MESSIAH: Missing Element-Conscious SLCA Nodes Search in XML Data

Ba Quan Truong[§] **Sourav S Bhowmick**[§] **Curtis Dyreson**[†] **Aixin Sun**[§]

[§]School of Computer Engineering, Nanyang Technological University

[†]Department of Computer Science, Utah State University

bqtruong|assourav|axsun@ntu.edu.sg, curtis.dyreson@usu.edu

Abstract

Keyword search for smallest lowest common ancestors (SLCA s) in XML data has been widely accepted as a meaningful way to identify matching nodes where their subtrees contain an input set of keywords. Although SLCA and its variants (*e.g.,*MLCA) perform admirably in identifying matching nodes, surprisingly, they perform poorly for searches on *irregular schemas* that have *missing elements*, that is, (sub)elements that are optional, or appear in some instances of an element type but not all (*e.g.,* a <population> subelement in a <city> element might be optional, appearing when the population is known and absent when the population is unknown). In this paper, we generalize the SLCA search paradigm to support queries involving missing elements. Specifically, we propose a novel property called *optionality resilience* that specifies the desired behaviors of an XML keyword search (XKS) approach for queries involving missing elements. We present two variants of a novel algorithm called MESSIAH (**M**issing **E**lement-conscious **S**ystem **H**igh-quality SLC**A** searc**H**), which are optionality resilient to irregular documents. MESSIAH *logically* transforms an XML document to a *minimal full document* where all missing elements are represented as empty elements, *i.e.,* the irregular schema is made "regular", and then employs efficient strategies to identify *partial* and *complete full* SLCA nodes (SLCA nodes in the full document) from it. Specifically, it 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. Our experimental study demonstrates the ability of MESSIAH to produce superior quality search results.

Dataset	Number of Labels	Missing Label Ratio
Mondial	70	64.3%
SIGMOD	12	0%
DBLP	41	68.3%
Uniprot	94	72.3%
PDB	267	1.12%
Interpro	68	41.1%
Shakespeare	22	40.9%

Figure 1: Missing label ratio on different datasets.

1 Introduction

Keyword search on XML data (XKS) has gained popularity as it relieves users from learning complex XML query languages (*e.g.,* XPath, XQuery) and from having to know the structure of underlying data. However, the lack of expressivity and inherent ambiguity bring in two key challenges in performing XKS [9]. First, we need to automatically connect the nodes that match the search keywords in an intuitive, meaningful way. Second, we should effectively identify the desired return information. Specifically, the first challenge involves finding some nodes (*e.g.,* SLCA nodes) whose subtrees contain all matching nodes [2, 8, 17] while the second challenge focuses on filtering nodes within these subtrees to produce relevant and coherent results [1, 4, 7, 9, 10]. In this paper, we focus on the first challenge.

The *smallest lowest common ancestor* (SLCA) [17] is arguably the most popular technique for locating highly-related data nodes and has become the foundation for many recent XKS approaches [1, 4, 7, 9, 10]. A keyword search using the SLCA semantics returns nodes in the XML tree that satisfy the following two conditions: (a) the subtrees rooted at the nodes contain all the keywords, and (b) the nodes do not have any proper descendant that satisfies condition (a). The set of returned data nodes is referred to as the SLCA s of the keyword search query. For example, the only SLCA node satisfying the query Q_1 (`Provo area`) on the XML document D_2 in Figure 2 is the node with ID 0.4.3 (for brevity, in the sequel we will use n_{id} to denote a node with ID *id*). Figure 3(b)(i) shows the SLCA node $n_{0.4.3}$ (shaded) along with its matches to all Q_1's keywords.

1.1 A Problem with SLCA

Nested, tagged elements are the building blocks of XML. Each tagged element has a sequence of zero or more attribute/value pairs, and a sequence of zero or more subelements. An implication of this "relaxed" structure of XML data is that a subelement may appear in one nested substructure of an XML document but not in another "similar" substructure. For example, consider the XML document D_1 in Figure 2. Notice that although the `area` element appears in the first `city` substructure, it is missing in the last two substructures.

There can be many reasons for such missing elements. For instance, a specific

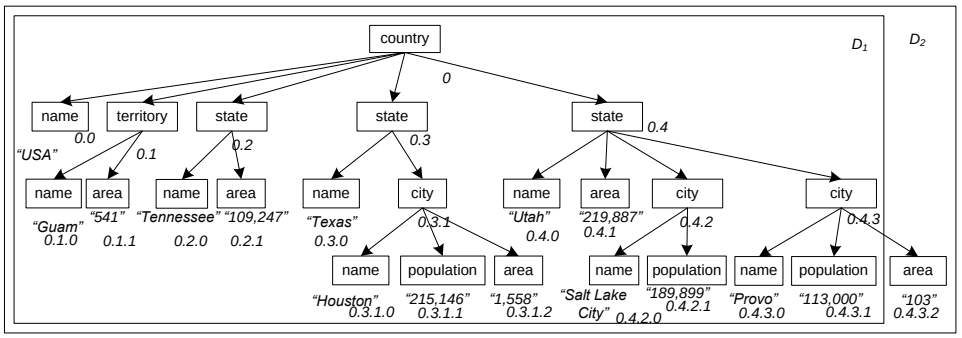


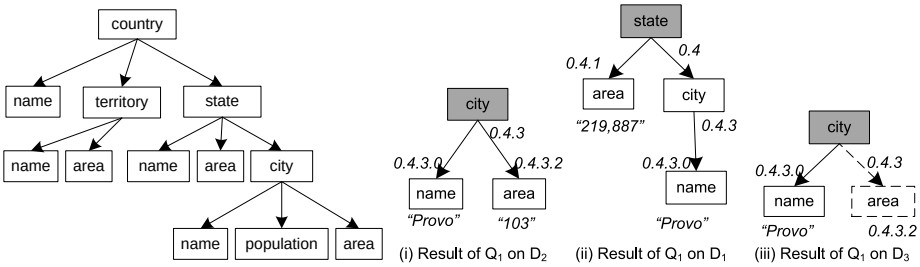
Figure 2: Sample XML documents D_1 and D_2 (D_2 contains all the nodes of D_1 and an additional `area` element (rightmost node)).

entity may not have an element (as an attribute) as it is meaningless in this context (*e.g.,* the `spouse` attribute of a `person` entity who is single is meaningless). Elements may also be missing due to incompleteness of data or human error. Whatever may be the semantics of a missing (or optional) element in a specific context, we refer to this scenario collectively as the *missing element*[1] phenomenon and the label of the missing element (*e.g.,* `area`) a *missing label*. Note that "missing element" covers the "missing attribute" scenario as well. Furthermore, the missing element could contain subelements that are also missing, *i.e.,* it could be a missing subtree in a data model instance. Note that in XML documents it is not mandatory to explicitly mark such missing data using *empty elements* (*i.e.,* `<tag></tag>` or `<tag/>`) or *empty attributes* (*i.e.,* `tag=""`)).

Is the missing element phenomenon prevalent in real XML documents? Interestingly, our initial investigation with several popular real-world XML datasets reveal that it is indeed so. Figure 1 shows a glimpse of this phenomenon. The third column shows the *missing label ratio*, which is the ratio between the number of missing labels and the number of unique labels in a specific dataset. Notice that in all but one of the datasets more than 40% of the labels are missing labels. Hence, it is highly possible for users' queries to involve missing elements. However, it is unrealistic to expect the users to be aware of elements which may be missing. Hence, it is imperative for the underlying XKS engine to handle this phenomenon appropriately.

Surprisingly, SLCA-based XML keyword search (XKS) techniques perform poorly in the presence of the missing element phenomenon. Specifically, the quality of SLCA nodes is adversely affected when a keyword query contains a missing label. For instance, for Q_1 on D_1, $n_{0.4}$ is selected as the SLCA node by [14, 17] as shown in Figure 3(b)(ii). However, $n_{0.4.1}$ is not a relevant match as it is not `Provo`'s area and does not conform to the user's search intention in Q_1. That is, selecting $n_{0.4}$ as

[1] We adopt the "missing element" term from [13] where it refers to elements that are declared optional in the DTD.



(a) Data Guide tree of D_1, D_2 and D_3 (b) Results of Q_1 (`Provo area`)

Figure 3: The DataGuide tree of the sample documents and a sample results.

an SLCA node would produce a result subtree containing irrelevant data.

At first glance, an advocate of classical SLCA computation techniques may argue that the selection of $n_{0.4}$ as an SLCA node may be considered relevant and meaningful if we accept that it gives an "approximation" of the desired result. However, we argue that such approximation is often irrelevant to the search intention. To illustrate this point further, let us consider the query Q_M(`York latitude`) on the *Mondial* dataset. One of the cities containing the keyword `York` but without latitude information is `York` city in the UK. For this result, SLCA-based approaches [14, 17] will return the `country` node of the UK which contains *all* cities in the UK. Observe that this result has two core drawbacks. First, it is too broad. There are nearly 100 cities of the UK in the dataset and it is unrealistic to expect users to explore all of them to find desired results. Second, it does not provide a precise or direct answer to the query and user's search intention. Since Q_M is specifically about `York`'s latitude information, retrieving "approximate" results containing many irrelevant data without answering the question directly may annoy users. Note that the core reason for such poor performance is that most XKS techniques consider matches "close" to each other as relevant but when the most relevant matches (*e.g.,* $n_{0.4.3.2}$ or `York`'s latitude) are missing, even the closest one may be irrelevant.

It may also seem that approaches built on top of SLCA nodes that effectively identify the desired return information (*e.g.,* [1, 4, 7, 9, 10]) can address the above limitation. Although these efforts certainly improve the result quality, when querying in the presence of missing elements, they make the same assumptions as SLCA-based techniques and cannot improve the poor input from [14, 17] significantly. In fact, for Q_1 on D_1, these techniques produce the same result as [14, 17].

1.2 Challenges

A straightforward approach to address the aforementioned limitation is to first modify the XML document by adding *empty* elements (`<tag/>`) to *represent missing* elements and then adopt an existing technique (*e.g.,* [17]) to find SLCA nodes. We refer to such a modified XML document as a *full document*. The intuition being that if there is no missing element in the full document, the aforementioned problem does not occur. For example, document D_3 in Figure 4 is a full document of D_1 in

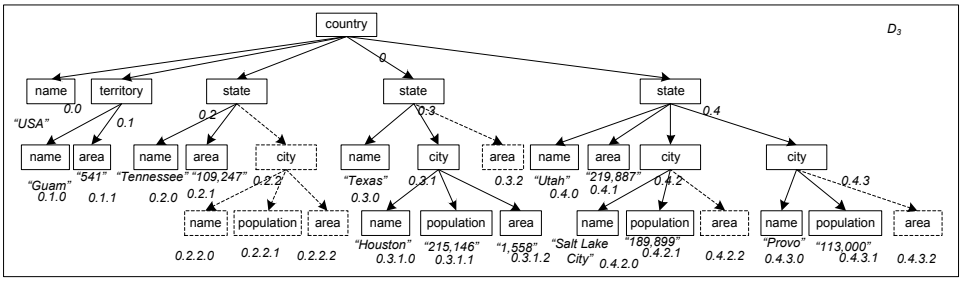


Figure 4: An example of full document

which all missing elements are represented by empty elements (depicted by a dotted box). For instance, the `area` node for `Provo`, which is missing in D_1, is now shown in D_3 using an empty `area` element. Clearly, an existing SLCA-based XKS can now find high quality SLCA nodes in the full document. For instance, Figure 3(b)(iii) shows the SLCA node $n_{0.4.3}$ (and all of its matches) for Q_1 on D_3. Notice that the irrelevant `area` node $n_{0.4.1}$ is no longer considered a match. The node $n_{0.4.3.2}$ is depicted in a dotted box to indicate that it is not originally in D_1.

