Challenges, Techniques and Directions in Building XSeek: an XML Search Engine

Ziyang Liu Peng Sun Yu Huang Yichuan Cai Yi Chen
Arizona State University
{ziyang.liu, peng.sun, yu.huang.1, yichuan.cai, yi}@asu.edu

Abstract

The importance of supporting keyword searches on XML data has been widely recognized. Different from structured queries, keyword searches are inherently ambiguous due to the inability/unwillingness of users to specify pinpoint semantics. As a result, processing keyword searches involves many unique challenges. In this paper we discuss the motivation, desiderata and challenges in supporting keyword searches on XML data. Then we present an XML keyword search engine, XSeek, which addresses the challenges in several aspects: identifying explicit relevant nodes, identifying implicit relevant nodes, and generating result snippets. At last we discuss the remaining issues and future research directions.

1 Introduction

Information search is an indispensable component of our lives. Due to the vast collections of XML data on the web and in enterprises, providing users with easy access to XML data is highly desirable. The classical way of accessing XML data is through issuing structured queries, such as XPath/XQuery. However, in many applications it is inconvenient or impossible for users to learn these query languages. Besides, the requirement that the user needs to comprehend data schemas may be overwhelming or infeasible, as the schemas are often complex, fast-evolving or unavailable. A natural question to ask is whether we can empower users to effectively access XML data simply using keyword queries.

Unlike text document search where the retrieval unit is an entire document, keyword search on XML data has every XML node as a retrievable unit, thus has a significant potential for fine-grained and high-quality results. Yet it poses a lot of unique challenges of inferring relevant fragments in XML data, composing query results, relevance based ranking, and result presentation. To gracefully process keyword searches on XML, an XML search engine should ideally satisfy a set of desiderata, varying from generating meaningful results to helping users quickly select relevant results. The desiderata include but are not limited to the following ones.

Identifying Explicit Relevant Nodes. A user can specify the required information explicitly using keywords. However, not all nodes matching keywords are necessarily relevant to the query, which need to be distinguished by a search engine. Consider query "*Galleria, Houston*" on Figure 1. For keyword "*Houston*", the match associated with *city* (6) is relevant, as it belongs to a store whose name is Galleria. The *Houston* node associated

