Generic Keyword Search over XML Data

Manoj K Agarwal
Search Technology Center
Microsoft India
agarwalm@microsoft.com

Krithi Ramamritham
Dept. of Computer Sc. and Eng.
IIT Bombay – India
krithi@cse.iitb.ac.in

Prashant Agarwal
Dept. of Computer Sc. and Eng.
NIT Allahabad - India
agprashant.mnnit@gmail.com

ABSTRACT

XML and JSON have become the default formats to exchange the information for web application or within enterprises. Keyword Search over XML data has been motivated by the need to relieve users from writing difficult XQueries since otherwise users are required to know the complex XML schema. In existing XML keyword search techniques the XML nodes returned for a keyword query are the Lowest Common Ancestor (LCA) nodes for the query keywords. In this paper, we argue that the LCA based techniques *still* require users to be well versed with the XML schema and also the data to be able to obtain meaningful query results.

To address these shortcomings, we present a novel system, Generic Keyword Search (GKS), - for a given keyword query *Q,* instead of identifying (and returning information) only from LCA nodes, GKS returns 'meaningful' information from *any* XML node*,* which contains a subset of keywords in the search query *Q*. GKS response includes LCA nodes, if any, that would have been returned by LCA based techniques.

GKS is also able to find highly relevant keywords and XML schema elements, deeper analytical insights - called *DI* - in the XML data *in the context of the user query*. *DI* enables users to navigate the XML data and to refine their queries even if they are not familiar with the data and the schema. Our experiments on real data sets show that GKS is able to return highly relevant responses to keyword queries efficiently.

1. INTRODUCTION

Semi-structured data, e.g. XML and JSON, are default formats to represent and exchange data within and across enterprises and web [18]. XML data is represented as a labeled, ordered tree *T* as shown in Figure 1(i). The nodes in *T* are either XML schema elements or text nodes. In response to a given keyword query, XML keyword search systems return one or more nodes in *T*, each of which is a Lowest Common Ancestor (LCA) node for *all* the query keywords in the XML data tree *T* [2][5][6][16][17][4]. For instance, in Figure 1, node x_2 is the LCA node for query Q_1. We refer to XML keyword search technique that return LCA nodes in the XML tree, in response to a given keyword query, as *LCA based techniques*. LCA based techniques follow the AND-semantics, i.e., each LCA node contains at least one instance of each query keyword [4].

1.1 Motivation

For a given keyword query $Q=\{k_1,..k_n\}$ ($|Q|=n$), instead of identifying LCA nodes and returning information only from these nodes, Generic Keyword Search (GKS) returns *any* node in the labeled tree *T,* if it contains *s* or more keywords in the search query *Q* ($s \leq n$). More formally, the GKS problem is defined as follows: For a keyword query *Q,* an integer $s \geq 1$, search returns all the XML nodes which contain at least $min(s, |Q|)$, keywords from *Q*. The set of XML nodes returned by GKS in response to query *Q* is denoted by $R_Q(s)$. $|R_Q(s_1)| \leq |R_Q(s_2)|$ if $s_1 > s_2$ (cf. Section 2.2).

There are many notions of LCA nodes in the literature but SLCA (Smallest LCA) [13] and ELCA (Exclusive LCA) [17], are most widely used. An SLCA node contains all the query keywords in its sub-tree and there is no node in its sub-tree which contains all the keywords. An ELCA set of nodes is a superset of the SLCA nodes. In Figure 1, for query Q_1, node x_1 is an ELCA node but not an SLCA node due to the presence of x_2 in its sub-tree. In the figure, k_i is an instance of keyword k (e.g. $\boldsymbol{a}_i$s are instances of $\boldsymbol{a}$). For different notions of LCA nodes, progressively faster algorithms have been proposed to retrieve them [16]. The nodes in GKS response set follow the semantics of SLCA.

As pointed out by the authors of [19] "LCA based techniques work poorly for documents having *irregular schema* that have *missing elements*" because the schema allows certain XML nodes to be optional. Further noting that if a document is not complete, the resulting output could be different from the intended output. Authors of [19] develop an alternate approach whose basic premise is: for a given keyword search query, specific XML node types are targeted [15][19]. However, if the document has "missing XML elements", nodes other than targeted nodes could also be returned due to the constraint on LCA based techniques (only LCA nodes are returned for the keyword query). Clearly, the motivation for [19] highlights that for LCA based techniques a) users need to be aware of the schema (i.e., users need to be aware which XML nodes to target); b) query keywords must be chosen by taking into account the semantic relationship between them (query must be formed such that the target nodes could be returned); and c) users need to be aware of the keywords in the XML document(s) and their distribution in XML tree *T* (otherwise nodes other than targeted nodes could become LCA nodes). In other words, in order to be able to effectively search the data using LCA based techniques, users have to be well acquainted with the data and the schema.

AND-semantics constraints underlying LCA based techniques are further highlighted by the following example:

Example 1: Consider keyword queries Q_1, Q_2, Q_3, on the XML document in Figure 1(i). Each leaf node in the XML document is a text node (text node is an XML element directly containing its value). We have represented the document as shown in Figure 1(i) for brevity. Response of SLCA and ELCA based algorithms are

shown in Table 1. For query Q_3, even though the user is able to select all the keywords present in the document, the response of LCA algorithms is root {r}. 'r' is not a meaningful response as it is available to the user even in the absence of any query.

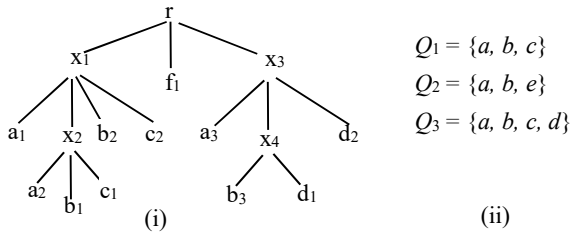

Figure 1. Labeled XML data tree and a set of queries.

Table 1. Nodes returned for different queries on labeled XML tree by different keyword search algorithms

Queries	GKS (ranked)	ELCA	SLCA
Q_1,s=$\|Q_1\|$	{x_2}	{x_1, x_2}	{x_2}
Q_2, s=2	{x_2}, {x_3}	NULL	NULL
Q_3, s=2	{x_2,}, {x_3}, {x_4}	{r}	{r}

Therefore, to construct meaningful queries, users need to know the keywords distribution in the document. To know the keyword distribution, it is imperative to know the semantic relationship between query keywords. In order to be aware of the semantic relationship between the query keywords, users must know the schema of the XML repository. Users must also know the schema to form the query such that targeted XML nodes are returned.

MESSIAH [19] addresses the issues arising due to the AND-semantics of LCA based techniques. Its authors propose an efficient FSLCA algorithm to identify intended nodes, in case of missing XML elements in the data. MESSIAH addresses the missing element problem, if the data is 'imperfect'. However, the issues of 'missing data elements' is still not handled in [19] if the user query is 'imperfect'. For instance, if a query keyword occurs in the wrong sub-tree, it is difficult to determine the intended return nodes. Hence, for a keyword query, possibly containing semantically uncorrelated keywords, nodes other than targeted nodes will be returned by [19] even if the missing XML elements are identified.

Consider the following scenario: User starts with query Q_2 and Q_3 (shown in Figure 1). GKS returns a set of XML nodes to the user, as shown in Table 1, which contain a significant fraction of the query keywords but not necessarily all the query keywords (s=2). Besides returning these nodes, let us say, GKS system also suggests to the user that query Q_2 can be morphed to {*a, b, c*} or {*a, b, d*} from {*a, b, e*} The user may not be aware of the existence of the keywords {'*c*', '*d*'} or their relevance in the context of the query. Similarly, for Q_3, the system suggests that query be partitioned into {*a, b, c*} and {*a, b, d*}. Such refinements of the user queries are non-trivial. Overall, we are motivated by the following goals 1) to relax the need for users to know the XML data precisely; this enables them to browse the XML data in a manner similar to web search; 2) to relax the need for users to know the XML schema as user queries can be refined progressively (as for query Q_3).

1.2 Generic Keyword Search

In this paper, we introduce a novel concept of ***Generic Keyword Search*** (*GKS*) over XML data to address the shortcomings listed above. It enables the users to navigate XML data with ease, as demonstrated with the help of an example on real data below run on the implemented GKS system [20]:

Example 2: We have a DBLP dataset with more than 2.5 million articles. A query Q_d = {`"Peter Buneman" "Wenfei Fan" "Scott Weinstein" "Prithviraj Banerjee"`} is run on this dataset. The user is most likely interested in articles jointly written by these authors. In its response, a total of 234 articles (for s=1) are found by GKS, i.e., GKS return all the articles by any of the authors in the keyword query since s=1.

Since the response of GKS contains a large number of XML nodes (i.e., <*inproceedings*>), with different XML nodes in the response containing different number of authors, the results are ranked such that the more relevant XML nodes are ranked higher (cf. Section 5). For query Q_d, the <*inproceedings*> nodes with higher number of query keywords (i.e., author names) in their sub-tree are likely to be ranked higher. We use just <*ip*> for <*inproceedings*> later on.

In the DBLP dataset, there is no article jointly written by `Prithviraj Banerjee` with any of the remaining authors. Of the five articles jointly written by the remaining three authors in DBLP dataset, 4 were returned as top 4 results in the ranked list of XML nodes by GKS. The remaining article was also in top 10 (it was ranked lower due to many co-authors, details Section 5). In the context of this example, we now explain how GKS overcomes the shortcomings of the LCA based techniques:

GKS relaxes the need for users' familiarity with the contents: For the given query, an LCA based technique would have returned `{DBLP root}`, containing millions of articles as the response due to the presence of one "wrong" keyword '`Prithviraj Banerjee`' in the query. On the other hand, GKS produced a more "meaningful" response in the presence of "wrong" query keyword(s). This helps the users as follows;

a) With GKS, users can navigate the XML data without complete awareness with underlying data (for LCA based techniques, users need to know, which authors have published articles together). GKS returns a ranked list of most relevant XML nodes, in the context of the query, considerably enhancing the users' ability to search the data with high precision and recall.

b) More importantly, even when users are able to formulate the query precisely, there is a lot of information which could be of their interest, which are not returned by LCA based techniques due to the constraint that *only* LCA nodes must be returned. For instance, in Example 2, the articles by a large enough subset of authors in the query Q_d could also be of interest to the user in the context of the query. Exposing such results in the data helps users navigate the data as well as to refine their queries (cf. Section 6.1).