Unfortunately, the aforementioned naïve solution to identify high quality SLCA nodes demands modification of the XML document which is usually undesirable in real-world applications. Firstly, an XML document may be accessed by multiple, heterogeneous applications such that altering it may have undesirable effect on some of these applications. Secondly, creating a copy of the existing document and adding empty elements to it will necessitate strict consistency checks, which is usually expensive to maintain especially for frequently updated documents. Thirdly, a full document version is not as space efficient as the original document, potentially affecting XML query performance. Fourthly, as mentioned earlier, an element may be missing in a document for various reasons. For instance, it may be meaningless for certain entities. Hence, explicitly adding it into these entities compromises data semantics. Lastly, often write permission of an XML document on the Web may not be available due to technical or legal reasons, making the aforementioned approach impractical. Furthermore, a direct consequence of such modification is the necessity of automatically pruning SLCA nodes in the full document which are empty elements. Otherwise, it may confuse end-users as they will not be able to locate some SLCA nodes when they browse the original XML document.

1.3 Contributions

Given a query Q on document D, how can we identify high-quality SLCA nodes in D that are also SLCA s in the full document version of D without physically creating the latter? In this paper, we address this core challenge. To the best of our knowledge, this is the first work that systematically addresses the SLCA computation problem germinated from the missing element phenomenon in XML data. More specifically,

we make the following contributions.

- In Section 3, we propose a novel property called *optionality resilience* that specifies the desired behaviors of an XKS approach for queries with missing elements. We analyze the optionality resilience of existing XKS approaches to systematically identify their limitations in handling the missing element phenomenon.
- In Section 4, we introduce the notion of *full* SLCA (FSLCA) that satisfies the optionality resilience property. *Full* SLCA S are SLCA S in the original document as well as in the full document version of it. Specifically, the set of FSLCA nodes is identical to SLCA nodes when the query does not contain missing elements but avoids irrelevant results when missing elements are involved. Hence, identifying FSLCA nodes in a document enables us to produce high-quality SLCA nodes even when the query involves missing elements.

  We introduce two variants of FSLCA nodes, namely *partial* and *complete* FSLCA, to give users flexibility in viewing results containing missing elements. *Complete* FSLCA does not return result nodes containing missing elements ($n_{0.4.3}$ in D_1 for Q_1). Alternatively, *partial* FSLCA returns result nodes with missing elements (*e.g.,* Figure 3(b)(iii)). In this representation the user is explicitly informed that the desired data is missing.
- In Section 5, we propose two variants of a novel algorithm called MESSIAH (**M**issing **E**lement-conscious **S**uboptimal **H**igh-quality SLCA searc**H**), that efficiently identifies sets of complete and partial FSLCA nodes for a given keyword query (possibly containing missing labels) on document D. An important feature of MESSIAH is that it *does not physically* add missing elements to D. Particularly, a full document is *logically* created with the sole goal of facilitating the generation of superior quality FSLCA nodes. Furthermore, MESSIAH can be integrated seamlessly with state-of-the-art techniques for retrieving relevant information (Step 2 of XKS) [1, 7, 9–11], potentially inheriting the strengths of these approaches.
- In Section 6, extensive experiments with real datasets are conducted to validate our solution's effectiveness, superiority, efficiency, and scalability. Specifically, we show that the quality of results generated by MESSIAH is identical to state-of-the-art SLCA computation techniques when the query does not contain missing labels. However, when the query contains missing labels then MESSIAH consistently generates superior quality results compared to these state-of-the-art techniques. Importantly, such superior result quality is achieved *without* any significant performance overhead of the proposed algorithms.

2 Preliminaries

We model an XML document D as an ordered and node-labeled tree. Each node $n \in D$ is assigned two functions $\mathbb{L}(n)$ and $\mathbb{F}(n)$ returning n's label (*i.e.,* tag) and

text value, respectively. A node n is called an **empty node (element)** when $\mathbb{F}(n)$ returns empty string. We use notation $n_1 \prec n_2$ to denote n_2 is a descendant of n_1. Each node $n \in D$ is assigned an identifier (ID), denoted as $id(n)$, satisfying the following condition: $\forall n_1, n_2 \in D$, $id(n_1) < id(n_2)$ iff n_1 precedes n_2 in document order. Note that the preorder attribute in containment encoding scheme [18] or Dewey number in Dewey encoding scheme [15] can be used as identifiers of nodes. Consistent with several existing XKS techniques, we also use Dewey numbers as node identifiers which enable fast LCA computation. The IDs of nodes in Figures 2 and 4 are their Dewey numbers in the corresponding documents.

Similar to [1, 7], we use prefix paths to indicate the **type** of a node. Two nodes are considered of the *same* type when they share the *same* prefix path. Using path as type enables us to use *DataGuide* [3] as type structure. 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 $\mathbb{S}$ also corresponds to a type and the hierarchical relationship among schema nodes represents type relations. Specifically, a type t_2 is called a *child (descendant) type* of another type t_1 if t_2's corresponding schema node is a child (descendant) of t_1's schema node in $\mathbb{S}$. We shall use $t_1 \prec_{\mathbb{S}} t_2$ to denote t_2 is a *descendant type* of t_1. Notice the subscript $\mathbb{S}$ is added to indicate type relation on $\mathbb{S}$. For instance, Figure 3(a) depicts the DataGuide of document D_1 in which $t_{state} \prec_{\mathbb{S}} t_{city}$. We also denote $N_D(t)$ as the set of all nodes of type t and $N_D(T)$ as $\bigcup N_D(t)$, $t \in T$. For example, $N_{D_1}(t_{state}) = \{n_{0.2}, n_{0.3}, n_{0.4}\}$ and $N_{D_1}(\{t_{state}, t_{territory}\}) = \{n_{0.1}, n_{0.2}, n_{0.3}, n_{0.4}\}$. Note that it is not mandatory for an XML document to be accompanied by its DataGuide as it can be easily extracted from a "schemaless" document in linear time [3]. In fact, DataGuides have been exploited in research related to XKS [1, 7, 9].

Given a query $Q(w_1, \ldots, w_k)$, if a keyword w_i is contained in either $\mathbb{L}(n)$ or $\mathbb{F}(n)$ then the node n is called a **label match** or a **value match** to w_i, respectively. The sets of all label and value matches to w_i in D are denoted as $L_D(w_i)$ and $V_D(w_i)$, respectively. The set of all **matches** to w_i in D, $M_D(w_i)$, is equal to $L_D(w_i) \cup V_D(w_i)$. Notably, if a node is a label match to w_i then all nodes of the same type are also label matches to w_i. We call this type a **type match** to w_i whose set is denoted as $T_D(w_i)$. When the context is clear, we denote $V_D(w_i)$, $L_D(w_i)$, $M_D(w_i)$, $T_D(w_i)$ as V_i, L_i, M_i, T_i, respectively. For example, in D_1 (Figure 2), V_{D_1}(`city`)= $\{n_{0.4.2.0}\}$, L_{D_1}(`city`)= $\{n_{0.3.1}, n_{0.4.2}, n_{0.4.3}\}$, T_{D_1}(`city`)= $\{t_{city}\}$.

Given a query $Q(w_1, \ldots, w_k)$, the XKS process can be broadly divided into two steps: (a) locating result nodes and (b) retrieving result matches [9]. The first step corresponds to selecting a set of **result nodes** $R(Q, D)$ from document D satisfying Q. For each result node $r \in R(Q, D)$, the second step retrieves a set of matches $S_D(r) \subseteq (M_D(w_1) \cap \cdots \cap M_D(w_k) \cap desc_D(r))$ where $desc_D(r)$ is the descendant set of r in D. In this paper, we focus on the first step.

As mentioned in Section 1, one of the most popular approaches to represent results nodes is SLCA [17]. Formally, for a query $Q(w_1 \ldots w_k)$ on a document D, let $lca(m_1, \ldots, m_k)$ be the LCA of k matches m_1,...,m_k and $LCA(Q, D)$ =

$\{lca(m_1, \ldots, m_k) | m_i \in M_k \forall i \in [1, k]\}$. Then, SLCA returns the result nodes $R(Q, D) = SLCA(Q, D) = \{n \in LCA(Q, D) | \nexists n' \in LCA(Q, D), n \prec n'\}$. Since each type corresponds to a schema node in $\mathbb{S}$, we extend the LCA and SLCA concepts to types as well. For instance, $lca_{\mathbb{S}}(t_{state/name}, t_{state/city}) = t_{state}$.

3 Optionality Resilience

Optionality resilience describes the desired behaviors of an XKS system when handling the missing element phenomenon. Specifically, it captures how the query result changes when an optional node is missing from the document. In this section, we first identify two types of optionality resilience, namely *result resilience* and *match resilience*, corresponding to the two steps of XKS. Next, we analyze existing XKS approaches in terms of these two properties.

3.1 Result Resilience

In an XKS query, the relevance of a result depends on the search intention as well as result matches. Since users are rarely aware of the missing element phenomenon, missing nodes should not change the search intention (*i.e.,* an XKS system should not return new results that are irrelevant to the search intention). More specifically, a missing element may make an existing result irrelevant but should not add a new result. For example, consider the documents D_1 and D_2 where $n_{0.4.3.2}$ is the only node missing in D_1 but appears in D_2. Consider now the query Q_2`(city, area)` on D_1 and D_2. The search intention is probably to find the areas of all cities. Hence, the results should include all `city` subtrees in the document with an `area` child (*e.g.,* $n_{0.3.1}$ and $n_{0.4.3}$ on D_2). However, when the `area` of $n_{0.4.3}$ is missing as in D_1, the corresponding `area` subtree becomes irrelevant. Consequently, introducing new result $n_{0.4}$ related to `state` for Q_2 on D_1 is likely to be irrelevant. Similarly, absence of the `area` of *Provo* ($n_{0.4.3.2}$) should not make `area`s of other `cities` such as *Salt Lake City* (*e.g.,* $n_{0.4.2}$) more relevant to *Provo* than *Salt Lake City*.

Definition 1 ***[Result Resilience]*** *Let Q be an* XKS *query on two documents D and D' where $D' = D \backslash \{n\}$ and n is a label match to Q. Then, an* XKS *system is called result-resilient if $R(Q, D') \subseteq R(Q, D)$.*

3.2 Match Resilience

In an XKS query, label keywords reflect the desired data in the final result tree [1,9]. Reconsider the query Q_2 on D_1 and D_2. The node $n_{0.3.1}$ is a common result of Q_2 in both these documents. The *match resilience* property asserts that the match set of $n_{0.3.1}$ should remain the same in D_1 and D_2 as $n_{0.4.3.2}$ is not a descendant of $n_{0.3.1}$. On the other hand, consider the query Q_3`(Utah, city, population, area)`. The common result of this query is the subtree rooted at $n_{0.4}$. Then, the match resilience property also asserts that the result tree of $n_{0.4}$ should be (non-strictly)

smaller in D_1 than in D_2 as the missing node $n_{0.4.3.2}$ in D_1 is a descendant of this subtree in D_2.

