LotusX: A Position-Aware XML Graphical Search System with Auto-Completion

Chunbin Lin [1], Jiaheng Lu [1], Tok Wang Ling [2], Bogdan Cautis [3]

[1]*Renmin University of China*
{chunbinlin, jiahenglu}@ruc.edu.cn

[2]*National University of Singapore*
lingtw@comp.nus.edu.sg

[3]*Télécom ParisTech*
cautis@telecom-paristech.fr

***Abstract*— The existing query languages for XML (e.g., XQuery) require professional programming skills to be formulated, however, learning such complex query languages is a tedious and a time consuming process that can be very challenging especially to novice users. In addition, when issuing an XML query, users are required to be familiar with the content (including the structural and textual information) of the hierarchical XML, which is difficult for common users. The need for designing user-friendly interfaces to reduce the burden of query formulation is fundamental to the spreading of XML community.**

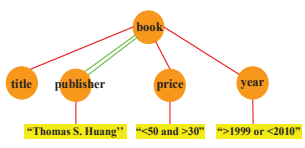
We present a twig-based XML graphical search system, called *LotusX*, that provides a graphical interface to simplify the query processing without the need of learning query languages and data schemas and the knowledge of the content of the XML document. The basic idea is that LotusX proposes "position-aware" and "auto-completion" features to help users to create tree-modeled queries (twig pattern queries) by providing the reasonable candidates on-the-fly. In addition, complex twig queries (including order-sensitive queries) are supported in LotusX. Furthermore, a new ranking strategy and a query rewriting solution are implemented to rank the results and automatically rewrite queries, respectively.

I. Introduction

XML plays an important role in information exchange nowadays. As a result, a wide spectrum of users, including those with minimal or no computer programming skill at all, have the need to query hierarchical XML. Therefore, designing effective and efficient systems to simplify the query processing over XML documents attracts lots of research interests. The well known XML query languages (e.g., XQuery) are provided to process XML queries. However, these languages are far too complicated for unskilled users, who might only be aware of the basics of the XML data model or even lack the knowledge of the content (i.e., structural and textual information) of the XML documents.

```
<bib>
{ for $b in doc ("bib.xml")/bib/book
  where $b//publisher="Thomas S. Huang"
        and ($b/year>1999 or $b/year <2010)
        and ($b/price>30 and $b/price<50 )
return <book> { $b/title } </book> }
</bib>
```

(a) Xquery expression



(b) Twig Pattern Query

Fig. 1. The XQuery and twig pattern expression of the query.

For example, assume that a user wants to issue the query on XML database "List the title of books written by Thomas S. Huang and published before 1999 or later than 2010, and the price should be distributed in 30 ~ 50 dollars". This query can be formulated as the XQuery expression in Figure 1(a). Unfortunately, formulating such query often demands considerable cognitive effort from the end users and requires "programming" skills that is at least comparable to SQL, which can be both time-consuming and error-prone. In order to deal with the problem, XML graphical languages are developed (e.g., XQBE[4], GLASS [7], XQE[3]) to allow the users, who do not know the professional query languages, to express queries. They allow users to create queries through simple graphical languages and then map the queries directly to XQuery in the background. However, (i) users are required to learn the syntax of the graphical languages, furthermore, (ii) users need to have the knowledge of the structural (i.e., the parent-child (P-C) and ancestor-descendant (A-D) relation) and textual (i.e., node names and values) information of the XML documents, since the content of each node in the query should be input by users instead of the systems. E.g., when issuing the query in Figure 1, the user needs to know the name of the publisher is "Thomas" rather than "Thomason" (i.e., textual information) and the *price* is a child of the *book* (i.e., structural information).

In order to simplify the query processing, (i) XML keyword search systems are proposed (e.g., XReal [2]), which return the subtrees containing all the keywords. However, keywords can only express simple textual information but cannot describe the structural information and complex content. For example, these systems cannot answer the query in Figure 1, since keywords cannot describe the structures (e.g., *year* is a child of *book*) and the content conditions (e.g., "year>1999 or year <2010"). (ii) Visual search systems are implemented (e.g., Xing[5]). They present the structural and textual information of the document in visual interfaces, which allows the users to exploit the relationships of the elements and update the values directly. However, they need to load the whole document into memory and cannot answer the complex queries.

