SpiderX: Fast XML Exploration System

Chunbin Lin, Jianguo Wang
Computer Science and Engineering, California, San Diego
La Jolla, California, USA
chunbinlin@cs.ucsd.edu, csjgwang@cs.ucsd.edu

ABSTRACT

Keyword search in XML has gained popularity as it enables users to easily access XML data without the need of learning query languages and studying complex data schemas. In XML keyword search, query semantics is based on the concept of Lowest Common Ancestor (LCA), e.g., SLCA and ELCA. However, LCA-based search methods depend heavily on hierarchical structures of XML data, which may result in meaningless answers. To obtain desired answers, a successful system should be able to (i) match a semantic entity for each keyword, (ii) discover the relationships of the matched entities, (iii) support efficient query processing, (iv) release users from having the knowledge of the XML content, and (v) visualize the search results. None of the existing XML keyword search systems completely meet the above requirements.

In this paper, we design a system called SpiderXto completely solves the above challenges. We propose a query semantics *Entity-Relationship Graph (ERG)*, which adopts the RDF subject-predicate-object semantics to capture the information of search entities along with associated attributes and the relationships between entities. SpiderX proposes a novel index structure, which has small space cost by combining the optimizations of column databases and the data compression schemes. In addition, SpiderX processes queries in a bottom-up way to achieve high performance, which is about 100× faster than the state-of-the-art algorithms. To demonstrate the high performance of SpiderX, we implement an online demo for SpiderX, which operating on three real-life datasets. The demo also provides (1) query auto-completion to guide users to formulate queries; and (2) visualization panel to display the query answers, which interacts with users by providing zoom-in and zoom-out exploration features. Demo link: `http://chunbinlin.com/spiderx`.

1. INTRODUCTION

Keyword Search provides a user-friendly information exploration mechanism for users to easily access XML data without the need of learning query languages (such as X-Path and XQuery) or studying complex data schemas [3, 4, 15, 12, 13, 1]. However, due to the lack of expressivity and inherent ambiguity, it is challenging to capture the semantics on keyword queries.

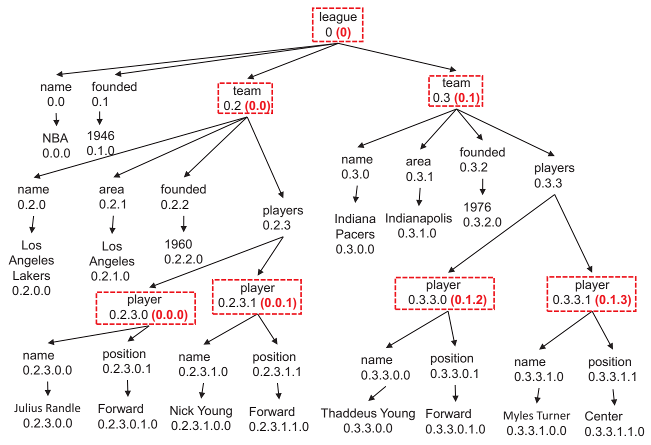
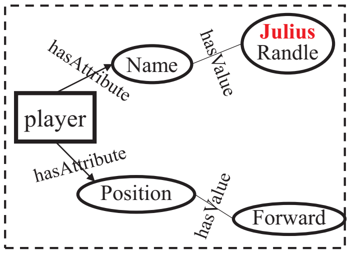


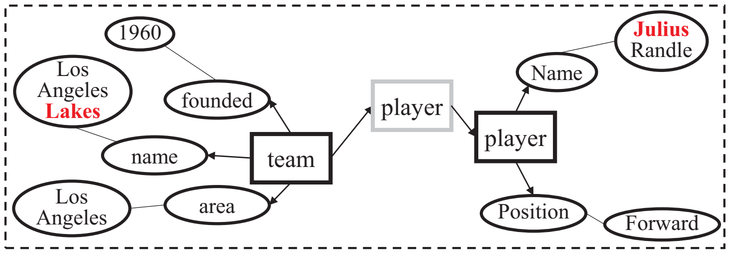
Figure 1: An example of NBA XML data. Each node is associated with a label (in black fonts) in existing systems. The nodes in red rectangles are entity nodes. SpiderX only assigns labels for entity nodes (in red fonts).