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

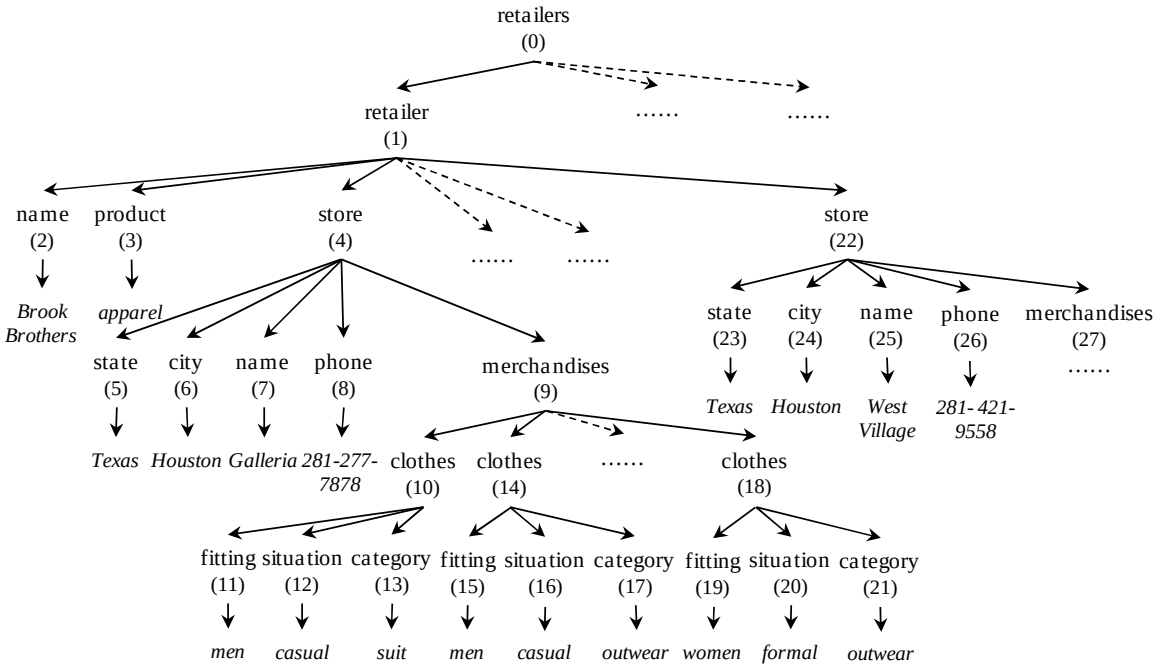


Figure 1: Sample XML Data Fragment

with *city* (24), on the other hand, is irrelevant.

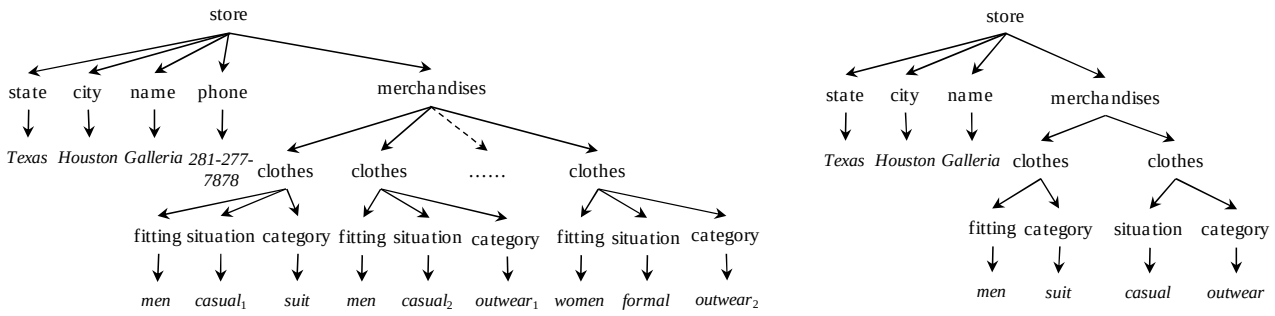
Identifying Implicit Relevant Nodes. Besides the keyword matches that can be relevant, other XML nodes, although do not match keywords, can also be interesting to the user, which are referred to as *implicit relevant nodes*. For the sample query "*Galleria, Houston*", the user is likely interested in the general information of *Galleria* which is located in *Houston*. Therefore, besides returning the keyword matches and their connections, other information of the store, such as its phone number, is also relevant. A result of the sample query is shown as the subtree in Figure 2(a).

Composing Meaningful Results. Given explicit and implicit relevant nodes, the next step is to compose query results using these nodes. Different from text search where a retrieval unit is a document, a result in XML search is a subtree. Given a set of relevant nodes, there can be many ways to separate them into different results, and a meaningful way to do so is desired.

Result Ranking. A keyword search engine usually returns many results, which are not equally relevant to the user. Therefore, it is highly important to rank the results so that the ones that are likely more relevant to the user will be returned earlier.

Generating Result Snippets. While trying to measure the relevance and rank the results, a ranking function is impossible to always correspond to users' preferences. To compensate, result snippets are used by almost every text search engine. The principle of result snippets is to let users quickly judge the relevance of query results by providing a brief quotable passage of each query result, so that users can easily choose and explore relevant ones among many results. For instance, Figure 2(b) is a snippet for the result in Figure 2(a), which highlights the most important information in the result.

In the remaining of this paper we mainly discuss how to achieve three of the desiderata introduced above in building an XML keyword search engine, XSeek [6, 8, 9, 10]: identifying explicit relevant nodes (Section 3.1), identifying implicit relevant nodes (Section 3.2), and generating snippets for query results (Section 3.3). Future directions are discussed in Section 4.