GKS relaxes the need for users' familiarity with the schema: GKS identifies the XML nodes, which are not necessarily LCA nodes but that could be of interest to the user in the context of the query. This ability of GKS can be exploited to discover most relevant keywords and their semantics in the underlying XML data, in the context of the user query. This information is called *deeper analytical insights* or *DI*. For the query in Example 2, GKS exposes <*ip: journal: SIGMOD Record*>, <*ip: year: 2001*>, <*ip: author: Alok N Choudhary*> and <*ip: booktitle: ICPP*>, etc., as *DI* from the XML data in the context of the query (GKS returns a well-constructed XML chunk. Truncated representation is due to lack of space). *DI* exposes the most relevant journals, year and authors in the query response. The user may not be aware of these keywords or their relevance in the context of the query.

DI is defined formally in Section 2.3 (Def. 2.3.1). Discovery of *DI* (Section 6) enhances the users' ability to navigate the data even if they are unaware of the schema details and the semantic relationship between the various data keywords. To discover *DI,*

we exploit the XML schema, embedded in the structure of the XML data, in the context of a user query. A novel node categorization model is proposed that identifies certain XML node types as Least Common Entity nodes or LCE nodes (cf. Section 2.2). LCE nodes are central to our methodology to discover *DI.* A subset of XML nodes $E_Q \subseteq R_Q(s)$ in GKS response for query Q can be Least Common Entity (LCE) nodes; $0 \leq |E_Q| \leq |R_Q(s)|$. For an XML node u containing a sub-set of query keywords (of size $\geq s$) in its sub-tree, its corresponding LCE node will be either u itself or its ancestor.

In Example 2, <*ip*> node is an LCE node. GKS exposes the semantics of the *DI* keywords, i.e., 2001 is a <year> with the aid of XML elements on the path from the root of LCE node <*ip*> till the keyword "2001". Semantics are important as in a different context, 2001 could be a street number. Query keywords, either XML element names or text keywords, may carry different meaning in different context [9]. Exposing the relevant keywords and their semantic meaning helps users refine their queries in the absence of knowledge about the schema and the data.

GKS returns meaningful response: The meaningfulness of the results of a search query is defined by their recall and precision. In LCA based search, a keyword query typically targets XML nodes belonging to specific schema elements **E** in the associated XML schema [19]. <E> ∈ **E** is an XML schema element. The target nodes are the LCA nodes of the query keywords. If the returned LCA nodes are of targeted schema element type(s), it constitutes a meaningful response. For the query Q_d in Example 2, the meaningful LCA nodes for this query are all of type <*ip*> in the corresponding XML schema.

However, due to imperfect data with missing XML elements or due to imperfect query, LCA based techniques often return LCA nodes other than the target XML elements type [19]. For instance, for query Q_d, LCA based techniques will return the DBLP root. A more meaningful response is a ranked list of articles, jointly written by a sub-set of authors in the query, i.e., returning nodes of same type, which were targeted. For GKS system, all the XML nodes that contain any subset of keywords in a query (of size $\geq s$) are returned. Therefore, recall of GKS is likely to be high since GKS query response is likely to have XML nodes which are instances of target XML schema element in **E** for a user query Q (any XML node containing $s \leq |Q|$ keywords in its sub-tree is returned).

In the context of a keyword query, the relevance of a XML node is high if it contains a large fraction of query keywords. The precision of the GKS system will be high if the most relevant XML nodes in the GKS query response are ranked higher. We present a novel ranking methodology (Section 4) to ensure high precision.

1.3 Research Challenges and Contributions

Similar to a web search engine, Generic Keyword Search has the twin objectives of: a) locating the most relevant XML nodes for the given keyword query efficiently; and b) ordering the search results to rank more meaningful results higher.

GKS has three primary challenges; 1) **Efficiency** –GKS has much larger search space as opposed to LCA based techniques (Lemma 3). Therefore, a major challenge for GKS is to be able to retrieve the relevant nodes efficiently (Section 4); 2) **Ranking** – Number of XML nodes retrieved by GKS could be large and the structure of the different XML nodes in the search results could be different. Therefore, it is imperative to rank the nodes such that more meaningful and relevant nodes are ranked higher (Section 5); 3) **Analysis -** GKS aims to enable the users to refine their queries without needing them to be familiar with schema and data. GKS meets this challenge by exposing relevant keywords in the data and their semantics in the context of the user query (Section 6).

In this paper, we make the following contributions:

1. Existing XML Keyword Search techniques work within LCA framework. We introduce Generic Keyword Search (GKS) that enables XML search beyond LCA framework.
2. We propose a XML node categorization model. With the aid of this model, we expose most relevant XML elements and data keywords, called *DI*, in the context of a given keyword query. Users can refine their query with the aid of *DI*. *DI* is discovered *because* GKS does not impose the LCA constraint.
3. We introduce a ranking methodology to rank more meaningful XML nodes, retrieved by GKS, higher. Node ranking is further exploited for *DI* discovery.
4. We present an evaluation of GKS system on real data sets. Our results show that GKS is able to return highly relevant response for the given keyword queries efficiently. We further show that our system is able to find highly relevant *DI* that enables the users to navigate the XML data seamlessly.

The organization of the paper is as follows. Section 2 introduces the GKS node categorization model along with the definitions of LCE nodes, *DI* and the GKS indexing structure. Related work is presented in Section 3. Our methodology to identify the relevant XML nodes efficiently is the subject of Section 4. In Section 5, we present a novel XML node ranking methodology. In Section 6, we discuss our mechanism to discover *DI*. In Section 7, we present experimental results followed by conclusion in Section 8.

2. XML NODE CATEGORIZATION AND DEFINITIONS

In this section, we first present a novel XML node categorization model. The XML node categorization helps us exploit the XML schema, embedded in the XML data, to identify relevant data keywords and XML schema elements in the context of a user query. We also present the definitions of LCE nodes and *DI*, GKS system architecture and the indexes maintained by GKS.

2.1 Preliminaries

An XML document is a rooted tree T as shown in Figure 2(a). Nodes in the tree are labeled with Dewey id [1]. Dewey id is a unique id assigned to a node that describes its position in the tree T. A node with Dewey id 0.2.3 is the fourth child of its parent node 0.2. n_{id} represents an XML node with Dewey id *id*. $v \prec_a u$ denotes that node v is an ancestor of node u. $v \lhd_a u$ denote that $v \prec_a u$ or $v=u$. U represent a set of XML nodes (or keywords) in XML tree T. $v \prec_{lca} U$ denotes that v is the lowest common ancestor of nodes in set U. For a text keyword or XML node k, $k \in v$ denotes that k occurs in the sub-tree rooted at XML node v and $k \notin v$ denotes that k does not occur in v's sub-tree. u^* denote that one or more siblings of node u exist in tree T with same XML element label. $v \prec_e u$ denotes that v is an entity node w.r.t. u (Def. 2.1.3) and $u \in v$ or $v=u$. $v \prec_{lce} u$ denotes that XML node v is the lowest common entity node (LCE) w.r.t. node u (Def. 2.2.1) and $u \in v$ or $v=u$.

2.2 Node Definitions

We divide the XML nodes in the following categories, based on the structure of their sub-trees in T.

***2.1.1. Attribute Node* (AN):** A node which contains only one child that is its value. For instance, in Figure 2(a) node <Name> ($n_{0.1.0}$) is an attribute node. Attribute nodes are also represented as 'text nodes' in XML data. The parent node of an attribute node is

considered the lowest ancestor for keyword(s) in its value (and not the attribute node itself). Thus, ancestor of 'Databases' is node $n_{0.1}$.

***2.1.2. Repeating Node* (RN):** Let $v \prec_{lca} u^*$, i.e., v is the lowest common ancestor of multiple instances of node u. u is called the repeating node w.r.t. node v. For instance, in Figure 2(a), nodes with label <Student> are repeating nodes w.r.t. <Students>. The repeating nodes most likely correspond to a physical world object which could be a concrete or an abstract object [3]. A node that directly contains its value *and* also has siblings with the same XML tag is considered a repeating node (and not an attribute node), i.e., <Student> nodes in Figure 2(a).

***2.1.3. Entity Node* (EN):** Let v be an XML node in XML tree T such that $v \prec_{lca} (u^*, A) \mid \forall a \in A, a \notin u^*$. v is an entity node. A is a set of attribute nodes. An attribute node $a \in A$ does not occur in any repeating node u, i.e., a does not have u in its XPath from root.

An *entity* node v is a lowest common ancestor of repeating nodes u and one or more attribute nodes ($|A| \geq 1$). In Figure 2(a) <Area> ($n_{0.1}$) is an *entity node*; it is the lowest common ancestor of attribute node <Name> ($n_{0.1.0}$) and repeating nodes <Course> ($n_{0.1.1.x}$). <Course> nodes are not the direct children of $n_{0.1}$ (Attribute nodes and Repeating nodes can be indirect children of *entity* node). Similarly, <Course> nodes ($n_{0.1.1.0}$, $n_{0.1.1.1}$, $n_{0.1.1.2}$, ..) are the entity nodes.

***2.1.4. Connecting Node* (CN):** Nodes which are in none of the above categories. In Figure 2(a), <Courses> ($n_{0.1.1}$) is a connecting node.