Existing XML keyword search systems use an LCA (Lowest Common Ancestor) node, or its variants such as SLCA [15], ELCA [4], of keyword match nodes as an answer to a keyword query. However, the LCA-based approaches only rely on the hierarchical structure of the XML tree and ignore the semantics of queries, which results in returning incomplete answers, overwhelming answers or meaningless answers [11]. For example, the returned answer of query "Julius" on Figure 1 is the node "Julius Randle (0.2.3.0.0)"[1], which is an LCA node. However, it is meaningless to users since the answer does not contain more knowledge than the query. The answer of the same query generated by SpiderX (our system) is shown in Figure 2(a), which is more meaningful as it captures the relevant entity and attributes. More precisely, the "player" is an entity and the "name" and "position" are two attributes of the entity. The answer is organized using the "subject, predicate, object" RDF triple semantics. The

[1]0.2.3.0.0 is a Dewey label [15]. Each node in XML is labeled by a Dewey label, as shown in Figure 1 (in black fonts).



(a) Result of query "Julius"



(b) Result of query "Julius, Lakes"

Figure 2: Query Results in SpiderX. Entities are in rectangles, attributes are in the ovals. The predicates in (b) are omitted.

subject-predicate-object triples in Figure 2(a) are (1) player-hasAttribute-name, (2) name-hasValue-JuliusRandle, (3) player-hasAttribute-position, and (4) position-hasValue-Forward.

To identify semantics on queries, several recent attempts have been made, such as XRank [4], XKeyword [7] and XSeek [13]. They are able to return entity nodes instead of meaningless nodes, but they are weak in expressing the meaningful answers. Note that, a single entity node is far from enough to be a complete and meaningful answer. To return answers, they either use the whole subtrees rooted at LCA nodes as answers, called *Subtree Return* [3, 4, 15], or return the paths in the XML tree from each LCA node to its descendants that match an input keyword as answers, called *Path Return* [7]. However, The *Subtree Return* method outputs overwhelming answers, while the *Path Return* approach yields incomplete answers. For example, the matched entity node of the query "Julius, Lakers" is "term (0.2)". Neither subtree return nor path return gives the compact and complete answer. SpiderX returns the answer for the same query in Figure 2(b), which captures the entities matching keywords, and attributes of the entities and also the relationship between two entities. Compared with subtree return, it omits the other player nodes from the answer to make the answer compact. Compared with the path return, it gathers the attributes of the entities to make the answer complete.

Contributions. The contributions of this paper can be summarized as follows:

- We design an entity-relationship exploration system SpiderX for XML databases, which supports keyword search and organizes answers as Entity-Relationship Graphs (ERGs), which uses the RDF subject-predicate-object semantics.
- We propose a new index by combining the optimizations of column databases and the data compression schemes, which is theoretically smaller than existing Dewey-based indexes.
- We propose a bottom-up algorithm to efficiently answer queries by using the index, which is about 100× faster than the state-of-the-art methods.
- We implement a demonstration system of SpiderX operating on three real life XML datasets.

Paper Organization. The rest of paper is organized as follows. Section 2 gives an overview of SpiderX. We illustrate all the technical details of SpiderX in Section 3. Finally, we provide the demo scenarios in Section 4.

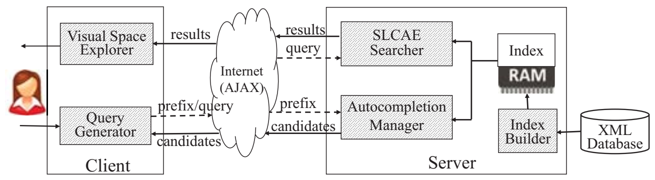


Figure 3: Architecture of SpiderX.