(a) Result Fragments

(b) Snippet of (a)

Figure 2: Result Fragments of Query "*Galleria, Houston*" and its Snippet

2 The *XSeek* System: Architecture

The system architecture of XSeek is presented in Figure 3, which is composed of three main modules: *Index Builder*, *Result Generator* and *Result Presenter*. As their names indicate, *Index Builder* builds data indexes which speed up query processing; *Result Generator* takes query keywords as input and generates results; *Result Presenter* presents the results to the user in such a way that the users can quickly find relevant results. The modules are described in detail in the following.

Index Builder. The *Index Builder* parses the input XML data, infers the inherent entities and attributes (to be explained in Section 3) in the data, and builds indexes for retrieving the information about node category, parent and children.

Result Generator. Once a user issues a query, *Explicit Relevant Node Identifier* accesses the indexes, retrieves matches to each keyword, and identifies the matches that are relevant to the query. Then, *Implicit Relevant Node Identifier* identifies other relevant nodes besides keyword matches. Specifically, *Keyword Categorizer* analyzes the match patterns and categorizes input keywords as predicate nodes and return nodes. Based on the analysis of keyword match patterns and entities in the data, *Return Node Identifier* recognizes return nodes for the query. *Query Result Composer* generates query results based on the relevant nodes identified in the previous steps.

Result Presenter. *Query Result Ranker* ranks the results generated by *Query Result Composer* based on diverse ranking factors, such as variants of Term Frequency (TF), Inverse Document Frequency (IDF), PageRank, the number of keyword matches, proximity of keyword matches, query result size, etc.. However, instead of having all the results generated and then ranked, *Query Result Composer* and *Query Result Ranker* work interactively in order to generate query results in an (approximate) ranked order, so that the top-ranked results can be returned in a short time. *Result Snippet Generator* provides a self-contained, distinguishable, representative and concise summary for each query result to help users quickly judge the relevance of results.

3 Important Modules in XSeek

In this section, we discuss XSeek modules that address several challenges of XML search. We model XML data as a rooted, labeled and unordered tree, such as the one shown in Figure 1. Every internal node in the tree has a node name, and every leaf node has a data value. A keyword search is a query consisting of a set of keywords, and a result of a keyword search is a subtree which consists of selected nodes in the XML tree along with their connecting paths. XSeek adopts AND semantics, i.e., each result contains all keywords in the query.