Table 2: Notation

s	Minimum number of keywords from a query that must appear in the sub-tree of a XML node.
$R_Q(s)$	Set of XML nodes for a given s, returned by GKS in response of query Q
$R(e)$	For an LCE node $e \in R_Q(s)$, $R(e)$ is a subset of text keywords, extracted from attribute nodes of e.
S^w_Q	Weighted set of text keywords, identified from the LCE nodes in set $R_Q(s)$.
$\prod_r R_Q$	Set of XML nodes, after recursively applying the GKS algorithm r times over the query results $R_Q(s)$.

XML documents follow pre-order arrival of nodes. Hence, different node types are identified in a single pass over the data. GKS does not need the XML schema in order to categorize nodes. XML nodes are categorized at the instance level. This information is stored in an index (Section 2.4). Hence, each node is categorized based on the structure of its sub-tree. For example, all the instances of <Course> node in Figure 2(a) are *entity* nodes (Def. 2.1.3). However, if a <Course> node had just one student in its sub-tree, that instance would have been stored as 'Connecting node' in the index. GKS can be easily extended to take into account the XML schema to categorize the nodes. This is part of our future work.

The node categories described above extend the node categorization model in [3]. It is argued in [3] that in the hierarchical structure of XML data, repeating nodes (Def. 2.1.2) capture the concept of physical world object. The physical object could be a concrete or an abstract object. In normalized XML data, attributes of an XML node that contains repeating nodes in its sub-tree, represent the information that is common to these repeating nodes [14]. The fundamental design principle underlying the normalized XML schema is, the attribute nodes of an XML node define the context of the repeating nodes in its sub-tree through their values. In GKS node categorization model, such XML nodes are termed entity nodes (Def. 2.1.3). As shown in the experiment in Section 7.2, we count the total number of XML nodes and XML nodes that were labeled as entity nodes, attribute nodes and repeating nodes, respectively by GKS for many standard XML data repositories. The result shows that the real world data repositories are normalized. The note categories described above naturally capture the normalized XML data.

A node can be an *entity node* and at the same time a *repeating node* for another *entity node* higher up in the hierarchy. For instance, in Figure 2(a), <Course> nodes are both entity nodes as well as repeating node within the sub-tree of node <Area> ($n_{0.1}$). Let Q be a keyword query, $|Q| > s$, and $Q' \subseteq Q$; $|Q'| \geq s$. Let LCA node u for Q' is not an *entity* node and v is the lowest ancestor of node u such that $v \prec_e u$. Hence, node u can either be a connecting node or a repeating node w.r.t. v. Since u does not have the attribute nodes, as it is not an *entity* node, the context of the node u is most specifically defined by the attribute nodes of node v. In Figure 2(a), attribute <Course: Name: Data Mining> defines the context that <student> nodes in its sub-tree are registered in this course.

For a given keyword query, the closer the entity node is to the query keywords in its sub-tree, the more specific the context would be for those keywords. As we move up in the hierarchy, the context of the corresponding sub-tree becomes more general. In Figure 2(a), <Dept> node and <Course> node both are entity nodes and both contain the query keywords for a query Q= {'Karen, 'Mike'}. However, the context of entity node <Dept> is much more general compared to more specific context of the node <Course>. Hence, to find the more meaningful response for a given query, we discover the entity node closest to the query keywords or Least Common Entity (LCE). LCE is formally defined below.

Let S_c be a set of all the *entity* nodes in the sub-tree rooted at an *entity* node e_c, i.e., $e_c \prec_a e; \forall e \in S_c$. Let Q be a keyword query, $Q=\{k_1,..k_n\}$.

Def 2.2.1 LCE Nodes: An *entity* node e_c is an LCE node for query Q *if* $\exists k \in Q \mid k \in e_c \wedge \forall e \in S_c, k \notin e$.

Hence, for an *entity* node e_c to be LCE node for a given query Q, there exists at least one keyword $k \in Q$ in the sub-tree of e_c, which is *not* contained in any other *entity* node e such that $e_c \prec_a e$.

Keyword k is called an independent witness for LCE node e_c. Similar to an SLCA node, an LCE node also needs at least one independent witness.

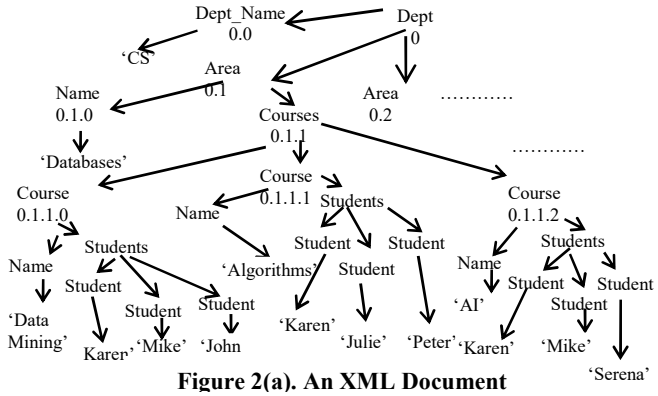

Figure 2(a). An XML Document

Lemma 1: Let $v \lhd_a u$ denote a relationship that $v \prec_a u$ or $v=u$. Let u be an XML node that is an LCA node for a set of keywords $Q_s \subseteq Q$, $|Q_s| \geq |s|$. Let v be an LCE node for keywords in Q_s. $v \lhd_a u$

Proof: Obvious. □

For a given user query Q, GKS returns a set of XML nodes $R_Q(s)$ such that for each node $u \in R_Q(s)$, u contains at least s keywords from query Q.

Lemma 2: For a keyword query Q and integers s_1 and s_2, $|Q| \geq s_1 > s_2$, $|R_Q(s_1)| \leq |R_Q(s_2)|$.

Proof: Since $s_1 > s_2$, $\forall v \in R_Q(s_1), \exists u \in R_Q(s_2) \mid v \lhd_a u$. However, $\forall u \in R_Q(s_2)$ there can be at most one $v \in R_Q(s_1) \mid v \prec_{lce} u$. Thus, for $\forall v, v \in R_Q(s_1)$ there exist a corresponding node in $R_Q(s_2)$ but vice versa is not true. Thus, $|R_Q(s_1)| \leq |R_Q(s_2)|$. □

Example 3: Let there be a user query Q_4={student, karen, mike, john, harry}, s=2. The intent of the query is to find the information about these students. For the data shown in Figure 2(a), 3 courses contain the names of at least one of these students. The GKS response constitutes the XML nodes as shown in Figure 2 (b). The XML nodes are LCE nodes since they are the lowest entity nodes, w.r.t. query keywords. Attribute nodes of respective entity nodes exposes the context, i.e., name of the respective courses students are enrolled in. The XML nodes are ranked (cf. Section 5).

As one can see, the user query in Example 3 is 'imperfect'. To construct a 'perfect' query, for a LCA based technique, user needs to be aware which students are enrolled in same courses. User still has to run multiple queries to get the complete response. GKS returns the relevant and meaningful information in the context of this 'imperfect' query. We further enhance a user's capability to refine an 'imperfect' query by exposing the deeper analytical insights in the query response as explained in the next section.

2.3 Deeper Analytical Insights (*DI*)

For the query in Example 3, let's say user runs a 'perfect' query Q_5={student, karen, mike, john}. The response of a LCA based technique [2][5] will be XML sub-tree rooted at node $n_{0.1.1.0.1}$ <Students> node. Even though the query is perfect, the response still does not yield any meaningful information. On the other hand, GKS response is node $n_{0.1.1.0}$ for $s=|Q|$ ($n_{0.1.1.0}$ is an LCE node for Q_5) with the aid of its node categorization model. Thus, GKS response exposes the information that the students are registered in 'Data Mining' course. This information, <Couse: Name: 'Data Mining'>, is called *deeper analytical insights* or *DI*. *DI* enables users to navigate the XML data by exposing relevant schema and the data elements that help users not only understand the query response but also help refine their queries.

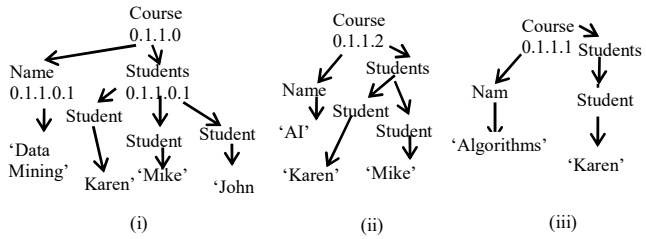

Figure 2(b). Response of the GKS System for Q_4

To discover *DI*, for a user query Q and s, GKS prepares a set of keywords S^w_Q from nodes in set $R_Q(s)$ as follows: For each node $u \in R_Q(s)$, if u is an LCE node, GKS extracts the text keywords from its attribute nodes and put them in set S^w_Q. For instance, for the query in Example 3, entity nodes $n_{0.1.1.0}$, $n_{0.1.1.1}$ and $n_{0.1.1.2}$ are the LCE nodes in the set $R_Q(s)$ (Figure 2(b)). Each of the entity nodes has an attribute node <Name: Data Mining>, <Name: AI> and <Name: Algorithms>. The set S^w_Q will contain keywords {"Data Mining", "AI", "Algorithm"}. $R(e)$ represents the set of attribute nodes in the sub-tree of entity node e (see Table 2). Given a query response $R_Q(s)$, we prepare a set of keywords $S^w_Q = \{k_1 \ldots k_n\}$ containing the text keywords embedded in the attribute nodes for each of the entity nodes in $R_Q(s)$.

Def 2.3.1 DI: Let $E_Q \subseteq R_Q(s)$ be the set of all LCE nodes in GKS response for keyword query Q and let $S^w_Q = \bigcup R(e) \mid e \in E_Q$. $DI \subseteq S^w_Q \mid \forall k \in DI; k \notin Q$.

For a keyword k in *DI*, let e be its corresponding LCE node. For the keyword k, we also associate the XML elements in the path from node e till keyword k. The keywords and the associated XML elements with each keyword together form the *DI*.