Definition 2 ***[Match Resilience]*** *Consider a label query $Q(w_1, \ldots, w_n)$ and two documents D, D' where $D' = D \backslash \{n\}$ and n is a label match to Q. An* XKS *system is match-resilient if for each result $r \in R(Q, D) \cap R(Q, D')$ any one of the following holds: (a) $S_{D'}(r) = S_D(r)$ if $n \notin S_D(r)$; (b) $S_{D'}(r) \subseteq S_D(r)$ if $n \in S_D(r)$.*

3.3 Optionality Resilience Analysis of Existing Approaches

Smallest Lowest Common Ancestor (SLCA). The SLCA approach introduced in [17] has been arguably the most popular technique to locate the result nodes and has become the backbone of various other approaches such as XSeek [9], XReal [1], XBridge [7], MaxMatch [10], etc. However, SLCA violates result resilience which explains its inability to handle missing elements as highlighted in Section 1. For example, consider the query Q_1(`Provo, area`) on D_2. The SLCA approach returns the node $n_{0.4.3}$ but it returns $n_{0.4}$ on $D_1 = D_2 \backslash \{n_{0.4.3.2}\}$.

Meaningful Related Lowest Common Ancestor (MLCA). The MLCA-based technique [8] returns each result tree as a group of matches, called *pattern match*, containing exactly one match for each keyword. Two nodes n_1 and n_2 that match keywords w_1 and w_2, respectively, are *meaningfully related* if there does not exist n_1' and n_2' that match w_1 and w_2 such that the LCA of n_1 and n_2 is an ancestor of the LCA of n_1' and n_2'. When a set of nodes are pair-wise meaningfully related, their LCA is called an MLCA. Each returned pattern match contains an MLCA node and its corresponding pair-wise related matches.

MLCA does not satisfy result resilience. Consider the query Q_4(`USA, Tennessee, Utah, area`) to find the areas of both *Tennessee* and *Utah*. The results on D_1 is empty since no matter which `area` node is chosen in the pattern match, it is not related to at least one value match. Specifically, node $n_{0.4.1}$ is not related to $n_{0.2.0}$ and node $n_{0.2.1}$ is not related to node $n_{0.4.0}$. Meanwhile, if $n_{0.4.1}$ is removed from the document, MLCA returns a pattern match in which $n_{0.0}$, $n_{0.2.0}$, $n_{0.4.0}$ and $n_{0.2.1}$ are matches for `USA`, `Tennessee`, `Utah` and `area`, respectively.

XSEarch. In XSEarch [2], two nodes n_1 and n_2 are considered *related* if the path between them consists of no same-label nodes except n_1 and n_2 themselves. For example, in D_1 (Figure 2), nodes $n_{0.4.3.0}$ and $n_{0.4.1}$ are considered related. A pattern match is considered related if all of its matches are related to each other. XSEarch considers two semantics, namely, *all-pair* and *star*. *All-pair* semantics requires all pairs of matches to be related. *Star*-semantics only requires at least one match which is related to all other matches.

XSEarch also violates result resilience. For Q_1, XSEarch returns same results as SLCA on D_1 and D_2. Notice that $n_{0.4.3.0}$ is considered related to both $n_{0.4.3.2}$ and $n_{0.4.1}$.

Structural Consistency. In [6], the authors propose a new constraint for the SLCA list called *structural consistency*. A list of SLCA nodes are *structural consis-*

tent when there are no ancestor-descendant relationships among them not only in instance-level (as in the original definition of SLCA) but also in schema-level. However, structural consistency is not sufficient to ensure result resilience when missing elements are involved. In fact, [6] returns same results as SLCA for Q_1 on D_1 and D_2 since there is only one SLCA node in both cases.

XSeek. A sub-task in retrieving result nodes is to deduce the result type. XSeek is a result type deduction technique built on top of SLCA [17]. Specifically, it attempts to locate entity nodes using a heuristic that entity nodes usually have same-label siblings. Then, for each SLCA node n_a, XSeek's result node is the lowest entity node n_e such that $n_e \preceq n_a$. However, XSeek still violates result resilience. For example, for Q_1, it produces identical results to SLCA on D_1 and D_2 since both $n_{0.4}$ and $n_{0.4.3}$ are entity nodes.

XReal and XBridge. XReal [1] and XBridge [7] are also result type detection techniques but based on *match statistics*. In a nutshell, XReal computes the *confidence* of a type T to be the result type by counting the number of T-typed nodes containing the matches for each query keyword but penalizes types that are too close to the root node. On the other hand, XBridge computes the result type confidence by aggregating the confidence for all results of that type. The confidence of each result is computed based on its match frequency and distribution in the query answer. Notably, the confidence value from both XReal and XBridge is zero when the result type has no results with at least one match for each keyword (*i.e.,* non-zero match frequency). Consequently, both these approaches fail to satisfy result resilience. For example, for Q_1 on D_2, the returned result type is `city` since the only result with non-zero match frequency for all keywords is of that type. On D_1, however, the returned result type is `state`. Since the result types are different, their results fail to satisfy result resilience.

AllMatch. *PathReturn* and *SubtreeReturn* are two typical techniques for the second step of XKS [4, 9, 10]. *PathReturn* returns all descendant matches of each result node and the paths connecting them while *SubtreeReturn* returns the full subtrees rooted at each result node. Since *PathReturn* and *SubtreeReturn* return the same set of matches (*i.e.,* all descendant matches) for each result node, in this paper, we refer to them as AllMatch. The result trees from AllMatch satisfy match resilience. For each result node r, all descendant matches are returned. On the other hand, when a node is removed from the document, r will not get any new descendants. Thus, no new descendant matches of r are introduced.

MaxMatch and Relaxed Tightest Fragment (RTF). While *PathReturn* and *SubtreeReturn* do not violate result resilience, they suffer from low recall and precision [9]. MaxMatch [10] and RTF [4] are introduced to address this problem. MaxMatch filters irrelevant matches under an SLCA node by returning only *contributors*. For each node n, its *contribution* is defined as the set of query keywords whose matches are descendant of n. A node is a *contributor* if its contribution is not subsumed by any of its siblings. RTF extends this notion of contributor to the so-called *valid contributor*, which is a node whose contribution is not subsumed by any of its *same-label* siblings.

MaxMatch [10] also incorporates four desirable properties of XKS by comparing XKS results when the document or query changes. However, these properties do not guarantee desirable behavior when missing elements are involved in the query. Specifically, both MaxMatch and RTF violate match resilience. For instance, consider Q_5(`Utah, city, population, area`) whose intention is probably to find the population and area of all cities of *Utah*. The only result node from [1, 9, 17] on both D_1 and D_2 is $n_{0.4}$. For $n_{0.4}$, on D_2, both MaxMatch and RTF select $n_{0.4.3.1}$ as relevant match. Notice that $n_{0.4.2.1}$, the `population` of *Salt Lake City*, is not selected since it only has `population` but not `area` whereas *Provo* city has both these elements. On the other hand, on D_1 the matches for both *Salt Lake City* and *Provo* are returned as *Provo*'s area information is missing.

4 Full SLCA (FSLCA)

Recall from Section 1, if an XML document is *full* then all missing elements are specified by empty elements. Consequently, SLCA nodes identified during keyword search on a *full* XML document will be able to satisfy optionality resilience property as there are no missing elements. Among these identified SLCA nodes, we refer to the nodes that exist in the original document as *full* SLCA (FSLCA) nodes. For example, reconsider the query Q_1 on D_1. The only SLCA node matching the query is $n_{0.4}$ which is undesirable. Document D_3 (Figure 4) displays the full version of D_1. If we now compute the SLCA nodes on D_3 for Q_1, it would produce $n_{0.4.3}$ since its descendant node $n_{0.4.3.0}$ matches to `Provo` and node $n_{0.4.3.2}$ matches to `area`. Since $n_{0.4.3}$ also exists in D_1, it is an FSLCA of Q_1. We now formally introduce the concepts of *full* XML document and FSLCA.

4.1 Full XML Document

Definition 3 ***[Full Node]*** *Given a document D, a node $n \in D$ with type t is full, denoted as $FullNode(n)$, if for each child type t_c of t, n has at least one child node with type t_c.*

For example, in D_1, $n_{0.3.1}$ is a full node since type `city` has three child types, namely `name`, `population` and `area`, and $n_{0.3.1}$ has corresponding child nodes $n_{0.3.1.0}$, $n_{0.3.1.1}$ and $n_{0.3.1.2}$, respectively.

Definition 4 ***[Full Document]*** *An XML document D is considered full, denoted as $F(D)$, iff all of its nodes are full. That is, $\forall n \in D$, $FullNode(n) =$ `true`.*

For example, D_3 in Figure 4 is a full document. Noticeably, a document D can be transformed to a full document $F(D)$ by adding empty elements as children to nodes that are not full in D. For instance, D_1 can be transformed to D_3 by adding empty elements (in dotted rectangles). Observe that there are infinite numbers of full documents $F(D)$ of D. For example, adding any numbers of `area` nodes as

siblings of $n_{0.4.3.2}$ would create another full document of D_1. In practice, we only consider the *minimal* full document of D.

Definition 5 ***[Minimal Full Document]*** *Given an* XML *document D, a full document $F(D)$ is minimal, denoted as $F_{min}(D)$ iff the following conditions are true: (a) $\forall n \in D$, $n \in F_{min}(D)$; (b) for each $n' \in F_{min}(D)$, n' is either in D or removing it would make $F_{min}(D)$ no longer a full document; and (c) all nodes in $F_{min}(D) \backslash D$ are empty nodes.*

For example, D_3 in Figure 4 is a minimal full document of D_1. In the sequel, for brevity, we shall use $F(D)$ or F instead of $F_{min}(D)$ when the context is clear. Clearly, given a document D, the DataGuides of D and $F(D)$ are identical.

4.2 FSLCA Definition

Since a full document may contain empty nodes that do not exist in the original document, each FSLCA node belongs to one of the following two categories:

- *Category 1*: Both the FSLCA node *and* its matches are in the original document D. For example, consider the query Q_2(`area`, `city`) on D_1. Then the FSLCA node $n_{0.3.1}$ belongs to this category as its subtree includes matches for both `city` and `area`.
- *Category 2*: The FSLCA node is in D but its matches *are not* in D. For example, for Q_2 on D_1, FSLCA nodes $n_{0.4.2}$ and $n_{0.4.3}$ belong to this category as both do not have any `area` element as descendant.