Reviewing the existing XML search systems, we can now

derive some design goals for a user-friendly XML query system. First of all, we should not define another textual query language. Second, concrete XML syntax should be avoided. Third, since twig pattern query is a powerful concept that greatly supports the examination of structured data, pattern matching should be employed in the query system. Forth, twig pattern needs to be extended to support complex queries. Finally, all the users including the novice users should be able to issue queries in the system without learning the query languages and the content of the XML documents.

In this demo, we propose a position-aware XML graphical search system with auto-completion (called **LotusX**) that enables users to define accurate XML twig pattern queries without the need of learning professional query languages and the content of highly heterogeneous documents. To our best knowledge, LotusX is the first system that applies position-aware and auto-completion features on XML query processing. LotusX has the following novel features compared to the existing XML search systems.

- It designs a *position-aware* graphical interactive interface to guide users to create twig pattern queries (*Section II-B*).
- It develops *auto-completion* feature based on two kinds of trie indexes to support search-as-you-type for both element tags and values (*Section II-C*).
- It applies a holistic twig pattern algorithm to answer twig pattern queries efficiently. In addition, it supports complex queries, including order-sensitive queries[6] and queries with complex content predicates (*Section II-D*).
- It provides a novel ranking model to rank answers and a *relaxation similarity* to rewrite queries, which have no answer results.(*Section II-E*).

II. Overview Of LotusX

In this section we will give an overview of the architecture of LotusX, followed by multiple unique features in LotusX.

A. Architecture

The system architecture of LotusX is presented in Figure 2. The *Data Parser* parses the input XML data and schemas, and labels the inherent elements, attributes and values. Here we use region labeling, i.e., (*start: end, level*), to present the position of a tree node in the data tree. The *Index Builder* constructs inverted index for efficiently answering the queries and two kinds of tries for providing *auto-completion* feature. The *Twig Pattern Generator* provides an interface for users to generate XML twig patterns graphically. During the generation of twig queries, the client issues AJAX requests on-the-fly to the server. Then the Position-Aware Manager searches the candidates which satisfy the position information, and the *Auto-Completion Manager* supports search-as-you-type by searching in the trie indexes to return the instant feedback. When users submit queries (including complex queries), the *Twig Search Operator* employs a twig pattern algorithm to get the answers. Then *Result Generator* ranks the results according to their relevance and popularity, then demonstrates the results graphically as a form of trees. Finally, if there is no answer to match user's query, the *Query Rewriter* rewrites the original twig to generate new query-candidates to help users easily reformulate queries.

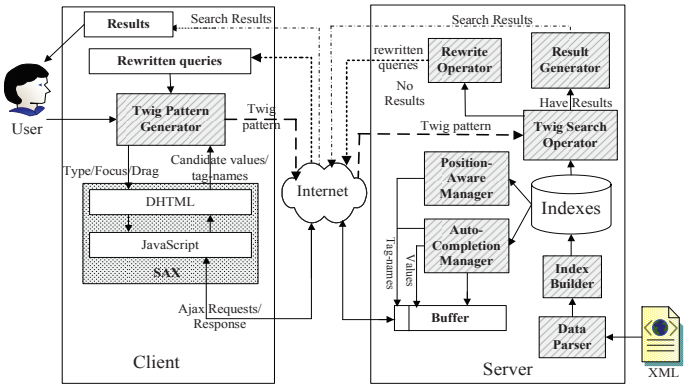


Fig. 2. Architecture of LotusX

B. Position-Aware Interactive Interface

A query is modeled as a small tree (i.e., twig pattern) in LotusX, since tree-structure can well express both the structural and textual information. The twig pattern node labels include element tags, attribute-values and string-values, and the query twig pattern edges are either parent-child edges (depicted using a single red line) or ancestor-descendant edges (depicted using a double green line). See an example twig pattern in Figure 1(b), which is equal to the XQuery expression in Figure 1(a). LotusX allows novice users to generate a meaningful twig pattern through a graphically interactive interface, which can provide reasonable candidate tag-names for the new created node in different positions. See the query in Figure 1(b), when a new node is presented below the node "book", then the system automatically shows the candidate tag-names, e.g., "publisher", "year". Note that the candidates must be children

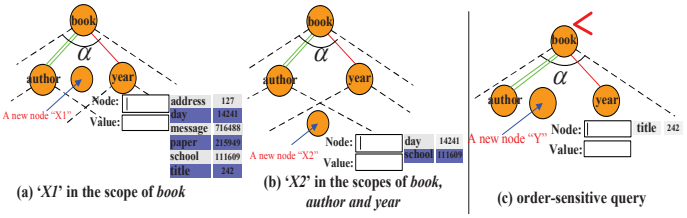


Fig. 3. Position-aware feature. In (a), the number of the candidate tag-names for the new node X_1 is greater than that of X_2 in (b), since node X_2 is affected by three nodes while X_1 is only affected by one node. In (c) the candidate tag-name is "title", since *author* $\ll$ *title* $\ll$ *year*.