DI can also be discovered recursively for a user query as described below. We use only set $R_Q(s)$ and not E_Q since context is clear.

i) GKS parses the LCE nodes in set $R_Q(s)$, for a given keyword query Q and prepares a weighted set of keywords S^w_Q by identifying a subset of text keywords in each of the LCE nodes (Section 6.2).

ii) *Top-m* most weighted keywords in the set S^w_Q are fed to GKS as a query. GKS identifies a set of XML nodes w.r.t. these keywords from set S^w_Q. This set of XML nodes is denoted as $\prod_1 R_Q(s)$. Set $\prod_0 R_Q$ ($\prod_0 S^w_Q$) is denoted by just $R_Q(s)$ (S^w_Q).

The above steps can be applied recursively -- $\prod_r R_Q(s)$ represents the set of LCE nodes after r^{th} recursion.

iii) *GKS* prepares the set of keywords $\prod_r S^w_Q$ from the nodes in $\prod_r R_Q(s)$. DI_r is extracted from $\prod_r S^w_Q$; $r \geq 0$.

DI can be discovered recursively for a user query Q by extracting a ranked list of most relevant keywords and their semantics from $\prod_i S^w_Q$ at each step i of recursion. In summary, *DI* is discovered 1) with the aid of GKS node categorization model; and 2) *because* GKS does not impose the LCA constraints and thus retrieves all the relevant XML nodes in the query context. These XML nodes help discover meaningful *DI*.

2.4 GKS Architecture and Indexes

In Figure 3, we depict the architecture of GKS. The GKS takes as input XML data and prepares an index on it. The XML data could be spread over multiple files. For a user query Q, GKS produces a) ranked search results on the data; b) deeper analytical insights (*DI*) by analyzing search results. GKS contains three modules; i) Indexing Engine; ii) Search Engine; iii) Search Analysis Engine.

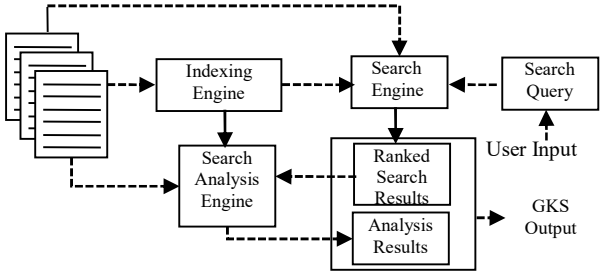

Figure 3. Architectural of the GKS System

For a given XML data repository, we first prepare an index on it. This is a onetime activity. We keep the following indexes:

Inverted Index for text keywords: For each unique text keyword that appears in the XML document repository, we keep an inverted index list. If text appearing under a 'text node' comprises multiple

keywords, a separate index entry is created for each of the keywords after stop words removal and stemming. A partial inverted index for document in Figure 2(a) is shown in Table 3. The inverted index list for a keyword k_i contains the Dewey id of all the nodes which contain that keyword. Dewey id for each node has been appended with the document id 'd_{id}'. Thus, GKS search is seamlessly expanded over multiple documents by prefixing Dewey ids with corresponding document id. For a keyword k_i present in the XML document repository, S_i denotes its inverted index list.

Hash tables: We keep two hash tables corresponding to XML elements. Hash table 1, called *'entityHash'*, keeps the Dewey id of entity nodes. Hash table 2, called '*elementHash*', keeps the Dewey ids of repeating nodes and connecting nodes. Both hash tables also store the number of direct children each node has. This information is used while computing the rank of a node (Section 5). If a XML element is both a 'repeating node' and an 'entity node', its entry is present in both the hash tables.

Since XML nodes arrive *pre-order* (an ancestor of an XML node always appears before it), the hash tables and the inverted index are created in a single pass over XML data.

Table 3. Partial inverted index for XML document in Fig 2(a).

Karen	d_{id}.0.1.1.0.1.0	d_{id}.0.1.1.2.1.0	……
Mike	d_{id}.0.1.1.0.2.0	d_{id}.0.1.1.2.2.0	…..

We provide two functions: i) *isEntity* (Deweyid); ii) *isElement* (DeweyId). Both the functions return the number of direct children the given node has if true, null otherwise.

3. EXISTING WORK in the GKS CONTEXT

A large body of work exists to understand the user's intent for a keyword query over XML data. The work related to GKS can be divided into 3 categories: 1) Identifying meaningful return nodes for a keyword query, 2) Result type deduction techniques and 3) Ranking the XML nodes retrieved in response to a user query.

Identifying meaningful return nodes: Users present their keyword query and the underlying algorithm interprets the user's intent and tries to identify the return nodes accordingly [2][3] [5][6]. The existing approaches for identifying most relevant return nodes are based on first discovering SLCA nodes [13]. Different heuristics are applied on the set of SLCA nodes to identify meaningful return nodes. In XSeek [3], authors propose a technique that first finds the SLCA nodes for a given keyword query. The keywords in the query are understood as the 'where' clause whereas 'return' nodes are inferred based on the semantics of 'query keywords'. MaxMatch [11] and RTF [12] are SLCA based approaches to identify meaningful return nodes. In [11], irrelevant' match results are filtered from each SLCA node. In [12], authors propose an improved algorithm to address *redundancy* and *false positive* problems of [11]. In all the approaches above, a set of SLCA nodes is identified for given keyword query. In [10] authors address the problem due to imprecise XPath queries.

Deducing result types: Deducing return node types is also an important goal for GKS since for most keyword queries, users target certain node types. However, due to lack of knowledge about the distribution of keywords in the document, different semantic meaning of same keywords or due to lack of familiarity with the document schema, the query may not by semantically 'perfect'. In [15][19], it is assumed that the keyword query is semantically correct and certain node types are the target nodes for a given query. XReal [9] and XBridge [4] address the problem of deducing the return nodes types. In [9] the authors count the confidence level to deduce the result node types. In [4] authors highlight the fact that keywords may exist in different context. XBridge automatically predicts the intended result types for XML keyword queries by considering the value and structural distributions of the data. The more generic solution to this problem is to enable users to further refine their queries. GKS approach is a step in that direction.

Ranking the XML nodes: The XML ranking techniques are divided into IR [9][8] based methods and relevance score based [15][7] methods. XRank [7], XSEarch [8] are techniques to rank the keyword query search results based on LCA nodes. XRank takes into account the keyword proximity in the XML nodes whereas XSEarch computes the node rank based on TF-IDF based method. The basic differences between these methods and GKS technique is: In existing XML ranking methods, each of the XML nodes that is ranked contain a fixed set of all query keywords. XML nodes in GKS response contain varying number of query keywords. We have outlined the issue arising due to this difference in Section 5 when we present GKS ranking methodology.

4. SEARCHING GKS NODES

The basic difference between the LCA based search and GKS-Search is: GKS has exponential search space compared to LCA based techniques. For query Q ($|Q|=n$), a total of $(2^n - \sum_{i=1}^{s}\binom{n}{i-1})$ sub-sets, of size at least s, can be formed; $s \le n$. To identify GKS nodes, a naïve approach would be to create all the keyword subsets (of size $\ge s$) for query Q, and for each of these keyword subsets, identify the LCA nodes. Together, all the LCA nodes thus discovered can be used to produce the GKS response. However, this approach results in an exponential number of sub-queries.

Lemma 3: For a given query Q, $|Q|=n$, $s \le \lfloor n/2 \rfloor$; GKS has exponential search space w.r.t. an LCA based techniques.

Proof: For a given query Q, $|Q|=n$, a total of $U = 2^n - \sum_{i=1}^{s}\binom{n}{i-1}$ sub-sets can be formed such that each set is of size at least s. Now, $\sum_{i=1}^{n}\binom{n}{i-1} = 2^n - 1 \Rightarrow \sum_{i=1}^{n}\binom{n}{i-1} \le 2^n$ since $\binom{n}{i} = \binom{n}{n-i}$. Hence, $\sum_{i=1}^{\lfloor n/2 \rfloor}\binom{n}{i-1} \le 2^n/2$; Therefore $\sum_{i=1}^{s}\binom{n}{i-1} \le 2^n/2; s \le \lfloor n/2 \rfloor$.

Since, $U = 2^n - \sum_{i=1}^{s}\binom{n}{i-1} \Rightarrow U \ge 2^{n-1}$. Therefore, an exponential number of sub-sets are formed when $s \le \lfloor n/2 \rfloor$ with each sub-set leads to one keyword query for an LCA based techniques. □

Lemma 3 shows GKS has exponential search space w..r.t. LCA based techniques. Further, the naïve approach does not discover the LCE nodes, in absence of GKS node categorization model, which allow GKS to expose *DI* in the context of the user query. Hence, LCA techniques cannot be applied as is for GKS-Search. In this section, we present an efficient method to find relevant XML nodes for GKS-Search. We call them GKS nodes. A subset of GKS nodes can be LCE nodes. We also present the correctness analysis and time complexity analysis of our method.

4.1 Efficient Method to Search GKS nodes

For the query keywords $k_i \in Q$, we first merge their respective inverted index lists such that in the merged list, keywords follow their arrival order in the XML document. Since the Dewey ids of the XML nodes follow *pre-order* traversal, if the merged list is sorted on Dewey ids, we achieve such ordering. Let d be the depth of the XML tree T being queried. Depth of the tree T is defined as

the number of edges from the root of the tree to its deepest leaf. Let $|Q|=n$, i.e., n lists are merged. Let $|S_i|$ be the inverted index length for keyword $k_i \in Q$. Let S_L be the merged and sorted list. $|S_L| \leq \sum_{i=1}^{n} |S_i|$. The time complexity to merge k sorted lists, of total length l, into a single sorted list in $O(l.logk)$. Since the inverted index list for each keyword is sorted on its Dewey id, therefore the n lists are merged in a single sorted list S_L in $O(d|S_L|log\ n)$.