In this paper, we refer to the result sets consisting of only Category 1 FSLCA nodes as *complete* FSLCA nodes while the results sets consisting of both the aforementioned categories are called *partial* FSLCA nodes. Formally,

Definition 6 ***[Partial & Complete*** FSLCA ***]*** *Given a query Q on a document D,*

- *the set of partial* FSLCA *nodes of Q on D, denoted as $PFSLCA(Q, D)$ is defined as, $PFSLCA(Q, D) = SLCA(Q, F(D)) \cap D$.*
- *the set of complete* FSLCA *nodes of Q on D, denoted as $CFSLCA(Q, D)$ is defined as, $CFSLCA(Q, D) = PFSLCA(Q, D) \cap SLCA(Q, D)$.*

Remark. The complete and partial FSLCA s are designed to allow two different strategies of returning result nodes containing missing elements. The most obvious approach is to ignore result nodes containing missing elements, which is realized by complete FSLCA. For example, consider the query Q_2`(city,area)` on D_1. Here, $CFSLCA(Q_2, D_1) = \{n_{0.3.1}\}$ as $n_{0.4.2}$ and $n_{0.4.3}$ do not have `area` element as descendant. Alternatively, partial FSLCA does not eliminate such results nodes and returns them explicitly indicating the missing elements. For example, $PFSLCA(Q_2, D_1) = \{n_{0.3.1}, n_{0.4.2}, n_{0.4.3}\}$ and the `area` elements in $n_{0.4.2}$ and $n_{0.4.3}$ are represented as empty nodes (an empty node is represented as a box with a dashed line border).

4.3 Proofs for Optionality Resilience

Theorem 1 *Both partial and complete* FSLCA *nodes satisfy result resilience.*

Proof 1 *For any pairs of document D and $D' = D\backslash\{n\}$ where $n \in D$, we have $F(D) = F(D')$. Therefore, $SLCA(Q, F(D)) = SLCA(Q, F(D'))$ for any query Q. Since $D' \subset D$, $PFSLCA(Q, D') \subseteq PFSLCA(Q, D)$.*

For all result nodes $r \in CFSLCA(Q, D')$, r has descendant matches to all keywords in D'. Meanwhile, since $D \subseteq F(D) = F(D')$, there are no results in $SLCA(Q, D)$ that is a descendant of r. Thus, $r \in SLCA(Q, D)$. But $r \in SLCA(Q, F(D')) = SLCA(Q, F(D))$. Therefore, $r \in CFSLCA(Q, D)$.

The following lemmas prove that existing XKS approaches built on top of FSLCA nodes instead of SLCA nodes satisfy the optionality resilience property.

Lemma 1 *The results of XSeek on top of either partial or complete* FSLCA *nodes satisfy result resilience.*

Proof 2 *Let n_0 be a node in document D. Let $xseek(n_0)$ be a function returning the lowest ancestor entity node of n_0 as described in Section 3.3. Since $PFSLCA(Q, D') \subseteq PFSLCA(Q, D)$ and $CFSLCA(Q, D') \subseteq CFSLCA(Q, D)$, their images through function $xseek()$ maintain inclusion relation $\subseteq$.*

Lemma 2 *If matches to empty elements are considered, the results of MaxMatch and* RTF *on top of either partial or complete* FSLCA *nodes satisfy match resilience.*

Proof 3 *Let Q be a query on two documents such that $D' = D\backslash\{n\}$, $n \in D$ and n is a label match of Q. Let r be a result node and r_0 be a node within the result tree rooted at r. If n is not a descendant of r_0, its contribution is unchanged in D and D'. If n is a descendant of r_0, r_0 would have at least one label match with same label with n in $F(D)$ and $F(D')$. Therefore, the contribution of r_0 is unchanged for both MaxMatch and* RTF.

5 FSLCA Computation

In this section, we present two variants of the MESSIAH algorithm, namely P-MESSIAH and C-MESSIAH, to compute partial and complete FSLCA nodes, respectively. These algorithms can retrieve FSLCA nodes without the need of physically generating full documents. Also, it is not necessary for the XML document to be accompanied by its schema or DTD as a DataGuide tree can be easily generated from a "schemaless" XML document [3]. Note that such DataGuide tree has also been used in several prior work related to XKS [1,7,9]. We first summarize the principle behind existing SLCA computation approach which serves as a foundation for the FSLCA computation problem.

5.1 Principles for SLCA Computation

The core idea behind all existing SLCA computation techniques can be summarized by the following lemmas (see [17] for full proofs).

Lemma 3 *Given a node $n \in D$ and a query keyword w, the level of the* SLCA *between n and matches of w on D, denoted as $slcaLvl(n, D, w)$, is equal to $max(lvl(lca(n, lm(n, M_D(w)))), lvl(lca(n, rm(n, M_D(w)))))$ where $lvl()$ returns the level of a node and $lm(n, M_D(w))$ and $rm(n, M_D(w))$ return the last node before n and the first node after n in $M_D(w)$, respectively.*

Lemma 4 *Given a node $n \in D$ and keywords $w_1, \ldots, w_k$, the level of the* SLCA *between n and matches of $w_1, \ldots, w_k$ on D, denoted as $slcaLvl(n, D, w_1, \ldots, w_k)$, is equal to $min(slcaLvl(n, D, w_1), \ldots, slcaLvl(n, D, w_k))$.*

Lemma 3 focuses on a special case of SLCA computation where there are only two sets in which one set is a single element n. It reduces the SLCA into the LCA of n to either $lm()$ or $rm()$. Lemma 4 extends this results to multiple sets. For brevity, for a query $Q = \{v_1, \ldots, v_k\}$, we shall denote $slcaLvl(n, D, w_1, \ldots, w_k)$ as $slcaLvl(n, D, Q)$ and the ancestor of n at level $slcaLvl(n, D, Q)$ as $slca(n, D, Q)$.

The complete SLCA candidate list is then computed by concatenating $slca(n, D, Q)$ for all nodes n of a set of matches M. The set M is selected so that, for all SLCA node s, there exists at least one match m in M such that $s \preceq m$ (see [17] and [14] for more details). We call the set satisfying this property an "*anchoring* match set". The difference between [17] and [14] lies on how to get the anchoring match set M. [17] chooses $M = M_D(w_1)$ in which w_1 is the keyword with smallest match set whereas [14] reduces M further by exploiting several optimizations. Meanwhile, [19] considers all ancestor-or-self of $M_D(w_1)$ as M but uses equality comparison instead of $lm()$ and $rm()$.

Finally, the SLCA candidate list is then validated using ancestor-descendant relationship following Lemma 5 (here, we show a more general lemma compared to the lemma used in [17] and [14]).

Lemma 5 *For a query Q on document D, given two sets of* SLCA *candidates S_1, S_2 such that $SLCA(Q, D) \subseteq S_1, S_2 \subseteq LCA(Q, D)$,*

$$SLCA(Q, D) = \{s \in S_1 | \nexists s' \in S_2, s \prec s'\}$$

Proof 4 *It follows the definition of* SLCA. *Notice that $\preceq$ is a partial order in $LCA(Q, D)$ and $SLCA(Q, D)$ is its minimal element set.*

Both [17] and [14] exploit a non-indexed and an indexed approach to support $lm()$ and $rm()$. The indexed approach uses random access to retrieve $lm()$ and $rm()$ from a B-tree, which is suitable when M is small. When M is large, non-indexed approach using sequential scan is more suitable. For all cases, the time complexity is $O(dk|M|log(|M_{max}|)$ where d is the document depth and M_{max} is the largest match set.

5.2 Key Strategies behind MESSIAH

The key idea of FSLCA computation using MESSIAH is based on the following two observations. First, SLCA or FSLCA computation only requires *checking* whether a node has descendant matches but not *retrieving* the matches directly. Note that existing approaches compute SLCA by retrieving the matches first and then computing SLCA from them. However, this approach cannot be used in MESSIAH as some matches are "virtual" (missing elements) and cannot be retrieved. Second, since a full document does not have missing nodes, we can infer the subtree structure of a node by analyzing its type only. For instance, in a full document, we can infer that each `state` has some `city` as child and each `city` has `name`, `population` and `area` as children. These two observations are materialized in the following theorem.

Theorem 2 *In a minimal full document F, given a node n with type t, n has a descendant label match n' to a keyword w iff there exists a type match t' to w such that $t \preceq_{\mathbf{S}} t'$.*

Proof 5 *It follows from the definition of minimal full document (Definition 5).*

Corollary 1 *Let F be the minimal full document of D. Given node $n \in D$ and a query keyword w, $slcaLvl(n, F, w) = max(lvl(lca(n, lm(n, M_D(w)))), lvl(lca(n, rm(n, M_D(w)))), slcaLvl(t, \mathbf{S}, w))$ where t is the type of n and $slcaLvl(t, \mathbf{S}, w)$ is the level of the SLCA between t and all type matches of w in $\mathbf{S}$.*

Corollary 2 *Given a node $n \in D$, minimal full document F, and keywords $w_1, \ldots, w_k$, $slcaLvl(n, F, w_1, \ldots, w_k) = min(slcaLvl(n, F, w_1), \ldots, slcaLvl(n, F, w_k))$.*

Corollary 1 and 2 are useful generalization of Lemma 3 and 4, respectively. Observe that the left-hand-sides of the equations in these corollaries leverage F while the matches are retrieved from D (*i.e.,* $M_D(w)$). Importantly, as DataGuides for most practical datasets are small [1], $slcaLvl(t, \mathbf{S}, w)$ can be computed efficiently (can even be pre-computed). In the following subsections, we shall denote $slca(n, F, Q)$ and $slcaLvl(n, F, Q)$ as $fslca(n, D, Q)$ and $fslcaLvl(n, D, Q)$, respectively.

5.3 Partial FSLCA Computation

Partial FSLCA computation needs to address a novel challenge compared to existing SLCA computation techniques. Observe that Lemma 3-4 and Corollary 1-2 assume that each SLCA/FSLCA node has a match n in D (called *anchor node* in [14]). However, some partial FSLCA nodes may not have any match in D. To address this challenge, we split the partial FSLCA computation problem into two cases, namely FSLCA node with at least one value match and FSLCA node with only label match, and address each them in turn.

Case 1: FSLCA with at least one descendant value match. Corollary 1 and 2 can be applied for Case 1 since all value matches are in D (converting D to F is

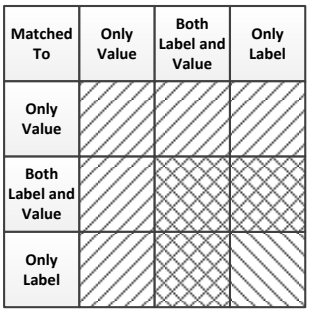


Figure 5: This table shows when the two cases L_1 and L_2 are considered for a query of 2 keywords. Each keyword can match to only labels, only values or both labels and values. NE-SW diagonal line means L_1 is computed while NW-SE diagonal line means L_2 is considered.

not necessary). However, since the value matches can be matches of any keywords, the set of anchor nodes is $V_1 \cup \cdots \cup V_k$ instead of M as used in SLCA computation. Formally, for query Q on D, the set of FSLCA candidates in case 1, denoted as L_1, is:

$$L_1 = \{fslca(n, D, Q) | n \in \bigcup_{i=1}^{k} V_i\}$$

Case 2: FSLCA with at least one label match to each keyword. When a partial FSLCA node n_a has at least one descendant label match n_ℓ to each keyword, it may not have any match in the original document D. Consequently, locating anchor nodes is challenging. We propose a technique to retrieve the candidate node n_a *without* using anchors.