or descendants of "book" in the XML data. Otherwise, the query is meaningless. In order to specify such "position-relationship", we define the sector of α degree below each node as its scope (see the doted lines in Figure 3(a)). If a node A is in the scope of node $B \in q$, then we say A is affected by B, i.e., the candidates of A should be all the potential descendant tag-names of node B. In practice, a new node would be affected by multiple nodes in the query, and the nodes distribute in different levels, since the query is a twig. Therefore, we carefully design the strategy to generate reasonable candidates satisfying the above conditions. Assume a new node X is in the scope of node $n_{ij} \in q$, which is the

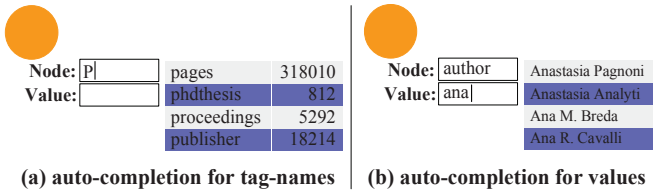
j'th node from left to right in the i'th level of q, then the candidates of X can be computed as follows:

$$\bigcap_i \{\bigcup_j Desc(n_{ij})\}$$

where $Desc(n)$ is a set containing all the children and descendant tag-names of node n. To better understand this, let us see examples in Figure 3(a) and (b). In Figure 3 (a), the candidate tag-names of X_1 are listed nearby the input-box and each tuple in the list composed of a tag-name and the total number of the occurrences of the tag-name in XML data set. The candidates are the children or descendants of *book*, since X_1 is in the scope of *book*. However, in Figure 3(b), X_2 is in the scope of three nodes (i.e., *book*, *author* and *year*) and *book* has higher level than *author* and *year*. Thus the candidates of X_2 is a set calculated by $Desc(book) \cap (Desc(author) \cup Desc(year))$.

C. Search-as-you-type with Auto-completion

In information retrieval, query auto-completion is proposed to support search-as-you-type. As we discussed above, LotusX suggests the reasonable candidates to users, however, the number of candidates might be large and it is not elegant to list all the candidates to users. So we provide auto-completion to support search-as-you-type for both tag-names and values from the candidates, which returns candidates on-the-fly as a user types letter by letter and gives the user instant feedback. Note the fact that tag-names and values have different data characteristics, i.e., the number of tag-names is limited, while the values are usually too large to be completely loaded into the memory. Therefore, we design two trie indexes for them respectively, they are **Static Tag-trie** for small tag data and **Dynamic Value-trie** for large value data. (i) We build a static



(a) auto-completion for tag-names **(b) auto-completion for values**

Fig. 4. The auto-completion for tag-names and values. In (a),the candidate tag-names starting with p are listed, and in (b), the candidate values of node "author" and starting with *ana* are chosen.

tag-trie tree for all the tag-names, which is not large and can be kept in the memory. Each tag-name in the tree corresponds to a unique path from the root to a leaf node. We first find the corresponding trie node of the prefix, which is input by users, and then traverse the subtree to get the values with a prefix of the input message. E.g., in Figure 4(a), user input a letter "p", then the returned candidate tag-names are all starting with "p". (ii) Due to the large size of the values and the values are related to different tag-names, e.g., "Anastasia Pagnoni" in Figure 4(b) relates to the "author" rather than other tag-names. Thus, we display a representative subset of values for each tag-name, which can be loaded in the memory, to construct a dynamic value-trie. After selecting the tag-name, the user focuses on the value input-box. At this time, the client sends the tag-name to the value-trie to locate the subtree rooted by the tag-name. In Figure 4(b), the value-trie locates the subtree of "author". Once the users type any new letter in the value inputbox, value-trie returns the values belonging to the tag-name and starting with the letters by searching the corresponding path in the subtree. However, if there is no such value satisfying the requirement, we read the corresponding disk-resident file through the indexes and display in a recursive way, and each time we only read the representative subset of the rest data to build a value-trie.