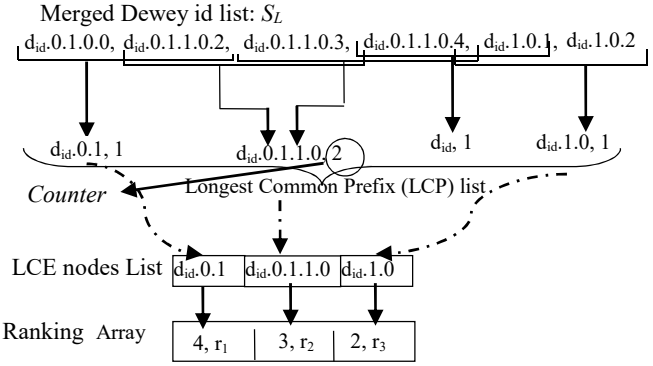

Figure 4. List of longest common prefixes for Dewey id blocks of size s=2

Generating list of candidate GKS Nodes: Our next objective is to generate the list of candidate GKS nodes that have $Q_s \subseteq Q$ keywords appearing in their subtree such that $|Q_s| \geq s$. Towards that end, in the merged list S_L, longest common prefix is identified for a continuous block of s entries, as shown in Figure 4 (in Figure 4, $s=2$). We traverse the list S_L from left to right. Since the list S_L is sorted, the Dewey ids of the nodes in a common sub-tree occur next to each other, with an ancestor node preceding its descendent. Therefore, in S_L, the longest common prefix of a block of s nodes will be the Dewey id of the common ancestor for the nodes in this block. There will be at most $(|S_L| - s)$ such prefixes.

The prefixes are put in Longest Common Prefix (LCP) list as shown in Figure 4. With each prefix entry in LCP list, we associate a *counter* which is initialized to 1. If a prefix exists in the LCP list (i.e., more than s query keywords exist in its sub-tree), its counter is increased by 1. Since the block of s entries in the list S_L slides to the right by 1 at a time, the *counter* can increase by only 1 at a time.

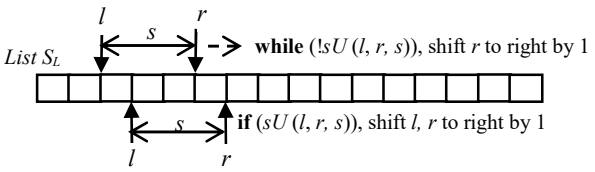

Figure 5. Traversal of list S_L

The objective of GKS is to collect s unique keywords from query Q in the sub-tree of a GKS node. However, it is possible that not all s keywords in the continuous block of length s are unique. Therefore, we first collect a block such that there exist s unique keywords in it, as shown in Figure 5. For a block of length s, let l and r represent the left and right end of the block respectively. Function sU (l, r, s) returns true if there are s unique keywords in the range of l to r (with the aid of hash tables, Section 2.4). Until sU(.) is true, we just move r to the right, keeping l fixed. When sU(.) is true, range l and r represent a block containing s unique keywords. Once the correct block is found, the longest common prefix of the block is added to the LCP list.

We generate the list of LCE nodes from LCP list. For each of the entries in LCP list, we check the *entityHash*, prepared at the time of parsing XML document repository. For each entry in the LCP list, we check if it is an entity node or any of its ancestors is an entity node (using *Dewey id* we can get the Dewey ids of all if its ancestors). If the node (or any of its ancestors) is found to be entity node, we add the corresponding Dewey id into a LCE node list. We also maintain a 'Ranking array' which has an entry corresponding to each GKS node. Each entry in the ranking array maintains two scores as shown in Figure 4. One is the number of keywords $k_i \in Q$ appearing in GKS node sub-tree and the other is its ranking score (computation of the ranking score is described in Section 5). The number of query keywords in the GKS sub-tree is (s+*counter*-1).

Algorithm GKSNodes (Set Q) //Q contains query keywords
Merge the sorted inverted index list $\boldsymbol{S_i}$ for $\forall\ k_i \in Q$ into list $\boldsymbol{S_L}$
//Find Longest Common Prefix (LCP) list
l=0; r=s-1;
Traverse $\boldsymbol{S_L}$ from left to right
while (!sU (l, r, s)) r++; *//Identify block of s unique entries*
Find longest common prefix (LCP) of s unique entries;
Add LCP to LCP list;
if (sU (l, r, s)) r++; l++;
for each entry e_n in LCP list *//Find LCE node list from LCP list*
e_c = null;
if (isEntity (e_n) > 0)
Add e_n to LCE node list; $e_c = e_n$; Remove e_n from LCP list;
else if (any ancestor $e \prec_a e_n$ & isEntity(e) & $e \notin$ LCE node list)
Add e to LCE node list; e_c=e; Remove e_n from LCP list;
if (e_c != null)
for ($\forall\ e \prec_a e_c$)
if (isEntity(e) > 0 & $e \in$ LCE node list)
Update LCE node (e);
Rank nodes in LCE/LCP node lists;
return ranked LCE/LCP lists;

Figure 6: GKS algorithm for finding XML nodes

Example 4: In Figure 4, d_{id}.0.1 is the longest common prefix (LCP) of block of first s nodes. Its entry is created in the LCP list with *counter* set to 1. In Figure 4, node d_{id}.0.1 is found to be entity node. An entry for it is created in LCE node list with keyword counter set to (s+*counter*-1=2). Similarly, the next entry in LCP list d_{id}.0.1.1.0 is initiated with counter set to 1. While checking its ancestors, node d_{id}.0.1 is found to be an entity node. Since d_{id}.0.1 already exist in the LCE node list, its entry (i.e., number of keywords in its sub-tree) is updated to 3 (since node d_{id}.0.1.1.0 appears in its sub-tree). Finally, the keyword count of node d_{id}.0.1 is incremented to 4 and for node d_{id}.0.1.1.0 to 3 (due to next keyword with Dewey id d_{id}.0.1.1.0.4). Once the LCE nodes list is computed along with the number of keywords in its sub-tree, we compute a ranking score r_i for each LCE node, as explained in Section 5. It is also possible that for some node in LCP list, no corresponding LCE node is found.

4.2 Correctness and Time Complexity

In this section, we present the analysis of our method and prove the correctness of our methodology to discover the LCE nodes.

For LCE node e, there must exist at least one keyword $k \in Q$ that is not contained in any other entity node within its sub-tree (Def. 2.2.1). k is called the independent witness of node e. Correctness is defined as discovering LCE nodes according to Def. 2.2.1. We now prove the correctness of our methodology to discover LCE nodes. To discover LCE nodes, LCP list is traversed from left to right. Let left and right pointers l and r of the current block under consideration are at position p_1 and p_2 respectively in list S_L when entity node e is first time being added to the LCE list.

Lemma 4: For entity node *e,* just being added to the LCE list, *only* Dewey id of the keyword at position p_1 or at position p_2 can be the smallest Dewey id which is independent witness for node *e.*

Proof: Omitted. □

Let *k* be the independent witness for node *e* with smallest Dewey id. We associate the Deweyid of keyword, *k,* with node *e.* Let e_n be the LCE node added immediately after node *e* in the LCE list. If $e \prec_a e_n$ and $e_n \prec_a k$, the entity node *e* is removed from the LCE list. The reason is: *k* is the earliest keyword in document order which was an independent witness for node *e* at the time of its addition to LCE list. Since *k* itself appears in the sub-tree of its descendent entity node e_n, *e* is left with no independent witness and hence removed from the LCE list. Note, *e* can come back in LCE list if any other keyword is found to be an independent witness for it later in list S_L. Any entity node *e* that survives has at least one independent witness. In Figure 4, entry $d_{id}.0.1$ survives in LCE node list at the time of addition of $d_{id}.0.1.1.0$ since it has an independent witness. If any entity node *e,* $e \prec_a e_n$, of a newly added entity node e_n remained in the LCE list, we update its ranking array.

Lemma 5: For each LCE node *e* that survives in LCE node list, there exists a keyword *k* that is an independent witness of *e.*

Claim 1: Let *e* be a lowest common entity node for a block of *s* keywords. We claim that at least one of the keywords in the block is an independent witness for node *e.*

Proof: Proof is by contradiction. Suppose no keyword in the block of *s* keywords is an independent witness for LCE node *e.* Hence there must exist another LCE node in the sub-tree of *e*, which contains all the keywords from the block. Hence node *e* is not the *lowest* common entity node, contradicting the initial assumption. □

Claim 2: Any ancestor entity node *e,* of entity node e_n, which is *not* already present in the LCE node list at the time of addition of node e_n in the LCE list, is *not* the LCE node.

Proof: As entity node *e,* $e \prec_a e_n$, is not in LCE list, therefore it has no independent witness keyword at the time of addition of e_n. Since e_n is the lowest entity node for the current block of keywords, *e* cannot be an LCE node. □

Thus, LCE node *e* that survives in LCE node list, there exists a keyword that is an independent witness. When the traversal of LCP list is complete, LCE list contains only true LCE nodes (Def. 2.2.1).

Time complexity to generate the longest common prefix list is $O(d.|S_L|)$ due to Lemma 6 below (the worst case time complexity could be $s.d.|S_L|$ where *s* is a small constant). Since the Dewey ids are sorted, we just need to find the longest common prefix of first and last Dewey id in the block of *s* Dewey ids. There are $O(|S_L|)$ entries in longest common prefix list and depth of the document is *d*. Hence, time complexity to generate LCE nodes list is $O(d.|S_L|)$.

Lemma 6: For lexically sorted block of *s* strings, the common prefix of first and last string is the longest common prefix for the strings in the block. □

As the time complexity to generate merged Dewey id list is $O(d.|S_L|.log\ n)$, total time complexity to generate LCE node list, along with its ranking score list is $O(d.|S_L|.(log\ n))$. Therefore, we efficiently identify LCE nodes in a single pass.

For the LCA based search, the time complexity of the state of the art algorithm to find LCA nodes for query *Q,* $|Q|=n$, is $O(d.n.|S_{min}|.log|S_{max}|)$ where $|S_{min}|(|S_{max}|)$ is the length of the shortest (longest) inverted index list consisting the DeweyId of the keyword in query *Q* [6][16]. We see that the time complexity of our algorithm to find the GKS nodes is only marginally worse than the time complexity to find LCA nodes, even though the search space for GKS is exponential compared to LCA nodes.