Based on Theorem 2, n_a has a descendant label match to keyword w if and only if its type t_a has a descendant type match to w. Therefore, t_a must have at least one descendant type match to each keyword. In other words, t_a is an SLCA of the type matches of all keywords w (*i.e.,* given a query $Q(w_1, \ldots, w_k)$ on D, $t_a \in SLCA_{\mathbf{S}}(T_D(w_1), \ldots, T_D(w_k))$). Thus, to retrieve n_a, we can find the set T_a of all t_a and then retrieve all corresponding n_a in D (i.e. $N_D(T_a)$). Notice that this strategy does not require any anchors. Furthermore, it is extremely efficient since label matches are not retrieved and the number of possible values of t_a is small. Formally, for query $Q(w_1, \ldots, w_k)$ on D, the set of FSLCA candidates in case 2, denoted as L_2, is:

$$L_2 = N_D(T_a)$$

where

$$T_a = SLCA_{\mathbf{S}}(T_D(w_1), \ldots, T_D(w_k))$$

Figure 5 illustrates when the two cases are considered for a query of two keywords. Importantly, although there are some combinations where both cases need to be considered, in most of the times, only one case is considered. Thus, while our algorithm needs to consider the more general and complete situation, in practice, the algorithm is usually much simpler and potentially more efficient. Note that in practice only few keywords (if any) in a dataset match *both* values and labels.

Finally, we shall provide the theoretical background of our technique to compute $PFSLCA(Q, D)$ in the general situation with both cases. First, as each SLCA node in F must contain either at least one label match to each keyword or at least one value match, it is obvious that:

$$SLCA(Q, F) \subseteq L_1 \cup N_F(T_a)$$

Following Lemma 5,

$$SLCA(Q, F) = \{s \in L_1 \cup N_F(T_a) | \nexists s' \in L_1 \cup N_F(T_a), s \prec s'\}$$

From Definition 6,

$$\begin{aligned} PFSLCA(Q, D) &= SLCA(Q, F) \cap D \\ &= \{s \in L_1 \cup N_D(T_a) | \nexists s' \in L_1 \cup N_F(T_a), s \prec s'\} \\ &= \{s \in L_1 \cup L_2 | \nexists s' \in L_1 \cup N_F(T_a), s \prec s'\} \end{aligned} \quad (1)$$

The P-MESSIAH Algorithm. Algorithm 1 outlines the procedure of P-MESSIAH which realizes Equation 1. Observe that the inputs are value matches and type matches for Q which can be retrieved directly from D. Lines 2-3 retrieves L_2 (Case 2). All candidate nodes generated for this case (*i.e.,* having at least one descendant label match for each keyword) are stored in $Cand_2$. The $Anchor_1$ stores all value matches for Q which shall serve as anchor matches. The **merge()** function in Lines 3 and 4 merges the input sets into a single sorted set of nodes and returns them in stream format. Note that if all of the input sets are sorted stream, the **merge()** function can be implemented efficiently without sorting and consumes minimal memory. Both $Anchor_1$ and $Cand_2$ are required to be sorted for pruning which we shall discuss later. The symbols a_1 and c_2 denote the current cursor node of streams $Anchor_1$ and $Cand_2$, respectively. They are assumed to be null when there are no more nodes in the stream.

Lines 7-14 materialize L_1 (Case 1). The loop in Lines 7-22 is executed for each anchor node a_1 in $Anchor_1$. Lines 10-14 use Corollary 1 and 2 to compute $fslca(a_1, D, Q)$. Notably, instead of using $lm()$ and $rm()$ functions, we use **peekLast**(i) and **peekNext**(i) which return the last and next node, respectively, in the i-th input stream (*i.e.,* V_i) in $Anchor_1$. Although the latter functions return same results as $lm(a_1, V_i)$ and $rm(a_1, V_i)$, respectively, **peekLast**(i) and **peekNext**(i) are more efficient since they can be easily supported in streams by storing the last retrieved node and looking ahead the next retrieved node of each input stream.

Algorithm 1: The P-MESSIAH Algorithm.

Input: A query Q with k keywords $w_1 \dots w_k$
Input: The list of value matches V_i and type matches T_i for each w_i in document D whose Dataguide is $\mathbf{S}$
Output: The list of FSLCA nodes of Q on D

```
1  Result = ∅;
2  T_a = SLCA_S(T_1, ..., T_k);
3  Cand_2 ← merge({N_D(t) | t ∈ T_a});
4  Anchor_1 ← merge(V_1, ..., V_k);
5  c'_1 ← root;
6  c_2 ← Cand_2.next();
7  while there are more nodes in Anchor_1 do
8      a_1 ← Anchor_1.next();
9      t_1 ← the type of a_1;
10     for i = 1 → k do
11         if a_1 ∉ V_i then
12             ℓ_i = max(lvl(lca(a_1, Anchor_1.peekLast(i))),
               lvl(lca(a_1, Anchor_1.peekNext(i))), slcaLvl(t_1, S, w_i));
13     ℓ = min_{1≤i≤k}(ℓ_i);
14     c_1 = ancestor(a_1, ℓ);
15     if type(c_1) is not an ancestor type of a type in T then
16         if id(c'_1) ≤ id(c_1) then
17             if c'_1 ⋠ c_1 then
18                 Result = Result ∪ {c'_1};
19             while c_2 is not null and id(c_2) ≤ id(c_1) do
20                 if c_2 ⋠ c_1 then
21                     Result = Result ∪ {c_2};
22                 c_2 ← Cand_2.next();
23             c'_1 ← c_1;
24 Result = Result ∪ {c'_1};
25 while c_2 is not null do
26     Result = Result ∪ {c_2};
27     c_2 ← Cand_2.next();
28 return Result
```

Specifically, they take $O(1)$ time compared to $O(log(V_i))$ time of $lm(a_1, V_i)$ and $rm(a_1, V_i)$.

Lines 15-23 check whether the candidate nodes c'_1 and c_2 can be partial FSLCA nodes. Specifically, Line 15 ensures that the candidate c'_1 has no descendant nodes in $N_F(T_a)$. Since c'_1 is also in F, such descendant checking can use type (Theorem 2). Meanwhile, Line 17 ensures that c'_1 has no descendant nodes in L_1. As $c'_1 \in L_1$, c'_1 only needs to be checked with the next node in L_1. Similarly, Lines 19-22 validates all c_2 between c'_1 and c_1 against c_1 to ensure that they do not have

descendant candidate in L_1. Notice that nodes in $N_F(T_a)$ do not have $\prec$ between themselves so that validating c_2 against $N_F(T_a)$ is unnecessary. Lines 24-27 are similar to Lines 15-23 to evaluate the last c'_1 and c_2.

Time Complexity. Let T be the result returned in Line 2 and d be the document depth. Then the time complexity of Algorithm 1 is $O(dk\sum_{1\leq i\leq k}|V_k| + dlog(k)\sum_{t\in T}|N_D(t)|)$. These two terms correspond to Case 1 and 2, respectively. In particular, Case 1 produces $O(\sum_{1\leq i\leq k}|V_k|)$ candidate nodes, each takes $O(dk)$ time (due to Line 12). Case 2 produces $O(\sum_{t\in T}|N_D(t)|)$ candidates, each takes $O(d\log(k))$ time (due to the **merge** function in Line 3).

Remark 1. Since the complexity of ILE algorithm [17] is $O(kd|M_1|log(|M_k|)$, it may seem that our complexity is worse. However, notice that the result size between two cases are not equal. In practice, as demonstrated in Section 6, P-MESSIAH outperforms ILE. First, the ILE algorithm uses $lm()$ and $rm()$ functions with random access while our algorithm exploits stream extensively without random access. Second, for most keywords w, $L(w) \gg V(w)$. Hence, $\sum_{1\leq i\leq k}|V_k|$ is potentially much smaller than $|M_1|log(|M_k|)$. Third, the $dlog(k)\sum_{t\in T}|N_D(t)|$ term only appears when $T(w_i) \neq \emptyset \forall i$.

Remark 2. P-MESSIAH follows an "eager" strategy *i.e.,* FSLCA nodes are returned in document-order and the first output node can be returned even before all of input nodes are read. This property is highly desirable in practice since it greatly reduces query response time (*i.e.,* the time the users need to wait to view the first result).

Example 1 Consider Q_2(`city,area`) on D_1. We have V_{D_1}(`city`) = $\{n_{0.4.2.0}\}$, T_{D_1}(`city`) = $\{t_{city}\}$, V_{D_1}(`area`) = $\emptyset$, T_{D_1}(`area`) = $\{t_{territory/area}, t_{state/area}, t_{city/area}\}$. Using the DataGuide in Figure 3(a), Line 2 returns t_{city}. Thus, $Cand_2 = \{n_{0.3.1}, n_{0.4.2}, n_{0.4.3}\}$. Meanwhile, $Anchor_1 = \{n_{0.4.2.0}\}$ so that the first and only value for a_1 is $n_{0.4.2.0}$. For this a_1, $\ell = 3$ and $c_1 = n_{0.4.2}$. The conditions in Line 15 and 16 are satisfied but the one in Line 17 is not. As $c_2 = n_{0.3.1}$ precedes $c_1 = n_{0.4.2}$, the while-loop in Lines 19-22 proceeds to check Line 20 condition and adds $n_{0.3.1}$ as the first result. However, when $c_2 = n_{0.4.2}$, it is not added as $c_2 = c_1$ (fails Line 20 condition). Next, $c_1 = n_{0.4.2}$ is assigned to c'_1 which is added to the result at Line 24. Finally, Lines 25-26 add the remaining node in $Cand_2$, $n_{0.4.3}$, as results. The final results is $\{n_{0.3.1}, n_{0.4.2}, n_{0.4.3}\}$. ∎

5.4 Complete FSLCA Computation

From Definition 6, we have:

$$\begin{aligned} CFSLCA(Q,D) &= SLCA(Q,D) \cap PFSLCA(Q,D) \\ &= SLCA(Q,D) \cap SLCA(Q,F) \end{aligned}$$

From Lemma 5,

$$SLCA(Q,F) = \{s \in LCA(Q,F) | \nexists s' \in L_1 \cup N_F(T_a), s \prec s'\}$$

Algorithm 2: The c-messiah Algorithm.