2. OVERVIEW OF SPIDERX

2.1 System Architecture

The system architecture of SpiderX is presented in Figure 3. On the server side, the *Index Builder* creates in-memory indices, i.e., (i) a trie tree for query autocompletion, and (ii) an inverted index for query processing. On the client side, a query is created by the *Query Generator* with the help of the *Autocompletion Manager*, which provides a list of candidate queries. Once a complete query is submitted from the client, the *LCAE Searcher* is triggered to obtain the query results. Those results are then organized as ERGs and sent to the *Visual Space Explorer* for visualization.

2.2 User Interface

The interface for SpiderX is shown in Figure 5, which consists of three parts: the query generator (Figure 5①), autocompletion manager (Figure 5②), and visual space explorer (Figure 5③ and ④).

Query Generator. A query generated by the SpiderX query generator consists of two parts: (1) a drop-down list choosing an XML data source; and (2) a list of keywords input by the user, which can either be typed directly or chosen from a list of candidate queries provided by SpiderX's autocompletion manager.

Autocompletion Manager. Query autocompletion [8] helps users to identify candidate queries when only prefixes are input. It significantly improves the efficiency of query generation since users are only required to input a few characters. In SpiderX, the server maintains a trie tree for all distinct tokens in the main memory. Every time a new character is entered, an Ajax request is sent to the server which looks up the trie and returns a list of candidate tokens with a matching prefix.

Visual Space Explorer The visual space explorer visualizes the search results of the query within the visual space.

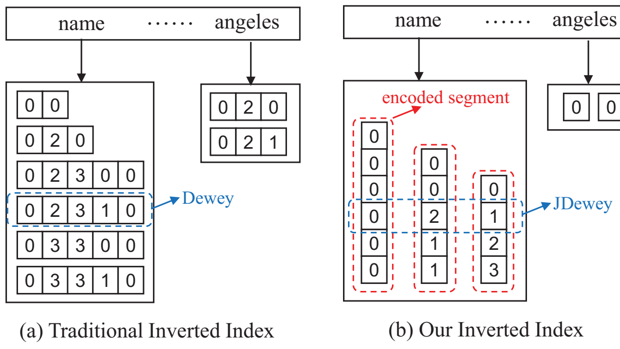


Figure 4: Traditional Inverted Index vs. Our Inverted Index.

Each result is displayed as either a list of (attribute: value) pairs or an entity-relationship graph (ERG), which will be formally defined in the next section. Users can explore this visual space by zooming in/out and dragging the visual space explorer.

3. TECHNICAL DETAILS

3.1 Index Structure

Labeling Scheme. An XML is a rooted and labeled tree. In existing XML keyword search systems, each node (even it is not an entity node) in the XML tree is assigned a Dewey label/ID [15] as a unique ID. (For example, the labels in black fonts in Figure 1 are Dewey labels.) Instead of labeling all the nodes, SpiderX only labels entity nodes. It employs the JDewey label [2], which is an extension of Dewey label. Let j_u be the JDewey of the node u. Let v be a child node of u and it is the j-th nodes from left to right in its current level, then $j_v = j_u.(j-1)$. The JDewey labels for the XML in Figure 1 are shown in red fonts. Note that, SpiderX only stores the labels in red fonts.

Inverted Index. To support efficient XML keyword query processing, inverted indexes are widely built for XML trees. For each distinct value v, there is an inverted list ℓ_v associated with it. ℓ_v contains all the Dewey IDs of the nodes in document order whose names match v. For example, $\ell_{angeles} = \{0.2.0, 0.2.1\}$ as shown in Figure 4(a).

Recall that SpiderX only labels entity nodes with JDewey labels. For each distinct value v, the associated inverted list ℓ_v' contains all the JDewey labels of the lowest ancestor entity node of the nodes whose names match v. For example, $\ell_{angeles}$'=$\{0.0\}$ as shown in Figure 4(b), since the entity node "team" with JDewey label (0.0) is the lowest ancestor entity node of nodes "Los Angeles Lakers (0.2.0.0)" and "Los Angeles (0.2.1.0)". All the inverted lists are organized in a hashmap with keywords as keys and inverted lists as values. Compared with the traditional inverted indexes such as Dewey-based index, the index of SpiderX has smaller space cost.