Nodes in Longest Common Prefix (LCP) list contain at least *s* keywords in their sub-tree. For each node *u* in LCP list we keep a mapping with its associated LCE node *v* in LCE list, $v \lhd_a u$. There may exist some nodes in LCP list such that no corresponding entity node is found for them due to the structure of the XML data.

The XML nodes in LCE list along with those nodes in LCP list for which *no* corresponding LCE node exist together constitute the GKS response $R_Q(s)$. These nodes are ranked with the aid of ranking function presented in the next section.

5. RANKING

Node ranks help GKS construct a more meaningful response. Number of GKS nodes can be large and the response may comprise a variety of XML node types. The relevance of these nodes varies in the context of a given query. For LCA based techniques, each LCA node is the common ancestor of *all* the keywords in query *Q*.

Due to the basic differences between GKS and LCA based search, existing ranking algorithms [8][15] are insufficient for GKS. Existing ranking methods work by using aggregated statistical information for entire XML repository. For a fixed set of keywords, nodes are ranked based on statistical relevance of a query keyword in the context of a given XML node. For GKS, any node containing a subset of keywords belonging to query (of size $\geq s$) is the node of our interest. Further, GKS response may contain a variety of differently structured XML nodes. Therefore, any statistical method is insufficient to compare the relevance of one XML node w.r.t. other due to the structural difference in their sub-trees.

Therefore, we introduce a novel ranking function that computes the rank of each XML node in $R_Q(s)$ for query *Q* based on i) the number of keywords from *Q* appearing in its sub-tree; and ii) the structure of the sub-tree rooted at that node.

5.1 Ranking Methodology

We use a potential flow model to compute the rank of the XML nodes in $R_Q(s)$. Potential of a node is like the amount of water present in a reservoir which flows in a network of pipes coming out from it. The potential flow model automatically incorporates the structure of the sub-tree rooted at an XML node.

We assign an initial potential, $P|e$ to each node $e \in R_Q(s)$. $P|e$, for node *e* is equal to the number of unique query keywords $k \in Q_s$, $Q_s \subseteq Q$, present in its sub-tree.

$$P|e=|Q_s|;\ Q_s \subseteq Q\ ;\ Q=\{k_1,\ldots,k_n\}$$

$P|e$ just accounts for the presence of a keyword $k \in Q$ in the sub-tree of node *e*. If the keyword *k* is present multiple times in node *e*, only its highest occurrence in its sub-tree is considered. This highest occurrence of a query keyword in the sub-tree is termed *terminal point.* If a keyword *k* is present multiple times at the highest level, *each* of its occurrences is considered a terminal point. For example, if a keyword $k \in Q$ is a repeating XML element name in the sub-tree of an LCE node, each of its occurrence will be considered as a terminal point (assuming that is the highest level at which keyword *k* occurs). Hence, for a user query $Q=\{k_1 \ldots k_n\}$, each candidate XML node has a starting potential. As shown in Figure 7, for node e_1, the highest occurrence of keywords k_1, k_2, k_3 are *terminal points.*

The rank of a node $e \in R_Q$ is computed as follows: The potential of a node *e* is equally divided into each of its child nodes. For a node

e with potential ($P|e$), with m children, each of its child nodes will receive ($P|e$)/m potential, where m is the number of direct child the node e has. The rank of the node is sum of the total potential received by each of the terminal points.

Let $i \rightarrow k$ denotes the relationship that node i is parent of node k. The rank of an entity node e is computed as follows:

$$Rank_e = \sum_{k \in Q} \frac{p_i}{m_i} \mid i \rightarrow k$$

where k is a terminal point in the sub-tree of node e, p_i is the potential received by its parent node i and m_i is the total number of direct children node i has. The potential received at terminal points depends on the structure of the sub-tree rooted at the XML nodes.

Intuitively, it implies that the number of distinct query keywords in its sub-tree and the structure of its sub-tree determine the rank of an XML node. Each LCE node is ranked independently, irrespective of its relative depth w.r.t. document root.

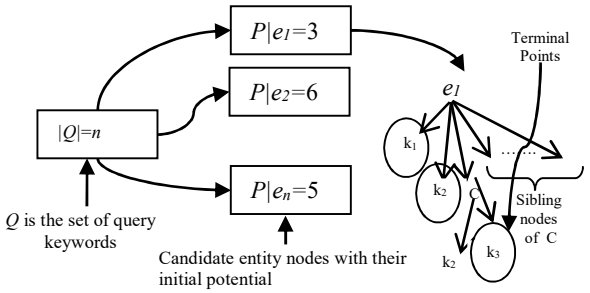

Figure 7. LCE nodes containing keywords in set $R_Q(s)$

Example 5: We illustrate the computation of XML nodes rank with the example XML document shown in Figure 1. For query Q_3= {*a, b, c, d*}, GKS returned 3 XML nodes x_2, x_3 and x_4. The initial potential of node x_2 is $P|x_2$=3. The rank of node x_2 is the potential received by the terminal nodes in its sub-tree, i.e., nodes a_2, b_1, c_1. The rank of node $x_2 = \sum_{k \in Q} P \mid x_2 / n = 3 \times 3/3 = 3$. Similarly, for node x_3, the initial potential $P|x_3$ is 3. The three terminal nodes are nodes a_3, b_3, d_3. Each of the three children of node x_3 received 1/3rd of the initial potential. The potential received by x_4 is further divided equally into its two children. Therefore, the total potential received by the terminal nodes, i.e., the rank of node x_3 is, $2 \times 3/3 + 3/3 \times 1/2 = 2.5$. Similarly, the rank of node x_4 is 2. Hence, GKS ranking methodology ranks the nodes as $x_2 > x_3 > x_4$.

6. SEARCH RESULTS ANALYSIS

GKS enables the users to refine their queries. The user query Q can be refined by either removing or adding the most relevant keywords to Q, in the context of the query. We now describe how GKS aids the users to refine their query by analyzing the search response.

6.1 Query Refinement

Let us consider the Example 1. For query Q_3 = {*a, b, c, d*}, the response of GKS comprised nodes, x_2, x_3 and x_4. GKS ranks the nodes such that most relevant nodes are ranked higher. The two top ranked nodes are x_2 containing keywords {*a, b, c*} followed by x_3 containing keywords {*a, b, d*}. With this information, the user is exposed to the fact that there is no XML node that contains all the query keywords and that the distribution of the query keywords in the document is as shown in the query results. With this insight, users can refine their queries. In the example above, user can refine the query Q_3 to {*a. b, c*} or {*a, b, d*} given the GKS response.

Therefore, for a user query Q, the query can be refined seamlessly to one or more sub-queries Q_r*s* with the aid of the GKS results. As one can see, for LCA based techniques [2][5][17], such refinement of the query Q is non-trivial as multiple sub-queries of Q needs to be run to collect the complete response.

6.2 *DI*-Discovery from the LCE Nodes

GKS enables the discovery of *DI* from the XML data in the context of the user query which can be used to refine the user query. For a given query, the attribute nodes of a LCE node expose the context for the keywords appearing in its sub-tree and are regarded as the relevant *DI* (Def 2.3.1).

A natural way to discover *DI* is by identifying *top-m* most popular attribute keywords in the LCE nodes present in the query response, i.e., identifying keywords that appear in maximum number of attribute nodes (m is tunable). At the same time, the *DI* must be relevant for most of the query keywords. However, these two goals may translate into two different set of top *DI* keywords. In response of the query in Example 2 (Section 1.2), the most popular keyword is found to be <booktitle: ICPP>. However, the keyword became most popular due to presence of keyword 'Prithviraj Banerjee'. He is the only author who had published articles in this journal but total number of articles by him alone in this journal made it the most popular keyword in the query response. However, this keyword is not relevant for majority of the other query keywords. The keyword <journal: SIGMOD Record> is relevant for the largest sub-set of the query keywords (for remaining three authors) but it is not *the* most popular. Therefore, to identify most relevant keywords in the context of a query, we adopt the following approach.

Rank of a LCE node is the function of number of query keywords present in its sub-tree. Each attribute node is assigned a weight equal to the rank of its LCE node. Therefore, each keyword in set S^w_Q is assigned its attribute weight and we prepare a weighted set S^w_Q. Let $E_Q \subseteq R_Q(s)$ be a set of all the entity nodes in $R_Q(s)$.

$$S^w_Q = \{k : w \mid w = \sum_{k \in R(e)} Rank_e ; e \in E_Q \wedge k \in attr(e) \wedge k \notin Q\}$$

Each element of set S^w_Q is a tuple k: w, where k is the attribute keyword. k is assigned a weight that is sum of the rank of all the LCE nodes in set E_Q that contain k. The *top-m* most weighted keywords constitute *DI*. If a keyword in the attribute node is part of the user query Q, it is not included in the set S^w_Q. We identify *top-m* elements in set S^w_Q, total time complexity to identify *DI* is $O(|S^w_Q| + m.log|S^w_Q|) = O(|S^w_Q|)$ as $|S^w_Q| >> m$. Since $|S^w_Q| = O(|R_Q(s)|)$ and $|R_Q(s)| \leq S_L$, the time complexity to identify *DI* is better by a factor of $O(log|Q|)$ compared to the time complexity to identify LCE nodes and *DI* discovery does not constraint the system. In Example 2, *DI* contained <year: 2001>, <booktitle: ICPP>, <author: Alok N Choudhary>, etc., as top *DI* keywords. Recursive *DI* can be discovered by preparing a keyword query using the text keywords identified from S^w_Q. The recursive *DI* may reveal deeper insights.

Therefore, a user query Q can be refined seamlessly to Q_r with the aid of *DI*. We see that with the aid of response produced by GKS and with the aid of *DI*, user queries can be refined by adding or removing the keywords from the initial keyword query.