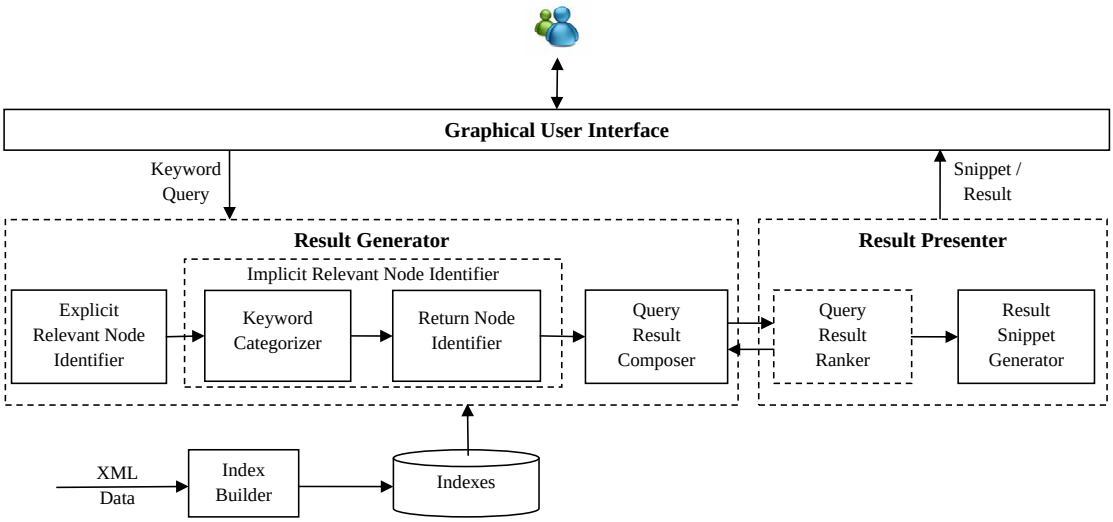


Figure 3: Architecture of XSeek

3.1 Explicit Relevant Node Identifier

As discussed in Section 1, explicit relevant nodes are matches to query keywords which are deemed relevant. To identify explicit relevant nodes, XSeek begins with computing the *smallest lowest common ancestor* (SLCA) nodes [12] of a keyword search. The SLCAs of a query Q consist of nodes which contain all keywords in their subtrees, and none of their descendants contains all keywords in its subtree. Computing SLCA helps prune some irrelevant matches. For example, the only SLCA node of query "*Galleria, Houston*" on Figure 1 is node *store* (4), thus the *Houston* node associated with *city* (24) is pruned. However, not all keyword matches in the subtrees of SLCAs are relevant, e.g., for query "*Brook Brothers, Galleria, Houston*", the SLCA is *retailer* (1), however, the *Houston* node associated with *city* (24) is still irrelevant.

XSeek further prunes irrelevant matches by pruning inferior subtrees. A subtree is considered inferior and is pruned, if the root of the subtree has a sibling whose subtree contains strictly more keywords. Consider query "*Brook Brothers, Galleria, Houston*" again. For node *store* (4), the set of keywords contained in its subtree is $\{$*Galleria, Houston*$\}$. Assume that there is no *Galleria* node in the subtree rooted at *store* (22), i.e., it only contains keywords $\{$*Houston*$\}$. As a result, node *store* (22) is pruned along with its subtree, thus the *Houston* node in this subtree is not considered as an explicit relevant node.

There are many other strategies for identifying explicit relevant nodes, and a question is how to evaluate their effectiveness. The effectiveness of a keyword search approach is usually evaluated based on the ground truth obtained from user study. We proposed a cost-effective evaluation alternative [9], which uses an axiomatic framework consisting of the following four desirable properties for identifying explicit relevant nodes: data/query monotonicity/consistency. These properties capture the reasonable behaviors of an XML search engine adopting AND semantics upon a change to the data or the query.

Data Monotonicity. If we add a new node to the data, then the data content becomes richer, therefore the number of query results should be (non-strictly) monotonically increasing. Analogous to keyword search on text documents, adding a word to a document that is not originally a query result may qualify the document as a new query result.

Query Monotonicity. If we add a keyword to the query, then the query becomes more restrictive, therefore the number of query results should be (non-strictly) monotonically decreasing. Analogous to keyword search on text documents, adding a keyword to the query can disqualify a document that is originally a query result to be a result of the new query.

Data Consistency. After a data insertion, each additional subtree that becomes (part of) a query result should contain the newly inserted node. Analogous to keyword search on text documents, after we add a new word to the data, if there is a document that becomes a new query result, then this document must contain the newly inserted word.

Query Consistency. If we add a new keyword to the query, then each additional subtree that becomes (part of) a query result should contain at least one match to this query keyword. Analogous to keyword search on text documents, after we add a new keyword to the query, if a document remains to be a query result, then it must contain a match to the new keyword.

As an example of data monotonicity, consider query "*Galleria, Houston*" issued on Figure 1. Suppose a search engine outputs a set of results, each being a subtree rooted at a *store* node that contains *Galleria* and *Houston* as descendants. Now suppose that we add one more *store* node to the data that has *Galleria* and *Houston* as descendants. If the same search engine outputs fewer results after the data insertion, then it is counter-intuitive, and violates data monotonicity.

As an example of query consistency, consider query "*Galleria, Houston*" again. For this query, suppose a search engine outputs a set of results, each rooted at a *store* node that has *Galleria* and *Houston* in its subtrees; the *Houston* node associated with *city* (24) is considered as irrelevant. Now if we add a new query keyword *Brook Brothers*, and the query becomes "*Brook Brothers, Galleria, Houston*". If the same XML search engine considers the *Houston* node associated with *city* (24) as relevant for the second query, it violates query consistency. Indeed, this is unlikely to be desirable.