Input: A query Q with k keywords $w_1 \dots w_k$
Input: The matches M_i in document D
Input: The value matches V_i in document D
Input: The DataGuide tree $\mathbb{S}$ of D
Output: The list of complete fslca nodes of Q on D

```
1  Result = ∅;
2  Choose M as an anchoring match set for Q in D;
3  T_a = SLCA_S(T_1, ..., T_k);
4  c' ← root;
5  while there are more nodes in M do
6      a ← M.next();
7      Compute c = slca(a, D, Q);
8      if type(c) is not an ancestor type of a type in T then
9          if id(c') ≤ id(c) then
10             if c' ⋠ c and checkDescFSLCA(Q,c,V_1,...,V_k,T_1,...,T_k)=False then
11                 Result = Result ∪ {c'};
12             c' ← c;
13 if checkDescFSLCA(Q,c,V_1,...,V_k,T_1,...,T_k)=False then
14     Result = Result ∪ {c'};
15 return Result
```

Therefore,

$$\begin{aligned} CFSLCA(Q,D) &= SLCA(Q,D) \cap \{s \in LCA(Q,F) | \nexists s' \in L_1 \cup N_F(T_a), s \prec s'\} \\ &= \{s \in SLCA(Q,D) \cap LCA(Q,F) | \nexists s' \in L_1 \cup N_F(T_a), s \prec s'\} \end{aligned}$$

Since $SLCA(Q,D) \subseteq LCA(Q,F)$, $SLCA(Q,D) \cap LCA(Q,F) = SLCA(Q,D)$,

$$CFSLCA(Q,D) = \{s \in SLCA(Q,D) | \nexists s' \in L_1 \cup N_F(T_a), s \prec s'\} \quad (2)$$

The c-messiah Algorithm. The c-messiah algorithm is outlined in Algorithm 2. Similar to [17], each anchor node a in the anchoring match set M is processed one-by-one (Lines 4-7) and the computed candidate c is pruned based on document order and $\prec$ relation (Lines 9-10). Our algorithm can work on any M satisfying the anchoring set condition mentioned in Section 5.3. The differences between slca and c-messiah algorithm lie on the validating of each slca candidate at Line 8 and 10 following Equation 2. Line 8 ensures that the candidate node does not have any descendants in $N_F(T_a)$ following Theorem 2. Notice that we do not need to retrieve either $N_F(T_a)$ or $N_D(T_a)$. Line 10 is used to ensure that all candidates do not have a descendant in L_1 (see Section 5.3) using the *checkDescFSLCA()* procedure described in Algorithm 3.

A naïve way to realize *checkDescFSLCA()* procedure is to compute all (partial) fslca in L_1 as in Algorithm 1 and then validate each candidate against these nodes.

Algorithm 3: The *checkDescFSLCA* algorithm.

Input: A query Q with k keywords $w_1 \dots w_k$
Input: A candidate c
Input: The list of value matches V_i and type matches T_i of each w_i in document D whose DataGuide is $\mathcal{S}$
Output: The list of complete FSLCA nodes of Q on D

```
1 for each i from 1 to k do
2     for each a_i ∈ V_i such that c ≤ a_i do
3         Compute c' = ancestor(a_i, lvl(c) + 1);
4         if c'.keywords is null then
5             Initialize c'.keywords by finding all keywords w_j such that
              ∃t ∈ T_j, type(c') ≤ t;
6         Add i to c'.keywords;
7         if c'.keywords = Q then
8             return True
9 return False
```

Obviously, it is expensive. To eliminate the FSLCA computation cost, we leverage the fact that a node has a descendant partial FSLCA node if and only if one of its children matches to all query keywords in F. In particular, for each child c' of the candidate c (Line 3), we first calculate its descendant label matches based on its type (through Theorem 2) (Lines 4-5) and then expand its matching keywords by traversing the value match lists (Line 6). Once a child contains matches to all keywords (Line 7), the candidate c must contain at least one descendant partial FSLCA node in L_1.

Time Complexity. Assuming that all computations on **S** can be computed in $O(1)$ since **S** is usually much smaller than D, the time complexity of Algorithm 3 is $O(d|N|)$ where d is the document depth and N is the number of value matches underneath the candidate. Then, in the worst case, the time complexity of Algorithm 2 is $O(dk|M|\log(|M_{max}|) + d(\sum|V_i|))$ where M_{max} is the largest match set.

Remark 1. From the definition of complete FSLCA nodes (see Definition 6), it is natural that C-MESSIAH algorithm costs about the combined cost of SLCA and P-MESSIAH algorithm. However, with some optimizations, the cost overhead, especially from P-MESSIAH algorithm, is minimized. Specifically, first, unlike in Algorithm 1, candidates from L_2 (i.e. $N_D(T_a)$) are never retrieved. Second, only value matches descendant of an SLCA candidate at Line 10 of Algorithm 2 are used. Third, the value match lists are not merged into a stream as document order among them is not required.

Remark 2. At first glance, it seems that C-MESSIAH introduces an overhead of $O(d(\sum|V_i|))$ to the SLCA algorithm. However, in practice, such overhead is usually small as demonstrated in Section 6. As mentioned in Section 5.3, the value match set V_i is usually small as the useful value keywords are usually selective and not

all value matches are used. Furthermore, in fact, in some cases, c-MESSIAH even outspeeds SLCA. The reason is that, using DataGuide and the type checking in Line 8, some SLCA candidates are pruned sooner, minimizing the validating cost.

Remark 3. c-MESSIAH follows an "eager" strategy *i.e.,* results nodes are returned in document-order and the first output node can be returned even before all of input nodes are read.

Remark 4. Algorithm 2 is proposed to compute complete FSLCA nodes for all queries and all methods of choosing M. However, its cost can be dramatically reduced for a majority of queries by choosing suitable M. Specifically, if the query Q contains at least one keyword w_i without optionality (*i.e.,* w_i matches to either only value matches or non-missing labels or, formally, $M_D(w_i) = M_F(w_i)$), by choosing $M = M_D(w_i)$, the only match to w_i in F within $slca(a, D, Q)$'s subtree is the anchor node a itself. Therefore, Algorithm 3 is reduced to evaluate whether $slca(a, D, Q) = fslca(a, D, Q)$ and the condition in Line 8 of Algorithm 2 is no longer necessary. Motivated by this observation, in our experiment, we shall select M to be the matches of the most selective keyword without optionality. If such keyword does not exist, we then choose the most selective keyword.

Example 2 Consider Q_2`(city,area)` on D_1. Let's choose $M = M_{D_1}$`(city)` $= \{n_{0.3.1}, n_{0.4.2}, n_{0.4.2.0}, n_{0.4.3}\}$. Choosing these nodes at anchor nodes, the list of SLCA candidates computed by Line 7 are $(n_{0.3.1}, n_{0.4}, n_{0.4}, n_{0.4})$. Similar to Example 1, $T_a = \{t_{city}\}$. Thus, $n_{0.4}$ is filtered out by the condition in Line 8 so that only $n_{0.3.1}$ is the remaining candidate. Evaluating $n_{0.3.1}$ using Algorithm 3, as the only value match of the query $n_{0.4.2.0}$ is not a descendant of $n_{0.3.1.2}$, Algorithm 3 returns *False* for $n_{0.3.1}$ and it is our only complete FSLCA node.

5.5 Heuristics-based Algorithm Selection

Recall that c-MESSIAH ignores result nodes containing missing elements (returns complete FSLCA) whereas p-MESSIAH does not eliminate such results nodes and returns them explicitly indicating the missing elements as empty nodes (returns partial FSLCA). That is, p-MESSIAH returns all complete FSLCA nodes of c-MESSIAH as well as additional results containing missing elements. Given a query Q, how can we automatically deduce which variant of MESSIAH needs to be executed? In this section, we present a heuristics-based selection strategy to answer this question. Specifically, our heuristic is based on the statistics of underlying XML data and not on their semantics.

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 FSLCA s as well as the number of results (denoted by N) desired by a user. In the context of keyword search, N is typically small. So if an XKS system returns more than N complete 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 complete 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 complete FSLCA s *apriori*. We address it by adopting a lightweight version of the *XSketch*-index [12]. In short, an *XSketch*-index is a directed acyclic graph synopsis G where each *synopsis node* g represents a set of data nodes with 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. For example, a synopsis node g_{city} representing all `city` elements in D_1 will have the structural distribution of 100% for `name` and `population` but 33% for `area` since all `city` elements have `name` and `population` but only one out of three has `area`. Similarly, for the $g_{city/area}$ synopsis node representing all `city/area` elements in D_1, the value distributions of `Houston`, `Salt`, `Lake`, `City`, and `Provo` are all 33%.

We use a *lightweight* version of *XSketch*-index where the synopsis graph G is represented using the DataGuide tree $\mathbb{S}$. Specifically, each synopsis node corresponds to exactly one schema node in $\mathbb{S}$. Note that G can be built while computing $\mathbb{S}$ and stored with it. It is worth mentioning that although a more detailed *XSketch*-index would provide a better estimation, it takes more space. A detailed discussion on the space-accuracy trade-off is provided in [12]. As we shall see later, such lightweight version is sufficient to select the correct variant of MESSIAH with high accuracy.

To illustrate the estimation process, let us reconsider Q_1 and Q_2 on D_1. For Q_1(`Provo area`), from the *XSketch*-index, all `cities` have `name` but only 33% of `name` have value `Provo`. Meanwhile, only 33 % of `cities` have `area`. Assuming independent distribution between the two, 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_2(`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_2. This estimation reflects the intuition that Q_1 is more likely to have no relevant results in the document so results with missing elements are desirable.

6 Experimental Study

We conducted experiments to compare performance of MESSIAH against state-of-the-art SLCA computation approaches, namely, SE and ILE [17]; and IMS and IIMS [14]. For complete FSLCA, we also provide two implementations, non-indexed (denoted by NC-MESSIAH) and indexed (denoted by IC-MESSIAH), corresponding to two methods to realize *lm*() and *rm*() functions as in [14, 17]. Notice that partial FSLCA does not use *lm*() and *rm*(), so we only provide one implementation (P-MESSIAH). All techniques are implemented using Java 1.7 on top of Berkeley DB 4.0.103. The experiments were performed on an Intel Xeon X5570 machine with 4GB memory.

Id	Queries	Num. of Complete FSLCA	Num. of Partial FSLCA	Num. of SLCA
QM1	London	4	4	4
QM2	California Arizona	1	1	1
QM3	country Muslim	98	98	98
QM4	country name Laos	1	1	1
QM5	Fresno longitude	0	1	1
QM6	York latitude	1	5	3
QM7	city longitude latitude	801	3111	801
QM8	river Lualaba area length	0	4	1
QP1	Kringle	38	38	38
QP2	Kringle Domain	8	8	8
QP3	journal Nature	1572	1572	1572
QP4	publication 2002 Science	281	281	281
QP5	Filaggrin parent_list	0	1	1
QP6	publication journal year	40292	41950	40292
QP7	Desmoglein interpro Pemphigus class_list	2	4	2
QP8	Peptidase S1A prothrombin Patthy sec_list	1	3	1
QD1	XML	7913	7913	7913
QD2	Torsten Grust	46	46	46
QD3	Aradhye title	16	16	16
QD4	XML Keyword Search inproceedings	50	50	50
QD5	inproceedings cite	6374	872535	6374
QD6	Aradhye inproceedings cite	0	11	1
QD7	volume XML book	7	13	7
QD8	XQuery XIME proceedings cite	0	4	1
QS1	Hamlet	469	469	469
QS2	Juliet speech	173	173	173
QS3	to be or not to be speaker	659	659	659
QS4	wherefore art thou Romeo Act	3	3	3
QS5	King Lear Tragedy prologue	0	1	1
QS6	play title subtitle	1	125	1
QS7	Othello Act IV epilogue	0	2	1
QS8	Troilus Cressida Act VI prologue epilogue	0	1	1

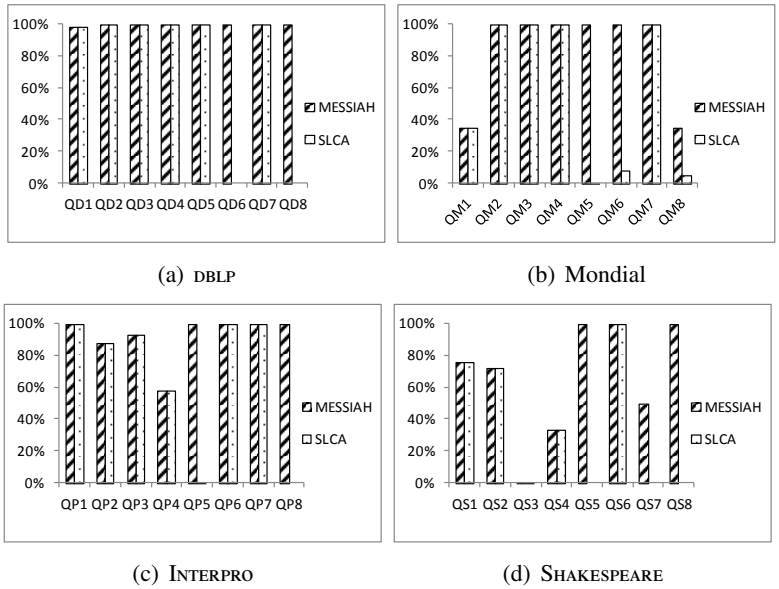
Figure 6: Query set.