D. Order-sensitive Queries

To capture the semantics of XPath expressions with order axes, such as the following-sibling and preceding-sibling, we extend the common twig pattern by adding order constraint, in addition to P-C and A-D edges. Our paper [6] has the detailed discussion about the meaning of the symbol "<" (see Figure 3(c)) and how to answer order-sensitive queries. In this paper, we focus on how to automatically generate an order-sensitive query. Given two node types A and B in an XML database D, we say that type A precedes B written A≪B, if there are two instance nodes a and b such that a precedes b in the depth-first traverse of D. Those order information about node types can be obtained in *Index Builder* in pre-processing. See the example in Figure 3 (b) and (c). Since (b) does not have the order constraint, the number of the candidate tag-names for X_2 in (b) is significantly larger than that for Y in (c).

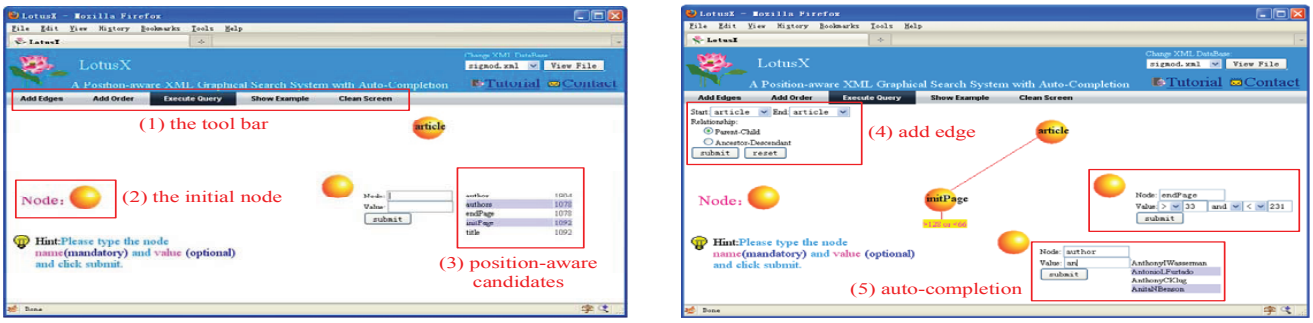
E. Results Ranking and Query Rewriting

We propose a ranking model to rank the results according to their importance and popularity. In addition, if there is no result based on users' initial query, we invoke a rewriting mechanism to provide refined queries. To rank a result r for a query q, we need to consider the structural and content factors to compute an overall relevance score. The followings are important factors: (1) weights of different tags, (2) length of each path, (3) number of irrelative nodes. The following is a scoring function that reflects the above three factors. The score of a result r for a query q is defined as:

$$S_T(r,q) = \sum_{(pq,pr)\in P} \left\{ \frac{1+|pq|}{1+|pr|} * \sum_{t_{pq}\in pq} \frac{wf - idf(t_{pq},r)}{wf - idf(t_{pq},q)} \right\} - \frac{\sum_{t_T\in T} wf - idf(t_T,r)}{\sqrt{\sum_{t_r\in r} wf - idf(t_r,r)^2}} \quad (1)$$

where *wf-idf* is similar to "XML tf-idf" in our paper [1], and $\{pq, pr\}$ is a mapping pair of paths from the query q and result r respectively. Let T be the set of node types only in result r but not in query q, and t_{pq} , t_r and t_T are node types in path pq, result r and set T respectively. To understand the S_T score, the first multiplier(i.e.$(1+|pq|)/(1+|pr|)$) actually addresses Factor 2, while the second multiplier addresses Factor 1 by computing the weights of the nodes. Finally, the third component reduces the score of the irrelative node types to address Factor 3.

In order to automatically rewrite queries, which have no results, our main strategy is to relax the conditions of the original query from the aspects of structural and/or value conditions. We define four basic relaxation operations from the lowest penalty to the highest, as follows: (i) remove the "order" constraint; (ii) P-C edge to A-D edges; (iii) remove the



(a) position-aware feature (b) auto-completion feature

Fig. 5. A snapshot of the demonstration of LotusX.

value constraint; (iv) prune the node with the smallest weight. Note that more relaxation operations result in less similarity with the original query.