7. EXPERIMENTS

We have built a prototype of GKS [20]. Observations in this section are based on experiments using this prototype over the XML data sets shown in Table 4 [21]. Shakespeare's plays are distributed over multiple files. The experiments were carried out on a Core2 Duo 2.1GHz, 4GB RAM machine running Windows 7 and Java. These data sets are used in many prior works [3][11][13][19]. The size of

Protein Sequence dataset is comparable to the biggest dataset used in a recent work [19]. Our DBLP dataset size is 100% bigger.

Our experiments are designed to assess 1) Performance of GKS; 2) Appropriateness of the node categorization model given the real world XML repositories; 3) Effectiveness of GKS in finding the relevant results for keyword queries and to rank them; 4) Ability of GKS in finding the relevant *DI*; 5) User feedback.

7.1 Performance of GKS

7.1.1 Size of Index

Creating the index is a onetime activity. The size of index and the time taken to prepare them are presented in Table 4. *Our technique is highly scalable as index preparation time increases linearly with the data size*. The number of entity nodes for different datasets varied from 535 for Mondial to 2.62M for DBLP.

Table 4. Index Size and Index Preparation Time

Data Set	Data Set Size	Index Size	XML Depth	Index Preparation Time
SIGMOD Records	483KB	416KB	6	0.15s
Mondial	1.7MB	1.45MB	5	0.28s
Plays	1.8MB	1.6MB	5	0.29s
TreeBank	82MB	79MB	36	19.3s
SwissProt	112MB	101MB	8	21.3s
Protein Sequence	683MB	612MB	7	108s
DBLP	1.45GB	1.13GB	6	238s

7.1.2 Response Time

In this experiment, we assess the response time (RT) of GKS for given user queries. We also validate GKS time complexity analysis to discover the XML nodes for the given queries. We give RT results for two datasets: i) NASA dataset containing astronomy data (24MB) and ii) SwissProt dataset containing protein sequence data (112MB). In our first experiment, the number of keywords for each query (n) was fixed at 8. However, the size of the merged Dewey id list (S_L) varied for each query. The average keyword depth d for the NASA dataset varied from 6.7-6.9 and from 3.1–3.5 for SwissProt dataset. Results are presented in Figure 8. As shown in Section 4.2, for given d and n, the RT increases linearly with S_L. Response time varies from 21.5ms to 139ms for different queries. Hence, the RT of GKS is only a few tens of ms, similar to LCA based algorithms on similar data.

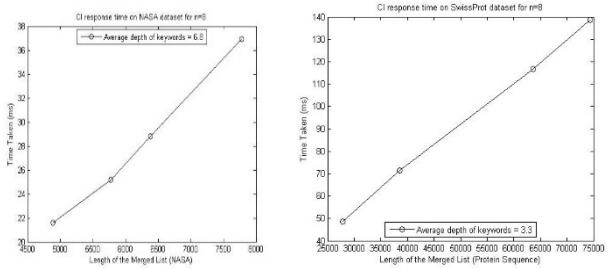

Figure 8. Response Time Vs. Merged List Size

In Figure 9, we plot the RT for queries by varying the number of query response keywords n from 2 to 16. The query response time validates our analysis (cf. Section 4.2). The list size $|S_L|$ for query on SwissProt dataset with n=16 was 102,233. In Figure 9, for the NASA dataset, when n is increased from 8 to 16 for a query, the increase in RT was less than twice, as the length of the list $|S_L|$ increased only marginally and the change in RT is logarithmic in n.

For a query run on the DBLP dataset, the RT was found to be 2ms for $|S_L|$= 213. Hence, *RT depends on the query, i.e., depth d, n and S_L (O(d. $|S_L|$.log n)), and not on the size of the data being queried.*

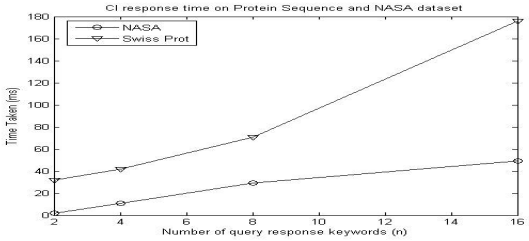

Figure 9. Response time vs. keywords in query response (n)

7.1.3 Scalability

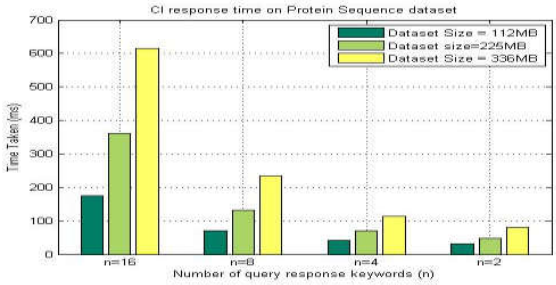

Figure 10. Response time for different dataset sizes

To assess the scalability of GKS, we replicated the Swiss Prot dataset to create three datasets of size 112MB, 225 MB and 336 MB. For same query, the number of LCE nodes scales linearly. In Figure 10, we plot our results. We can see that *query processing time is scaling linearly with data size*, as expected.

7.2 Validation of Node Categorization Model

In this experiment, we analyze the structure of the real world data repositories. XML nodes are placed in different categories as described in Section 2.2. In Table 5, we show the number of different XML element types belonging to different node categories for various XML repositories. As we see, the fraction of nodes labeled as Connecting Nodes (CN) varies from around 15% for InterPro to less than 3% for DBLP dataset. In DBLP/Sigmod Records some nodes with similar schema as that of *entity nodes* (EN) are marked as CN because of the presence of just a single author. We compared the results of our analysis with the ground truth, i.e., with XML schema. For Sigmod Records, two XML elements, <articles> and <authors>, were the connecting nodes as per the XML schema. The count of <authors> node, was 1504, and count of <articles> node was 67 (remaining 447 XML nodes were marked CN due to presence of <article> nodes with a single author). The results show *our node categorization model captures the structure of the real world data repositories very well.*

Table 5. Distribution of XML element

Data Sets	Count of AN	Count of EN	Count of RN	Count of CN	Total Nodes
Sigmod Record	10574	1022	3766	2018	15263
DBLP	27.58M	2.62M	10.56M	972367	39.52M
Mondial-3.0	7467	535	15074	663	22423
InterPro	515316	32614	1472021	303079	2088766
SwissProt	4044884	176128	1776676	187300	5166890

7.3 Finding and ranking the XML nodes

The purpose of this experiment is to assess the quality of GKS results. Therefore, some queries are designed for which SLCA

response is obviously inadequate (the response of an SLCA technique is either null or document root for these queries). For a few queries, SLCA returns meaningful response (QM1, QI1, QI2). Queries for different datasets are shown in Table 6. The number of keywords for the queries was varied from 2 to 8. We present the number of XML nodes retrieved by GKS and SLCA. We vary *s,* the minimum number of query keywords in the XML node sub-tree. *s* is set to be 1 and $|Q|/2$ respectively.

Table 6. Keyword queries run on different datasets

	$\|Q\|$	**SIGMOD Records**
QS1	2	"Anthony I. Wasserman" "Lawrence A. Rowe"
QS2	4	"S. Jerrold Kaplan" "Robert P. Trueblood" "David J. DeWitt" "Randy H. Katz"
QS3	6	"Sakti P. Ghosh" "C. C. Lin" "Timos K. Sellis" "David A. Patterson" "Garth A. Gibson" "Randy H. Katz"
QS4	8	"Barbara T. Blaustein" "Umeshwar Dayal" "Alejandro P. Buchmann" "Upen S. Chakravarthy" "M. Hsu" "R. Ledin" "Dennis R. McCarthy" "Arnon Rosenthal"
	$\|Q\|$	**DBLP**
QD1	2	"Dimitrios Georgakopoulos" "Joe D. Morrison"
QD2	4	"Peter Buneman" "Wenfei Fan" "Scott Weinstein" "Prithviraj Banerjee"
QD3	6	"E. F. Codd" "Mark F. Hornick" "Frank Manola" "Alejandro P. Buchmann" "Dimitrios Georgakopoulos" "Joe D. Morrison"
QD4	8	"E. F. Codd" "Kenneth L. Deckert" "Irving L. Traiger" "Vera Watson" "Jim Gray" "Chin-Liang Chang" "Nick Roussopoulos" "Jean-Marc Cadiou"
	$\|Q\|$	**Mondial**
QM1	2	country Muslim
QM2	3	Laos country name
QM3	6	Polish Spanish German Luxembourg Bruges Catholic
QM4	8	Chinese Thai Muslim Buddhism Christianity Hinduism Orthodox Catholic
	$\|Q\|$	**InterPro**
QI1	2	Kringle Domain
QI2	3	Publication 2002 Science

Results for different queries are shown in Table 7. We see a large number of XML nodes returned for the queries by GKS (s=1) compared to SLCA response. Thus, a lot of information that could be of the interest to the user for the given keyword query is not returned by LCA based techniques. Further, the number of XML nodes for (s=$|Q|/2$), is non-zero for all the queries. When we compared GKS with FSLCA [19], the top XML node for both QI1 and QI2 for GKS was present in FSLCA result set. For QM1, many XML nodes of FSLCA were among the top 10 nodes of GKS results. For QM2, no FSLCA node exists but GKS was able to find the XML nodes having subset of query keywords. There are no entity nodes which were relevant, i.e., contained at least *s* query keywords, but not identified by GKS. Therefore, *GKS is able to find valid response for the given user queries, without binding them to LCA framework, enhancing the users' ability to search the data*.

We next assess the ability of GKS to rank the discovered XML nodes. Since, the schema for DBLP and Sigmod Records is not very deep, the structure of all the XML nodes is similar for these two datasets. Hence, we adopt the measure that the higher the number of query keywords in an XML node sub-tree, the more relevant it is. Given this measure, we determine where GKS places the XML nodes with the highest number of query keywords in its ranking list.