Since the techniques to select relevant return nodes within the SLCA subtrees (*e.g.,* [1,9,10]) are orthogonal to MESSIAH, we do not compare them.

6.1 Experimental Setup

Dataset. The experiments are performed on four XML datasets, Mondial (1.72MB), INTERPRO (69MB), DBLP (740MB) and SHAKESPEARE (9.1MB). Mondial is a data-centric XML dataset with many short texts while SHAKESPEARE is a text-centric XML document consisting of mostly long texts. INTERPRO and DBLP are datasets with both short and long text. Note that these data sources are also used for empirical study in prior works [1,9,10,17].

Queryset. The set of queries studied for each dataset is reported in Figure 6. The queries for Mondial, INTERPRO, DBLP, and SHAKESPEARE are denoted as QM1-QM8, QP1-QP8, QD1-QD8, and QS1-QS8, respectively. They are selected as follow. We employ ten unpaid volunteers who have knowledge of XML but are not involved in this project. Since our focus is on queries involving missing elements, for each volunteer, we ask them to generate four queries without missing labels and four with missing labels for each dataset. The missing labels for each dataset are pro-



(a) DBLP (b) Mondial

(c) INTERPRO (d) SHAKESPEARE

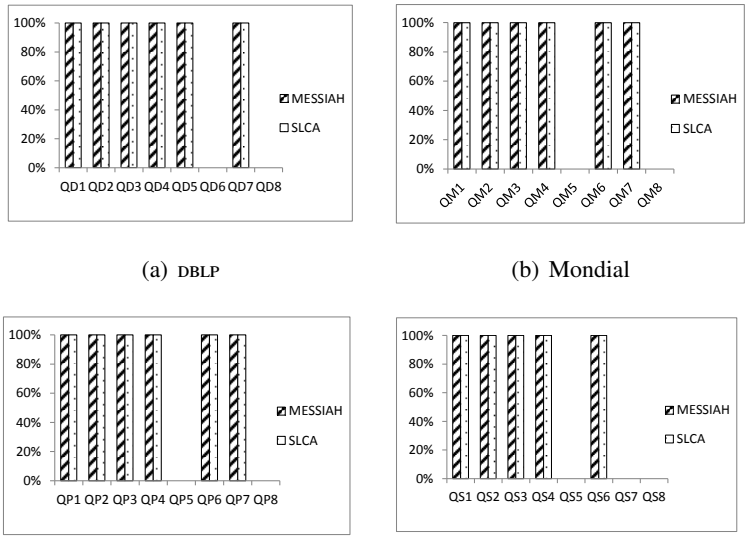
Figure 7: Precision.

vided beforehand. For each dataset, with 80 sample queries, we keep 8 queries with diversity in numbers of keywords, keyword selectivity and whether the keyword matches to a label or a value for experiments. The first four queries do not contain missing elements whereas the last four queries do. Thus, the result size of each of the first four queries is identical for all benchmark approaches. Observe that the result sizes of complete FSLCA, partial FSLCA and SLCA may not be identical in the last four queries. Note that the first four queries are used to compare the performance of MESSIAH with state-of-the-art techniques when missing labels are not in users' queries.

Indexes used. To support efficiently evaluation of Algorithm 1, we exploit the following two indexes. (a) *Inverted List:* Our inverted list maps each keyword w to a sorted set of value matches to w and a set of type matches to w. They are used to find the input set for Algorithm 1. Notice that while the value match is sorted, a sorted index such as B+-tree is not required for P-MESSIAH since we do not employ random access at all. Instead, a hash index can be used which allows faster sequential traversal. The node order is maintained when the index is built while parsing the document. (b) *Type Index:* It maps each type t to a sorted $N_D(t)$. It is invoked in Line 3 of Algorithm 1. Similar to the inverted list, the order of $N_D(t)$ is not required to be maintained by a B+-tree but by inserting them in correct order.

6.2 Results

Search quality. In this experiment, we compare the quality of results returned by SLCA and FSLCA approaches. Since all SLCA approaches return the same results, we



(a) DBLP (b) Mondial

(c) INTERPRO (d) SHAKESPEARE

Figure 8: Recall.

only use the results returned by [17] as representative results. Since the difference between partial and complete FSLCA approaches are some explicitly marked missing data results which do not contribute to precision or recall, we use complete FSLCA (C-MESSIAH) as representative for FSLCA.

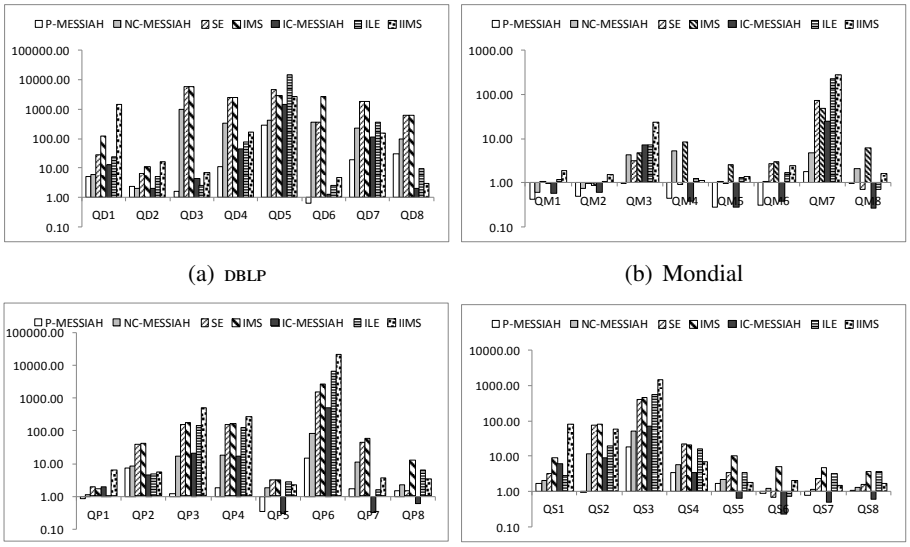
For each query, we ask each volunteer to specify the search intention. We then provide document schema and ask them to convert their intention to XQuery/XPath query. The results of these queries are considered as correct results. The volunteers can discuss among themselves but unanimity is not required.

To measure the search quality, we use *precision* and *recall*, defined as follow:

$$precision = \frac{|Rel \cap Ret|}{|Ret|}, recall = \frac{|Rel \cap Ret|}{|Rel|}$$

where *Rel* is the set of nodes retrieved by an XML query (as described above) and *Ret* is the set of result type matches returned by SLCA/FSLCA. For instance, for QM6, the XPath query agreed by all of our volunteers is `//city[contains(name,'York')]/latitude`. The set of `latitude` nodes retrieved by this query is denoted as *Rel*. *Ret* is the set of all `latitude` matches returned by FSLCA (resp. SLCA)-based techniques.

Since the focus of our paper is on queries with missing labels, some of our experimental queries actually do not have relevant results (QD6, QD8, QM5, QM8, QP5, QP8, QS5, QS7 and QS8). For these queries, recall is not available since the sets of relevant nodes, *Rel*, are empty. Hence, for precision measurement, if the approach returns empty results, we consider the precision in this case $(0/0)$ is 100%. On the other hand, if non-empty results are returned, the precision is considered 0%.



(a) DBLP (b) Mondial

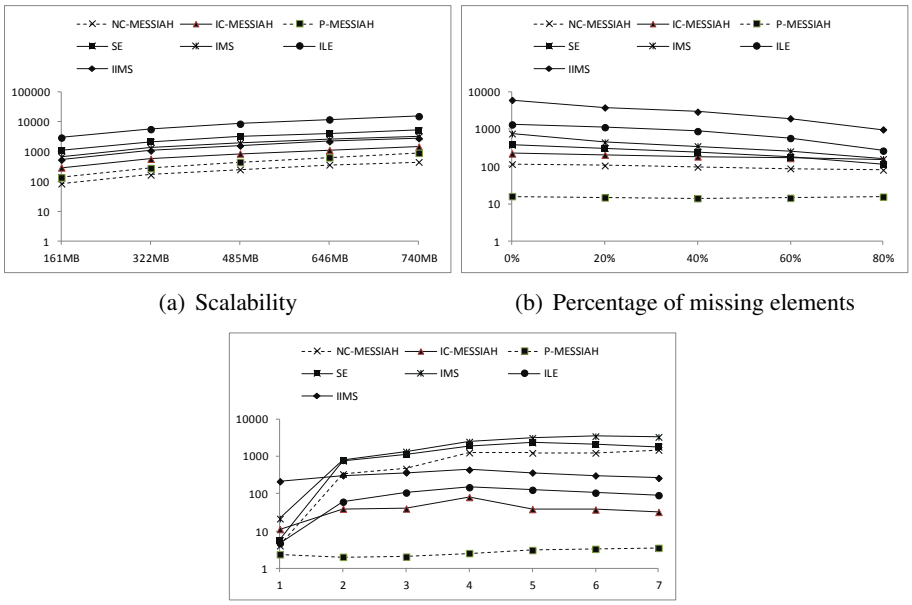
(c) INTERPRO (d) SHAKESPEARE

Figure 9: Execution times (in msec). Note that y-axis is in log scale.

The recall of both SLCA and FSLCA are consistently high. Specifically, except for the no-relevant-result queries (QD6, QD8, QM5, QM8, QP5, QP8, QS5, QS7 and QS8) whose recall is unavailable, the recall of both SLCA and FSLCA are 100% for all other queries. *It shows that our approach matches* SLCA*'s ability to produce high recall [9, 10].*

Figure 7 reports the precision of MESSIAH in comparison with SLCA-based techniques. It is clear that our approach has higher or equal precision than SLCA-based approaches for all queries, especially queries with missing elements (*i.e.,* queries with subscripts 5 to 8). In particular, for queries QD6, QD8, QM5, QM8, QP5, QP8, QS5, QS7 and QS8, SLCA approaches have zero precision. There are no relevant results for these queries. For complete FSLCA approach, no results are returned. On the other hand, SLCA returns lots of irrelevant result matches. For instance, consider QD6. The intention of this query is to find the citations in all of `Aradhye`'s papers. Notice that DBLP only stores the citation information for only few publications [5] and do not have any data on `Aradhye`'s citations. For this query, complete FSLCA returns empty results. Meanwhile, existing SLCA techniques' only result is the root node containing many irrelevant `cite` nodes.