To further reduce the space cost, we combine the optimizations in column databases and the data compression schemes. More precisely, the JDewey labels in each inverted list are stored vertically, i.e., the i-th JDewey numbers are stored in the i-th column. We call each column a *segment*. Each segment is compressed by a compression method (see Figure 4(b)) [9, 10]. In particular, SpiderX compresses each column with a word-aligned bitmap scheme VAL-WAH (Variable-Aligned Length WAH) [5]. Note that, VAL-WAH supports intersection (AND) and union (OR) directly on compressed bitmaps without decompression.

3.2 Search Algorithm

3.2.1 Entity-Relationship Graph (ERG)

. We follow the definition in [12] to define entity nodes in XML. A node is an entity node if it has siblings of the same name [2]. For example, the player nodes in Figure 1 are entity nodes. All the entity nodes are shown in dotted rectangles in Figure 1. SpiderX only labels entity nodes with JDewey label [2], which will be discussed later.

DEFINITION 1 (MATCHING ENTITY). *Given a keyword w, an entity node u is a matching entity of w if and only if (i) one of the descendants of u matches w, and (ii) u is not an ancestor of any matching entities of w.*

That is, the matching entity is the lowest ancestor entity node of the node matching the keyword w. For example, the matching entity of keyword "1960" is the entity node "team" in Figure 1.

Given a keyword query q, an *entity-relationship graph (ERG)* of q is a Steiner Tree [6] containing all the matching entities as leaf-nodes. SpiderX always return ERGs as results for given queries, which ensures the meaningfulness of the results.

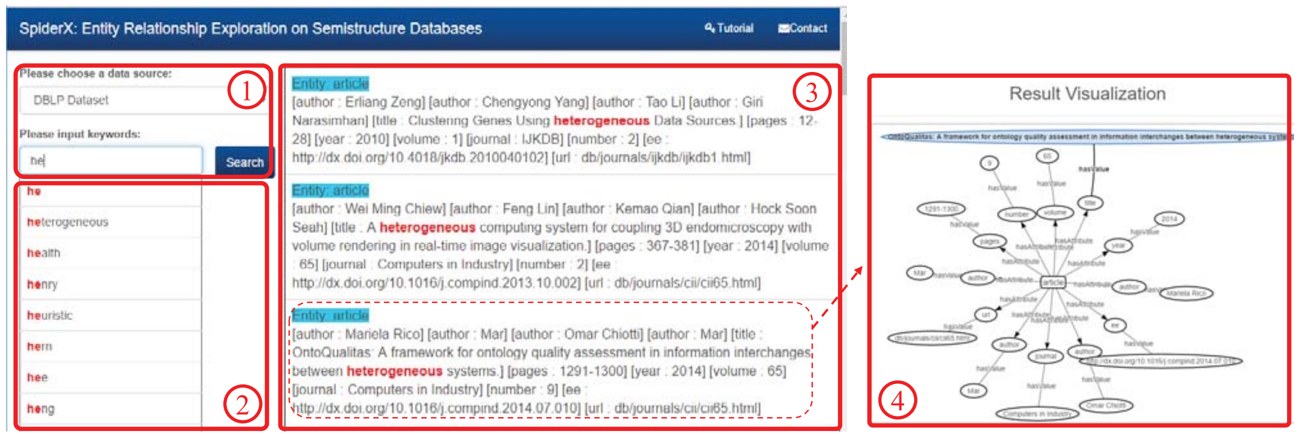
In order to construct ERGs efficiently, for any two keywords w_i and w_j, the *lowest common ancestor entity (LCAE)* should be obtained efficiently. Given two matching entity nodes u_i and u_j, an entity node u is the lowest common ancestor entity (LCAE) of u_i and u_j if and only if (i) u_i and u_j are descendants of u, and (ii) u is not an ancestor of any LCAE of u_i and u_j. For example, the LCAE of two entity nodes "player (0.0.0)" and "player (0.0.1)" is team "(0.0)" in Figure 1.