Although these properties appear to be intuitive and simple, it is not trivial to satisfy them. It was shown that XSeek is the only known approach that satisfies all four properties [9].

3.2 Implicit Relevant Node Identifier

Besides relevant keyword matches identified by the *Explicit Relevant Node Identifier*, other XML nodes that do not match keywords may also be relevant, which are referred to as *implicit relevant nodes*, and are identified by the *Implicit Relevant Node Identifier* module.

Consider two queries Q_1: "*Galleria, Houston*", and Q_2: "*Galleria, city*". By issuing Q_1, the user is likely interested in information about a store whose name is Galleria and which is located in Houston. In contrast, Q_2 indicates that the user is likely interested in a particular piece of information: the city of Galleria. Note that if we only consider explicit relevant nodes, then these two queries have similar results. However, they have different semantics and their results should be different. This can be addressed by considering implicit relevant nodes. The implicit relevant nodes for Q_1, intuitively, include those in the subtree rooted at *store* (4), while the implicit relevant nodes for Q_2 include those in the subtree rooted at *city* (6).

To infer implicit relevant nodes, we can obtain some hints by analyzing keyword categories. From these two queries, we can see that an input keyword can specify a *predicate* that restricts the search (analogous to "where" clause in XQuery), such as *Galleria* and *Houston* in Q_1. Or, a keyword may specify a desired return node (analogous to "return" clause in XQuery), such as *city* in Q_2. The subtree rooted at a return node, albeit not necessarily on the paths connecting relevant keyword matches, is usually interesting to the user. For Q_2, for instance, the return node is *city* (6), and therefore, its child node *Houston* is an implicit relevant node.

However, return nodes are not always specified in a keyword query, such as Q_1 in which both keywords specify predicates. In this case, we believe that the user is interested in the general information about the data entities related to the search, and consider such entities as return nodes. For instance, in Q_1 the user is likely to be interested in the general information about entity *store* (4), and thus *store* is considered as the return node for this query.

XSeek uses the following heuristics to infer keyword match patterns and data semantics to infer return nodes.

Analyzing Keyword Match Pattern. The *Keyword Categorizer* module of XSeek infers keyword categories using the following rules:

1. If a keyword k_1 has a match u which is a name node (i.e., element or attribute name), and there does not exist an input keyword k_2 with match v, such that u is an ancestor of v, then we consider k_1 as a return node.

2. A keyword that is not a return node is treated as a predicate.

Indeed, if a keyword matches a name node but none of its descendants are specified in the query, then very likely the user is interested in the detailed information of this node, which appears in its subtree. For example, keyword *city* in Q_2 is a return node as it matches name node (6) with no other keyword matching its child, *Houston*. This indicates that the user is likely interested in the information of *city*, which is *Houston*. In Q_1, *Houston* is considered as a predicate since its match is a value node.

Analyzing Data Semantics. In the case that return nodes are not specified in the keywords, XSeek infers return nodes by analyzing XML data semantics. Specifically, the *Return Node Identifier* module categorizes XML data nodes into three categories: entity, attribute and connection node, using the following rules:

1. A node represents an *entity* if it corresponds to a *-node in the DTD.

2. A node, together with its child, denotes an *attribute* if it does not correspond to a *-node, and only has one leaf child.

3. A node is a *connection node* if it represents neither an entity nor an attribute.

When there is no return node specified in the query, XSeek identifies return nodes as the entity nodes that are in the subtrees rooted at SLCAs.

Besides outputting the paths connecting each SLCA and relevant keyword matches, XSeek also outputs the subtree rooted at each return node to make the result meaningful. The result of Q_1 "*Galleria, Houston*", in which both keywords are inferred as predicates and entities *store* and *clothes* are considered as return nodes, is presented in Figure 2(a).

3.3 Result Snippet Generator

XSeek is the first XML search engine which generates snippets for query results. Compared with result snippets for text document search, XML presents more opportunities for generating meaningful result snippets due to its semi-structured nature with mark-ups providing meaningful annotations to data content.