Given an original query Q and a relaxation query Q', we define the Relaxation Similarity (RS) to measure the similarity between Q and Q'.

$$RS(Q, Q') = \frac{1 + \lg |Q'(D)|}{\sum_{|RT(Q,Q')|}^{i=1} 2^{w(i)}} \quad (2)$$

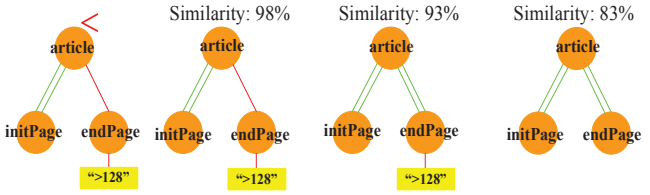
where $|Q'(D)|$ denotes the number of answers to Q' in document D; $|RT(Q, Q')|$ represents the times of employing the four basic operations to transform Q into Q'; $w(i)(> 1)$ is the penalty of the relaxation operation i. In this way, a relaxation query Q' is ranked higher if *RS(Q,Q')* is greater, meaning that Q' returns more answers and is more similar to Q.

III. DEMONSTRATION SCENARIOS

A prototype system (LotusX) has been implemented with a web-based interface, which is running on Ubuntu 9.10. In this demo, three XML databases are utilized, i.e., the DBLP(130MB), the airline(50M), and the Sigmod(30M).

Scenario 1: Demonstrate the position-aware and auto-completion features. The first scenario is to demonstrate how to generate an XML twig query conveniently with the aids of position-aware and auto-completion features. Assume that a user wants to create an XML query in the interface. (S)he can create a node by dragging the initial node (see (2) in the Figure 5(a)). Then the system returns the candidates instantly by the position-aware feature, as shown in (3) in the Figure 5(a). After the nodes are created, then (s)he can generate edges by clicking the button in the tool bar (see (4) in the Figure 5(b)) and then either P-C or A-D edges are created. Note that, the auto-completion feature (i)identifies the type of the tag-name, e.g., in (5) in the Figure 5(b), the value of *endPage* is number and that of *author* is string; (ii) supports search-as-you-type, e.g., the user input "*an*" and the system returns the corresponding values starting with the prefix.

Scenario 2: Generate a complex twig pattern query. This scenario demonstrates how to create a complex twig pattern query with "order-constraint" and content condition. Assume the user wants to issue the query "finding the initPage of articles whose endPage is greater than 128 and initPage should be the preceding of endPage", which is equivalent to an XPath "//article/endPage>'128'[preceding-sibling: initPage]". The user can create the query in the interface as we introduced in Scenario 1. In addition, (s)he can add "order" symbol to the query by simply clicking the "Add Order" button (see (1) in the Figure 5(a)). Thus, a complex twig pattern is successfully created, which is shown in Figure 6(a).



(a) Query Q (b)Rewritten Query Q1 (c)Rewritten Query Q2 (d)Rewritten Query Q3

Fig. 6. Rewritten queries of query Q, Q_1 has the highest similarity, i.e., 98%, while Q_2 and Q_3 have less similarities (i.e., 93% and 83%, respectively).

Scenario 3: Query executing and auto-rewriting features. This scenario is employed to demonstrate the power of query executing and query rewriting features. We show that the execution of the query in Figure 6(a) does not return any result, since in the document *initPage* is not in preceding of *endPage*. But LotusX can automatically rewrite the query by applying different relaxation operations. For example, Q_1 in Figure 6 (b) is rewritten by removing the "order" condition, which has the highest similarity and has meaningful results. In addition, Figure 6 (c) and (d) present the rewritten queries by changing edges and removing node values, respectively. Note that, all the three rewritten queries (i.e., Q1,Q2,Q3) have meaningful results. Finally, we return the results for the refined query (which is selected by users) which are sorted by the ranking scores computed by Formula 1.

REFERENCES

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

[2] Z. Bao, J. Lu, and T. W. Ling. Xreal: an interactive xml keyword searching. In *CIKM*, pages 1933–1934, 2010.

[3] V. Borkar, M. Carey, S. Koleth, A. Kotopoulis, K. Mehta, J. Spiegel, S. Thatte, and T. Westmann. Graphical XQuery in the aqualogic data services platform. In *Proc. SIGMOD*, pages 1069–1080, 2010.

[4] D. Braga and A. Campi. A graphical environment to query xml data with xquery. In *WISE*, pages 31–40, 2003.

[5] M. Erwig. Xing: a visual xml query language. *J. Vis. Lang. Comput.*, 14(1):5–45, 2003.

[6] J. Lu, T. W. Ling, Z. Bao, and C. Wang. Extended xml tree pattern matching: Theories and algorithms. *IEEE Trans. Knowl. Data Eng.*, 23(3):402–416, 2011.

[7] W. Ni and T. Ling. Translate graphical XML query language to SQLX. In *Proc. DASFAA*, pages 907–913, 2005.