3.2.2 Search Algorithm.

Let j_u and j_v be the JDewey label of entity nodes u and v respectively. The longest common prefix of j_u and j_v is the LCAE of u and v. The challenge is to compute the longest common prefix efficiently. Let $j_u(i)$ be the i-th JDewey number in the whole JDewey lable j_u. For example, assume $j_u = 0.1.5$, then $j_u(1) = 0$, $j_u(2) = 1$ and $j_u(3) = 5$. Let $j_u[i:j]$ be the JDewey numbers of $j_u(i)...j_u(j)$. Algorithm 1 shows the SpiderX search algorithm. It first computes the integer i by performing intersection over the i-th column in the inverted lists (lines 2-5). Then it returns LCAEs (lines 6-8).

To evaluate the performance of SpiderX search algorithm, we compared it with the Stack-based algorithm [12] and the Index-based algorithm [14] on DBLP dataset [3]. We vary the number of keywords from 2 to 5 while fixing the largest size of inverted list to be 10K and 1M respectively. As shown in Figure 6, SpiderX Join algorithm is around 100 times faster than Stack-based and Index-based methods. The improvement of the performance comes from the following aspects: (1) SpiderX adopts a back-forward access method,

[2]Except the root node.
[3]http://dblp.uni-trier.de/xml/


SpiderX: Entity Relationship Exploration on Semistructure Databases

Tutorial | Contact

Please choose a data source:
DBLP Dataset

Please input keywords:
he

Search

he
heterogeneous
health
henry
heuristic
hern
hee
heng

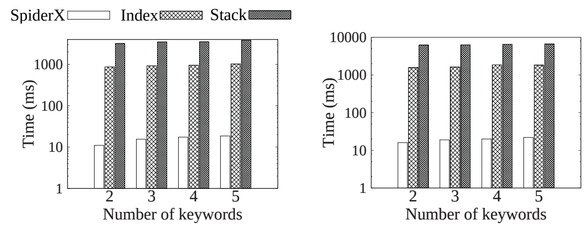
Entity: article
[author : Erliang Zeng] [author : Chengyong Yang] [author : Tao Li] [author : Giri Narasimhan] [title : Clustering Genes Using **heterogeneous** Data Sources] [pages : 12-28] [year : 2010] [volume : 1] [journal : IJKDB] [number : 2] [ee : http://dx.doi.org/10.4018/jkdb.2010040102] [url : db/journals/ijkdb/ijkdb1.html]

Entity: article
[author : Wei Ming Chiew] [author : Feng Lin] [author : Kemao Qian] [author : Hock Soon Seah] [title : A **heterogeneous** computing system for coupling 3D endomicroscopy with volume rendering in real-time image visualization.] [pages : 367-381] [year : 2014] [volume : 65] [journal : Computers in Industry] [number : 2] [ee : http://dx.doi.org/10.1016/j.compind.2013.10.002] [url : db/journals/cii/cii65.html]

Entity: article
[author : Mariela Rico] [author : Mar] [author : Omar Chiotti] [author : Mar] [title : OntoQualitas: A framework for ontology quality assessment in information interchanges between **heterogeneous** systems.] [pages : 1291-1300] [year : 2014] [volume : 65] [journal : Computers in Industry] [number : 9] [ee : http://dx.doi.org/10.1016/j.compind.2014.07.010] [url : db/journals/cii/cii65.html]

Figure 5: A screenshot of SpiderX. ① is the query generator with two components: I. drop-down list choosing an XML data source; and II. keywords input text box. ② is the autocompletion query suggestion list returned by SpiderX. ③ is the results exhibitor. ④ is the visual space explorer with zoom-in/out, drag and rotation features. Demo link: http://chunbinlin.com/spiderx.

```
Algorithm 1: SpiderX Search Algorithm
  Input: $L_{w_1}, ..., L_{w_k}$
  Output: $\mathcal{R}$
1 $\mathcal{R} = \emptyset$;
  // Let $h$ be the height of the XML tree
2 for $i = h \to 1$ do
      // Let $L_{w_i}(j)$ be the j-th column in $L_{w_i}$
3     $\mathcal{I}$=$L_{w_1}(i) \cap ... \cap L_{w_k}(i)$;
4     if $\mathcal{I}! = \emptyset$ then
5         break;
6 for each row id $r \in \mathcal{I}$ do
7     $\mathcal{R} \leftarrow L_{w_1}(r)[0 : i]$;
8 Return $\mathcal{R}$;
```