Continue with the sample query "*Galleria, Houston*" and the result shown in Figure 2(a). To generate result snippets, XSeek selects *informative items* from the result into *a snippet information list*, which is initialized with the list of keywords, such as the items in step 1 in Figure 4. Then, it selects instances of the items in the list from a result to construct a snippet. The items in the information list are selected from the result based on the following goals: self-contained, distinguishable, representative, and small.

Self-Contained. A result snippet should be *self-contained* so that the users can understand it. In text search, result snippets usually consist of one or more "windows" on the document containing the complete phrases/sentences where the keyword matches appear, which are self-contained and can be easily read. In contrast, in XML documents, each entity (Section 3.2) is a semantic unit. XSeek includes the names of the entities involved in the query result as context information, even though such entity names may not necessarily appear between two keyword matches in the XML document. For query "*Galleria, Houston*", entities *store* and *clothes* are included in the snippet information list, as shown in step 2 in Figure 4.

Distinguishable. Different result snippets should be *distinguishable* so that the users can differentiate the results from their snippets with little effort. To achieve this in text document search, result snippets often include the document titles. In XML keyword search, intuitively, the key of a result can be used for distinguishing results. A query result may contain several entities, thus the question is the keys of which entities should be considered as the key of the query result. XSeek classifies the entities in a query result into two categories.

1. return entities are what the users are looking for when issuing the query.

2. supporting entities are used to describe the return entities in a query result.

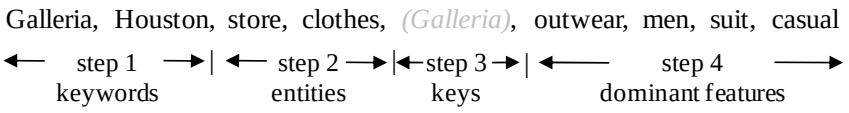


Figure 4: Snippet Information List

Since return entities reflect the search goal, XSeek considers the keys of return entities as the keys of a result. XSeek adopts the following heuristics for identifying return entities: an entity in a query result is considered a *return entity* if it is associated with a return node (i.e., it is itself a return node, or it has an attribute which is a return node). If there are multiple such entities, then the ones with biggest heights are considered as the return entities.

The key of XML nodes can be directly obtained from the schema of XML data (if available), specified as ID or Key node. Otherwise, XSeek finds the most selective attribute of the return entity and uses it as the key. For query "*Galleria, Houston*", the return node is *store*, therefore *store* is considered as the return entity. Its key, *Galleria*, is included in the snippet information list (step 3 in Figure 4).[1]

Representative. A snippet should be *representative* to the query result, thus the users can grasp the essence of the result from its snippet. Intuitively, the most prominent features in the result should be shown in its corresponding snippet. A feature is defined as a triplet (entity name e, attribute name a, attribute value v). For example *clothes:fitting:men* is a feature. The pair (entity name e, attribute name a) is referred as the type of a feature, and attribute value v is referred as the value of a feature.

Intuitively, a feature is considered *dominant* if it has a large number of occurrences compared with features of the same type, quantified by its dominance score. For example, if most clothes in Figure 2(a) are for *men*, then feature *clothes:fitting:men* is a dominant feature. Features are added to the snippet information list in the order of their dominance score, as shown in the last step of Figure 4.

Small. A result snippet should be *small* so that the users can quickly browse several snippets. XSeek generates snippets subject to an upper bound of snippet size (in terms of the number of nodes), which can be specified by users.

An item in the snippet information list can have multiple occurrences in the query result. Although different instances of the same informative item are not distinguishable in terms of semantics, they have different impacts on the size of the snippet. For example, in Figure 2(a), to include two items *casual* and *outwear*, choosing $casual_2$ and $outwear_1$ results in a smaller tree than choosing $casual_1$ and $outwear_1$.

The problem of maximizing the number of informative items selected in a snippet given the snippet size upper bound is hard. The decision version of this problem is proved to be NP-complete [6]. XSeek uses a greedy algorithm that can efficiently select instances of informative items in generating a meaningful and informative snippet for each query result given an upper bound on size. The snippet of Figure 2(a) is shown in Figure 2(b).