A similar problem also arises for QM6. Note that there are five cities in the world with name containing `York` but only one (*i.e.,* `New York`) contains `latitude` data in the Mondial dataset. For this query, complete FSLCA approach returns a single `city` node corresponding to `New York`. Meanwhile, partial FSLCA will return all 5 cities with `York`. Except for `New York`, the remaining cities' subtrees do not include any `latitude` data. Hence, these partial FSLCA s explicitly indicate that their `latitude` data are missing. On the other hand, for SLCA, besides the `latitude` of `New York city`, the `latitudes` of cities belonging to the same province or same



(a) Scalability

(b) Percentage of missing elements

(c) Number of keywords

Figure 10: Execution times (in msec).

country as `York` are returned instead.

While FSLCA consistently outperforms SLCA-based approaches, for some queries, both suffer poor precision. For instance, considering QM1, most of our volunteers expect the result to be the city of `London, UK`. However, there are other cities containing `London` such as `East London` in South Africa which are unexpectedly returned by both FSLCA and SLCA. Consider another example QP4 whose intention is probably to find proteins published in `Science` in 2002. However, a protein can be involved in multiple publications. Hence if a protein has one publication in `Science` but not in 2002 and another publication in 2002 but not in `Science`, then this protein is still returned as a result. All volunteers agree that both `Science` and 2002 should be associated with the same publication. Nevertheless, we note that *these problems lie on the semantics of* SLCA *itself which is orthogonal to this work.*

We also notice that both FSLCA and SLCA have low precision on text-centric documents (*e.g.,* SHAKESPEARE) compared to data-centric documents (*e.g.,* DBLP, Mondial). The reason is that all SLCA approaches match each keyword individually while, for long-text-attributes, collective matching is expected. For example, consider QS3. Its intention is probably to find the speaker of the line `to be or not to be`. Thus, the keywords `to be or not to be` are expected to be matched collectively to a single node in that order. However, FSLCA and all SLCA-based approaches process those keywords individually so that a result can have different matches for `to`, `be`, `not`, etc. Nevertheless, this problem can be solved by ranking [1, 16] or collective matching as used in IR.

Execution times. In this set of experiments, we study the execution times of MESSIAH against state-of-the-art algorithms to compute SLCA nodes. Notice that we only measure the time to retrieve the SLCA/FSLCA nodes without retrieving the matches. The results are reported in Figure 9 (MESSIAH is shown in solid colors).

Clearly, P-MESSIAH *is faster than most of the* SLCA*-based approaches for majority of the queries*. It is an order of magnitude faster than the *fastest* SLCA-based approaches for several queries (*e.g.,* QD4, QM7, QP3, QP4, QS3). Notice that these queries have a lot of label keywords (*e.g.,* `publication`, `inproceedings`). In an XML document, these keywords typically have a huge number of matches, significantly deteriorating performance of SLCA algorithms while P-MESSIAH is independent of the number of label matches (see Section 5.3). Also, C-MESSIAH is consistently fast for all queries with only one query exceeding 100*ms*.

On the other hand, the execution times of C-MESSIAH (both indexed and non-indexed) are generally worse than P-MESSIAH even when its result size is smaller. It is because, unlike P-MESSIAH, C-MESSIAH requires retrieval of label matches. As discussed in [17], the indexed implementations (IC-MESSIAH, ILE, IIMS) are generally faster and perform well when there are selective keywords in the query (*i.e.,* QM6, QP5, etc.) while the non-indexed implementations are generally slower but perform well when all keywords have high frequency (*e.g.,* QD5). However, notice that C-MESSIAH is still faster than its corresponding counterparts in [14] and [17].

Note that larger result size of P-MESSIAH compared to SLCA-based approaches does not mean slower subtree retrieval cost for the former. Since MESSIAH is conscious of the missing element phenomenon, results of P-MESSIAH are usually specific descendants of SLCA's results. Thus, the size of each result subtree returned by P-MESSIAH is smaller than that of SLCA-based approaches. For instance, consider QD6. The result size of SLCA is 1 but its only result subtree is, in fact, the whole XML tree!

Scalability. In this experiment, we vary the data size of DBLP dataset by trimming it to 161MB, 322MB, 485MB and 646MB. The performance of QD5 is then measured on these datasets. Note that QD5 is chosen because it involves missing elements and has a large result size, ensuring significantly different result size when the data size varies. The results are shown in Figure 10(a). Expectedly, the execution time increases when the data size increases for all approaches. More importantly, *our approaches are significantly faster than all* SLCA *approaches across all datasets*. Also notice that, in this case, NC-MESSIAH is faster than P-MESSIAH. It is because `cite` is a relatively rare label in DBLP [5] so that there are much fewer complete FSLCA nodes than partial FSLCA nodes.

Number of missing elements. In this experiment, we vary the number of nodes which are missing in the document and study its effect on the execution times of the benchmark approaches. The query used for this experiment is `interpro name` on the INTERPRO dataset. Notice that each `interpro` node in INTERPRO has exactly one leaf child with label `name`. We remove $K\%$ of these `name` nodes and measure the performance on the modified document with missing nodes. Here, we vary K from 0 to 80 where $K = 0\%$ refers to the original document. The results are

reported in Figure 10(b). We can make two key observations. Firstly, for existing SLCA algorithms and C-MESSIAH algorithms, more missing elements tend to reduce the execution time. It is expected since more missing elements means fewer SLCA (complete FSLCA) `interpro` nodes to be returned. Secondly, the execution time of P-MESSIAH does not change significantly when the number of missing nodes varies. Recall that the time complexity of Algorithm 1 is independent of the number of label matches. Observe that removing the `name` node for query `interpro name` only affects the number of label matches to `name`.

Number of keywords. Next, we study the effect of number of keywords in a query on the execution time. We use the query `XML title inproceedings author Torsten Grust 2007` (on DBLP) containing seven keywords for this purpose. We first start with the query `XML` and then incrementally add more keywords from left to right. In each step, the execution time is measured and reported in Figure 10(c). The results show that the performances of all approaches except P-MESSIAH vary with different number of keywords. On the other hand, the execution time of P-MESSIAH is faster than these approaches and generally do not vary significantly with the number of keywords. Observe that when the number of keywords increases, a query has more input nodes but the result size also decreases. For SLCA-based approaches the effect of the input nodes dominates since output size is generally not as large as input size. However, for P-MESSIAH, for each keyword, we only retrieve the value matches and type matches and not the label matches. Hence, the input size is typically much smaller resulting in significant performance gain. Also notice that, when the number of keywords increase, the query becomes more selective so that indexed algorithms tend to be faster than non-indexed algorithms. Figure 10(c) also clearly shows that our C-MESSIAH algorithms are faster than the corresponding SLCA counterparts.

Heuristics for algorithm selection. Lastly, we study the accuracy of our proposed heuristic in choosing C-MESSIAH or P-MESSIAH appropriately. As C-MESSIAH and P-MESSIAH share identical result set for all queries with subscripts 1 to 4, we only consider sample queries with subscripts 5 to 8. We set $N = 20$ (desired number of results). Figure 11 shows the results of our study. Clearly, our approach estimates the number of complete FSLCA nodes with reasonable accuracy (see Figure 6) and more importantly uses it to make accurate choice. Specifically, for all queries where C-MESSIAH would return empty results (*e.g.,* QD5, QP5, QD6) or very few results (*e.g.,* QP7, QS6), P-MESSIAH is suitably chosen to effectively inform a user existence of missing elements in desired result sets. On the other hand, for queries with reasonably large results size (*e.g.,* QM7, QP6, QD5), C-MESSIAH is employed as discussed in Section 5.5.

Id	Estimated Num. of Complete FSLCA	Id	Estimated Num. of Complete FSLCA	Id	Estimated Num. of Complete FSLCA	Id	Estimated Num. of Complete FSLCA
QM5	0.221	QP5	0.865	QD5	6373.971	QS5	0.000
QM6	1.104	QP6	40292	QD6	0.029	QS6	0.070
QM7	228.017	QP7	0.000	QD7	11.691	QS7	0.047
QM8	2.882	QP8	0.000	QD8	0.000	QS8	0.000

Figure 11: Heuristic-based algorithm selection (underline means C-MESSIAH is selected, otherwise P-MESSIAH).

7 Conclusions

The quest for high quality keyword search in XML data has become more pressing because many users favor the simplicity and familiarity of search queries to formulating a syntactically correct query using a complex XML query language. State-of-the-art XML keyword search techniques adopt smallest lowest common ancestors (SLCA S) and its variants as a meaningful way to identify matching nodes in XML data. However, SLCA-based approaches perform poorly for queries involving missing elements as they are not optionality resilient. In this paper, we present two variants of a novel algorithm called MESSIAH which identify optionality-resilient FSLCA nodes instead of SLCA nodes to address this limitation. MESSIAH exploits the notion of a full document and the small size of its DataGuide to efficiently identify superior quality FSLCA nodes. A compelling benefit of MESSIAH is that it can be integrated seamlessly with state-of-the-art techniques for relevant return nodes selection, potentially improving the strengths of these approaches. Our empirical study demonstrated that MESSIAH not only produces superior quality results but also is significantly faster than state-of-the-art SLCA computation techniques.

References

[1] Z. Bao, J. Lu, T. W. Ling, and B. Chen. Towards an effective xml keyword search. *IEEE TKDE*, 22(8):1077–1092, 2010.

[2] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[3] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[4] L. Kong, R. Gilleron, and A. Lemay. Retrieving meaningful relaxed tightest fragments for xml keyword search. In *EDBT*, 2009.

[5] M. Lay. Dblp - some lessons learned. In *VLDB*, 2009.

[6] K.-H. Lee, K.-Y. Whang, W.-S. Han, and M.-S. K. 0002. Structural consistency: enabling xml keyword search to eliminate spurious results consistently. *VLDB J.*, 19(4):503–529, 2010.

[7] J. Li, C. Liu, R. Zhou, and W. Wang. Suggestion of promising result types for xml keyword search. In *EDBT*, pages 561–572, 2010.

[8] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[9] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD*, pages 329–340, 2007.

[10] Z. Liu and Y. Chen. Reasoning and identifying relevant matches for xml keyword search. *PVLDB*, 1(1):921–932, 2008.

[11] Z. Liu and Y. Chen. Return specification inference and result clustering for keyword search on xml. *ACM TODS*, 35(2), 2010.

[12] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity estimation for xml twigs. In *ICDE*, pages 264–275. IEEE, 2004.

[13] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[14] C. Sun, C.-Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, 2007.

[15] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[16] A. Termehchy and M. Winslett. Effective, design-independent xml keyword search. In *CIKM*, pages 107–116, 2009.

[17] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD*, pages 537–538, 2005.

[18] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436, 2001.

[19] J. Zhou, Z. Bao, W. Wang, T. W. Ling, Z. Chen, X. Lin, and J. Guo. Fast slca and elca computation for xml keyword queries based on set intersection. In *ICDE*, pages 905–916, 2012.