(a) 10K (b) 1M

Figure 6: Query performance (ms).

which avoids accessing all the labels; and (2) SpiderX uses bitmaps to perform intersection directly on the compressed data without decompressing the data.

4. DEMO SCENARIOS

In this section, we demonstrate different use cases and scenarios in three real-life XML datasets, i.e., DBLP dataset [4], REED dataset [5] and WSU dataset. Figure 5 shows a screenshot of SpiderX.

4.1 Query Autocompletion

Suppose Rachel is an EDBT attendee who wants to find papers about "heterogeneous" in her phone. In other systems, e.g., DBLP search system [6], it is a challenge for her to type the correct characters. Nevertheless, SpiderX provides query autocompletion feature to guide her to complete her query. Once she inputs two letters "he", SpiderX shows "heterogeneous" on the query suggestion list (as shown in Figure 5②). She can click on it, then "heterogeneous" appears on the input-box. By simply clicking on it, the query is complete. In this scenario, she only inputs 2 letters, while in other systems she needs to type 13 characters and may be more if she deletes some wrong letters to refine the query.

4.2 Find Recent Papers

Assume Lucy is a PhD student who is interested in XML query processing. She wants to read the most recent XML papers. She chooses the "DBLP" dataset and issues a query "XML 2016", then SpiderX instantly returns top-30 articles published in 2016 with titles containing the keyword "XML". All the answers are meaningful and complete. In addition, she can visualize the results by a simply clicking. A pop-up window with a visualization graph appears (similar to Figure 5④), which allows zooming in/out and dragging.

4.3 Search for Courses

Suppose Angelina is a graduate student at Reed College. She wants to take an art course starting at 09:00AM. She uses SpiderX to first choose the "REED" dataset then enter "art 09". SpiderX shows four results instantly. Considering both the title and the instructor of the courses, she finally chooses one out of the four results. In addition, she clicks on the selected one to visualize it and saves the visualization result. The visualization result is convenient for her to post on her social networks and share with her friends.

[4]http://dblp.uni-trier.de/xml/

[5]http://www.cs.washington.edu/research/xmldatasets/

[6]http://dblp.org/search

5. REFERENCES

[1] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective xml keyword search with relevance oriented ranking. In *ICDE*, pages 517–528, 2009.

[2] L. J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in xml databases. In *ICDE*, pages 689–700, 2010.

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

[4] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, pages 16–27, 2003.

[5] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A tunable compression framework for bitmap indices. In *ICDE*, pages 484–495, 2014.

[6] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[7] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pages 367–378, 2003.

[8] C. Lin, J. Lu, T. W. Ling, and B. Cautis. Lotusx: a position-aware xml graphical search system with auto-completion. In *ICDE*, pages 1265–1268, 2012.

[9] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer. Fast in-memory SQL analytics on typed graphs. *PVLDB*, 10(3):265–276, 2016.

[10] C. Lin, J. Wang, and Y. Papakonstantinou. Data compression for analytics over large-scale in-memory column databases (summary paper). *CoRR*, abs/1606.09315, 2016.

[11] T. W. Ling, Z. Zeng, T. N. Le, and M. L. Lee. ORA-semantics based keyword search in XML and relational databases. In *ICDEW*, pages 157–160, 2016.

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

[13] Z. Liu, J. Walker, and Y. Chen. XSeek: a semantic XML search engine using keywords. In *VLDB*, pages 1330–1333, 2007.

[14] C. Sun, C.-Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[15] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, pages 527–538, 2005.