if an XKS system returns more than N C-FSLCA results, then a user may not be interested in the results with missing elements. Consequently, C-MESSIAH is relatively more appropriate for this case. On the other hand, if there are fewer than N C-FSLCA results, then displaying additional results with missing elements using P-MESSIAH will be potentially useful.

The challenge here is to estimate the number of C-FSLCAs *a priori*. We address it by utilizing the synopsis graph (*Synopsis Graph Builder*) [11]. To illustrate the selection process using it, let us reconsider Q_1 and Q_3 on D_1. For Q_1(`Provo area`), from the synopsis graph we know that all `cities` have `name` but only 33% of `name` have value `Provo`. Meanwhile, only 33% of `cities` have `area`. Assuming the distributions are independent, 11% of `cities` have both `Provo` and `area`. Since there are 3 `city` elements in D_1, the estimated result size is $3 \times 0.11 = 0.33$. Similarly, for Q_3(`city area`), 33% of `city` elements have `area` which leads to the estimated result size of 1. Let $N = 1$. Since $0.33 < 1$, P-MESSIAH is used for Q_1 but C-MESSIAH is used for Q_3.

The Result Subtrees Generator Module. This module selects relevant return nodes within the FSLCA subtrees satisfying the query Q and ranks them based on *certain* criteria (*e.g.,* subtree size). Since our FSLCA computation and result-based query expansion modules are orthogonal to it, any state-of-the-art approaches related to relevant information identification and result ranking (*e.g.,* [2, 7, 8]) can be used here. ASTERIX uses the strategy in [2].

The Result-based Query Expansion Module. Given the set of result subtrees of a query Q, this module generates top-k ($k = 4$ in the current version) expanded queries from the result set. Broadly, it consists of two key phases, *keyword gathering* and *keyword refinement.* In the *keyword gathering* phase, for each result tree, the root-to-leaf paths are extracted and node labels in the same level are grouped together. Each level is then associated with a set of distinct keywords. Then, duplicate keywords across different subtrees are removed. The intention here is to eventually select some of these keywords for expanding Q. However, since there could be many expanded keywords, the output of this phase needs to be *refined* based on certain heuristics.

In the *keyword refinement* phase, it first selects the top-m ($m = 10$) frequent keywords in the above collection. Next, keywords that do not produce any new information when they are added to Q (*i.e.,* produces the same set of subtrees as Q) are pruned. For instance, consider the query Q_4(`paris`). Adding the keyword `"france"` to Q_4 does not generate any new information as the results of Q_4 and the query Q_{4a}(`paris, france`) are identical. Then, it identifies keywords that can be used to *disambiguate* Q. Intuitively, the goal is to identify keywords that co-occur with Q but appear in different context and have very different distributions. For example, consider the query Q_2 and keywords `"country"` and `"mountain"`. Both these keywords co-occur with `alaska` in *Mondial* but have different context and result distributions. Specifically, for each keyword w it computes its result distribution and *intersection probability* (probability of shared subtrees with the results of Q). Next, it computes the *information gain* due to the addition of w to Q by leveraging KL-divergence. Hence, each keyword is associated with three measures and the goal is to select top-k keywords when added to Q maximize the result distribution and information gain and minimize intersection probability. We exploit Fagin's *Threshold Algorithm* (TA) [3] to efficiently generate the top-k expanded queries.

3 RELATED SYSTEMS AND NOVELTY

Several XKS result retrieval techniques have been proposed in the literature [2, 7] based on SLCA matching semantics and its variants (*e.g.,*ELCA) [12]. However, unlike ASTERIX these efforts do not address the missing element phenomenon effectively. There has also been recent research in improving user experience for XKS [1, 2, 5]. However, none of these efforts focus on expanding queries with multiple interpretations by analyzing the result set. In addition, since these efforts are grounded on SLCA or ELCA semantics, they also suffer from the missing element problem. Liu *et al.* [9] investigated the problem of query expansion based on clustering Web search results (*i.e.,* textual content). However, this method cannot be adopted effectively in XML data.

Several XKS systems have been demonstrated in major conference venues [1, 6]. These systems also suffer from the missing element problem as they leverage the SLCA semantics. Nevertheless, our demonstration is complimentary to them as we focus on the missing element problem and automatic expansion of ambiguous queries based on query results.

4 DEMONSTRATION OBJECTIVES

ASTERIX is implemented in Java JDK 1.7 on top of Berkeley DB 4.0.103. Our demonstration will be loaded with a few popular real XML datasets (*e.g.,* Mondial, INTERPRO, DBLP and Shakespeare) of sizes up to 1GB. Example queries with or without missing labels will be presented for formulation. Users can also write their own ad-hoc queries in Panel 3.

FSLCA computation and result display. One of the key objectives of the demonstration is to enable the audience to interactively experience the FSLCA *Computation* module that addresses the missing element problem. Specifically, we first set the Panel 4 in `manual` mode. Then, the user can formulate a query with missing label (Panel 3), select `PartialFSLCA` or `CompleteFSLCA` strategy in Panel 4 to invoke the P-MESSIAH or C-MESSIAH algorithm, respectively, and observe the differences in the result set in Panel 6. Through this experience, users will be able to appreciate the limitation of SLCA semantics in tackling the missing element phenomenon. Note that users can also formulate queries without any missing label and experience that the results returned by MESSIAH is identical to those returned by SLCA-based techniques.

Automatic selection of MESSIAH variant. Through the GUI, we shall also demonstrate the *Heuristic-based Selector* module, which selects the correct variant of MESSIAH automatically for a given query. First, we set Panel 4 to `automatic` mode. Then, the user can re-execute the above query (or any other query) and observe in Panel 6 the correctness of the selection of C-MESSIAH or P-MESSIAH based on the result size estimation technique described earlier. Additionally, through the aforementioned experiences users will be able to appreciate superior performance of these modules, consistent with the results reported in [11].

Result-based query expansion. Lastly, users can experience the working of the *Result-based Query Expansion* module by clicking on the search box in Panel 3 after executing a query. They will be able to view top-4 expanded queries generated from the result set. Selecting any of the expanded queries will enable the user to view a subset of the original results that contain these keywords.

REFERENCES

[1] Z. Bao, et al. XReal: an interactive XML keyword searching, *In CIKM*, 2010.
[2] Z. Bao, et al. Towards an effective XML keyword search, *In IEEE TKDE*, 22(8), 2010.
[3] R. Fagin, et al. *Optimal aggregation algorithms for middleware*, In PODS, 2001.
[4] R. Goldman, J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases, *In VLDB*, 1997.
[5] X. Liu, et al. Returning clustered results for keyword search on XML documents, *IEEE TKDE*, 23(12), 2011.
[6] Z. Liu, et al. Targetsearch: A ranking friendly XML keyword search engine. *In ICDE*, 2010.
[7] Z. Liu, Y. Chen. Identifying meaningful return information for XML keyword search, *In SIGMOD*, 2007.
[8] Z. Liu, Y. Chen. Reasoning and identifying relevant matches for XML keyword search, *In PVLDB*, 1(1), 2008.
[9] Z. Liu, et al. Query expansion based on clustered results, *In VLDB*, 2011.
[10] N. Polyzotis, et al. Selectivity estimation for XML twigs, *In ICDE*, 2004.
[11] B. Q. Truong, et al. MESSIAH: Missing element-conscious SLCA nodes search in XML data, *In SIGMOD*, 2013.
[12] Y. Xu, Y. Papakonstantinou. Efficient keyword search for smallest lcas in XML databases, *In SIGMOD*, 2005.