Let L be a ranked list of XML nodes, in set $R_Q(s)$, returned by GKS for a query Q and for a given s, $|L| = p = |R_Q(s)|$. The nodes are numbered 1 to p according to their ranks. The XML nodes that contain the highest number of keywords from query Q in their sub-tree are called the true XML nodes. Let L' (L' $\subseteq$ L) be a set of true XML nodes. Let w be the lowest rank of a true XML node in the list of XML nodes L. To each true XML node, we assign a weight of (w+1-i) where i is the rank of the true XML node in the list L. We compute the aggregated weight of true XML nodes as $w_a = \sum_{i \le w;} (w+1-i)$ for all the true XML nodes in the list L. The total score is computed as $w_t = w(w+1)/2$. Finally, we compute a rank score as w_a/w_t. We penalize the rank score if a true XML node occurs lower in the list L. Score of 1 means that no true XML node is ranked lower than a XML node which is not in set L'. In Table 7, we show our results. The ranking score is computed for GKS response when s=1. We see that *the aggregate weight, i.e., rank score is very high for all the queries*. For every query, except QM3, the top-most result is always a true XML node. For QM3, it appeared at 3rd position.

Table 7. Comparison with SLCA and Rank Score

Query	**#GKS, s=1**	**#GKS, s=$\|Q\|/2$**	**SLCA**	**Max keywords in a GKS node**	**Rank Score**
QS1	8	NA	0	1	1
QS2	43	13	0	2	1
QS3	28	4	0	3	1
QS4	36	2	1	8	1
QD1	30	NA	1	2	1
QD2	234	10	0	3	0.72
QD3	190	7	0	5	1
QD4	267	4	0	6	1
QM1	230	NA	98	2	1
QM2	234	NA	1	2	1
QM3	37	4	0	3	0.17
QM4	116	3	0	6	1
QI1	8170	NA	8	2	0.893
QI2	2517	2517	281	3	1

In summary, we see that *GKS is able to retrieve and appropriately rank the relevant XML nodes, in the context of the user queries*.

7.4 Quality of *DI* Discovered by GKS

One of the most important attractions of GKS is its ability to discover *DI* in the data. In Table 8, we show the *DI* discovered for the queries in Table 6 for different values of s. This experiment highlights that the *DI* discovered by GKS is highly relevant for the given queries. For instance, for QD3, *DI* exposes the most relevant year (1999) and the most relevant 'booktitle' (ICCD). The keywords exposed as *DI* also help users understand GKS response since the *DI* keywords also represent the summary of the query response. For some queries, *DI* varies for different values of s.

Table 8. *DI* discovered for different queries

Query	***DI, s=1***	***DI, s=$\|Q\|/2$***
QS1	<title: Third-Generation Database System Manifesto >	<title: Cache Consistency and Concurrency Control>
QS2	<title: Chair's Message>	<title: Database Research Activities at the University of Wisconsin >
QS3	<title: article title>	<title: Implementation of a Prolog-INGRES Interface>
QS4	<title: article title>	**NA**
QD1	NA	<year:2000>,<journal:TCS>
QD2	<year: 2001>, <journal: SigmodRecords>	<year:1998>, <volume:2>
QD3	<year: 1999>, <booktitle: ICCD>	<journal: TCS>, <year: 2001>, <number: 1>
QD4	<year: 2001>, <journal: JACM>	<journal:IBM Research Report>, <year: 2001>

QM1	<country:f0_475>, <Year : 90>	**NA**
QM2	<Name : Zimbabwe>, <population_growth : 1.41>	<Name:Zimbabwe>, <percentage : 100>
QM3	<country:f0_337>,<year: 90>	<country:f0_337>,<year:90>
QM4	<country:f0_663>, <percentage:100>	<country:f0_663>,<name:Brunei>
QI1	<author_list:Patthy L>, <taxon_data_name:Eukryot>	**NA**
QI2	<taxon_data_name:"Bacteria>, <proteins_count : "1">	**NA**

We now show, with the aid of query QD1, how *DI* helps refine queries. For QD1, GKS returned a total of 30 XML nodes (s=1). The *DI* was <author: Marek Rusinkiewicz>. After analyzing the query response, QD1 is refined to ("Dimitrios Georgakopoulos", "Marek Rusinkiewicz"). Interestingly, for the refined query we found that there were 10 articles jointly written by these two authors as opposed to just 1 joint article by authors in original query. This is an example of how *GKS helps users refine their queries and guides them to navigate the data by recursive application of GKS.*

7.5 Crowd-Sourced Feedback: GKS & SLCA

We asked 40 users to compare the GKS response with SLCA response on a scale of 1-4; 1 being 'GKS Very Useful' and 4 being 'SLCA Very Useful'. Results are shown in the table below. For almost all the queries, *GKS* response is found to be either very useful (**1**) or better than SLCA (**2**). If we categorize the response as 'GKS-better' (rating 1 or 2) and 'SLCA-better' (rating 3 or 4), 430 out of 480 responses found the *GKS* response better (89.6%).

Query	1	2	3	4
QS1	24	16	0	0
QS2	17	22	1	0
QS3	17	14	5	4
QS4	12	22	3	4
QD1	24	15	1	0
QD2	18	17	3	2
QD3	20	18	1	1
QD4	24	13	3	0
QM1	16	20	3	1
QM2	15	18	5	2
QM3	14	18	5	3
QM4	16	21	3	0

7.6 GKS Performance for Hybrid Queries

We further studied how well GKS behaves in the presence of clearly separable keywords in the query, i.e., subsets of the keywords in a query indeed refer to different entity type nodes. We call such queries 'hybrid queries'. To study GKS performance for hybrid queries, we merged DBLP and Sigmod Record datasets into a single dataset (with a 'common root'). We also increased the depth of Sigmod Record elements by introducing two connecting nodes between the 'common root' and the root of Sigmod Record data. We ran the query "`Jean-Marc Meynadier" "Patrick Behm" "Lawrence A. Rowe" "Michael Stonebraker`", s=2. First two authors appeared together only in `<inproceedings>` entity node type in DBLP dataset and last two in `<article>` entity node type in Sigmod Record. Clearly, the keywords in the query target two different XML node types. GKS was able to return all 8 corresponding XML nodes present in our dataset, 3 `<inproceedings>` nodes in DBLP data by first 2 authors and 5 `<article>` nodes in Sigmod Record data by last 2 authors. Thus, *GKS returned correct response even when multiple XML node types were targeted by a single query.* Note, only these 8 nodes were returned by GKS.

Further, two `<article>` nodes, by last two authors, were ranked higher (as they were the only authors in all the three articles) despite higher relative depth w.r.t. root (articles by first 2 authors had multiple other authors). Hence, *the entity nodes are ranked based on only the number of query keywords present in their sub-tree and the distribution of these keywords*, and not according to their absolute depth in the XML tree, as analyzed in Section 5.

Summary: In summary, experiments show that GKS is scalable, imposes low overhead and retrieves the XML nodes efficiently. The experiments validate our node categorization model and show that XML nodes and *DI* discovered by GKS are highly relevant

8. CONCLUSION

We presented a novel system GKS that enables generic keyword search over XML data and yields highly meaningful response without imposing the AND-Semantics of LCA based techniques. We show that our system exposes deeper analytical insights (*DI*) in the data in the context of user queries. GKS exploits the XML schema, embedded in the XML data, in the context of the query to find the most relevant data keywords and schema elements with the aid of a novel node categorization model. In conjunction with a novel XML node ranking method, GKS is able to expose the *DI* elegantly. One of our future research direction is to extend GKS to enable analytics over raw XML data.

9. REFERENCES

[1] I. Tatarinov, et al., "Storing and querying ordered XML using a relational database system", in SIGMOD, 2002.

[2] Y. Xu, Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases", in EDBT 2008.

[3] Z. Liu, Y. Chen, "Identifying Meaningful Return Information for XML Keyword Search", in SIGMOD 2007.

[4] J. Li, C. Liu, R. Zhou, W. Wang, "Suggestion of promising result types for xml keyword search", in EDBT, 2010.

[5] R. Zhou, C. Liu, J. Li, "Fast ELCA Computation for Keyword Queries on XML Data", in EDBT 2010.

[6] L. Chen, Y. Papakonstantinou, "Supporting Top-K Keyword Search in XML Databases", in ICDE 2010.

[7] L. Guo, et al., "XRANK: Ranked Keyword Search over XML Documents", in SIGMOD 2003.

[8] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv, "XSEarch: A Semantic Search Engine for XML", in VLDB 2003.

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

[10] H. Cao, et al., "Feedback-driven Result Ranking and Query Refinement for Exploring Semi-structured Data Collections", in EDBT 2010.

[11] Z. Liu and Y. Chen. "Reasoning and identifying relevant matches for xml keyword search", in PVLDB, 1(1), 2008.

[12] L. Kong, R. Gilleron, A. Lemay. "Retrieving meaningful relaxed tightest fragments for xml keyword search", in EDBT, 2009.

[13] Y. Xu, Y. Papakonstantinou. "Efficient keyword search for smallest lcas in xml databases", in SIGMOD, 2005, pp 537-38.

[14] M. Arenas, "Normalization Theory for XML", in SIGMOD Record, Vol. 35, No. 4, December 2006.

[15] Z. Bao, T. Ling, B. Chen, J. Lu, "Effective XML Keyword Search with Relevance Oriented Ranking", in ICDE 2009.

[16] J. Zhou et al., "Efficient query processing for XML keyword queries based on the IDList index", The VLDB Journal, February 2014, Volume 23, Issue 1, pp 25-50.

[17] J. Zhou et al., "Fast SLCA and ELCA Computation for XML Keyword Queries based on Set Intersection", in ICDE 2012

[18] Manish Bhide, Manoj K. Agarwal, et. al., "XPEDIA: XML Processing for Data Integration", in VLDB 2009. .

[19] B. Truong, et al., "MESSIAH: Missing Element Conscious SLCA Nodes Search in XML Data", in SIGMOD 2013.

[20] Manoj K Agarwal, Krithi Ramamritham, "Enabling Generic Keyword Search over Raw XML Data", ICDE, 2015, pp 1496-99.

[21] http://www.cs.washington.edu/research/xmldatasets/www/repository.htm