4 Conclusions and Future Research Directions

We have introduced the desiderata and challenges in developing XML keyword search engines, and presented how to address some of them in XSeek. In this section we summarize the remaining issues in building a full-fledged XML search engine and future research directions.

Ranking and Top-k Query Processing. Due to the inherent ambiguity of keyword searches, ranking is highly important for any keyword search engine. A ranking scheme considers both ranking factors for individual nodes, such as variants of TF, IDF, PageRank, and ranking factors for the entire result, such as the aggregation of individual node scores, keyword proximity and result size [3, 4, 5, 7].

[1] Item *Galleria* is already included in the snippet information list in step 1.

However, none of the existing approaches on XML keyword search supports top-k query processing. They need to generate all the results, compute a score for each result and finally sort them. This can be potentially very inefficient when the number of results is large. It is highly desirable but challenging to develop efficient top-k algorithms for XML keyword search, which, through the interaction of *Query Result Composer* and *Query Result Ranker* modules in Figure 3, generates top-k query results without producing all results.

Utilization of User Relevance Feedbacks. In information retrieval, user relevance feedbacks have been widely exploited for deducing users' interests and have achieved big success in improving search quality. Existing work on utilizing feedbacks for XML search focuses on finding the structural relationships among keywords using explicit feedbacks [11]. How to use relevance feedbacks, especially implicit feedbacks, in different aspects of XML keyword search is largely an open question. With the structure of XML data, there are plenty of opportunities of utilizing user relevance feedbacks to improve search quality, from relevant node inference in result generation to ranking and snippet generation in result presentation in a keyword search engine.

Evaluation of XML Keyword Search Systems. Since there are many approaches of generating and presenting results for XML keyword search, a framework to evaluate them is critical for users to understand the trade-offs of existing systems. The quality of an XML keyword search system is often gauged empirically with respect to the ground truth over a large and comprehensive set of test data and queries. INEX [1] is an initiative for developing empirical benchmark for evaluating XML search. Two important factors are proposed to measure result relevance: exhaustivity, measuring how well a node satisfies user's information need; and specificity, measuring how well a node focuses on the user's information need [2]. A cost-effective alternative to an empirical benchmark is to evaluate XML keyword search systems based on a set of axioms that capture broad intuitions, such as [9] which evaluates strategies for identifying relevant keyword matches using axioms (as reviewed in Section 3.1). We believe it is essential for the community to actively contribute to comprehensive frameworks for XML keyword search evaluation.

5 Acknowledgement

This material is based on work partially supported by NSF CAREER award IIS-0845647 and NSF grant IIS-0740129.

References

[1] INitiative for the Evaluation of XML Retrieval. http://inex.is.informatik.uni-duisburg.de/.

[2] S. Amer-Yahia and M. Lalmas. XML Search: Languages, INEX and Scoring. *SIGMOD Rec.*, 35(4), 2006.

[3] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective XML Keyword Search with Relevance Oriented Ranking. In *ICDE*, 2009.

[4] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML, 2003.

[5] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: Ranked Keyword Searches on Graphs. In *SIGMOD*, 2007.

[6] Y. Huang, Z. Liu, and Y. Chen. Query Biased Snippet Generation in XML Search. In *SIGMOD*, 2008.

[7] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: Efficient and Adaptive Keyword Search on Unstructured, Semi-structured and Structured Data. In *SIGMOD*, 2008.

[8] Z. Liu and Y. Chen. Identifying Meaningful Return Information for XML Keyword Search. In *SIGMOD*, 2007.

[9] Z. Liu and Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. In *VLDB*, 2008.

[10] Z. Liu and Y. Chen. Answering Keyword Queries on XML Using Materialized Views. In *ICDE*, 2008 (Poster).

[11] R. Schenkel and M. Theobald. Structural Feedback for Keyword-Based XML Retrieval. In *ECIR*, 2006.

[12] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*, 2005.