

# Identifying Meaningful Return Information for XML Keyword Search

Ziyang Liu  
Arizona State University  
Ziyang.Liu@asu.edu

Yi Chen  
Arizona State University  
Yi@asu.edu

## ABSTRACT

Keyword search enables web users to easily access XML data without the need to learn a structured query language and to study possibly complex data schemas. Existing work has addressed the problem of selecting qualified data nodes that match keywords and connecting them in a meaningful way, in the spirit of inferring a *where clause* in XQuery. However, how to infer the *return clause* for keyword search is an open problem.

To address this challenge, we present an XML keyword search engine, XSeek, to infer the semantics of the search and identify return nodes effectively. XSeek recognizes possible entities and attributes inherently represented in the data. It also distinguishes between search predicates and return specifications in the keywords. Then based on the analysis of both XML data structures and keyword match patterns, XSeek generates return nodes. Extensive experimental studies show the effectiveness of XSeek.

## Categories and Subject Descriptors

H.3.3 [Information Search and Retrieval]: Search Process

## General Terms

Algorithms, Design

## Keywords

XML, keyword search

## 1. INTRODUCTION

As XML becomes the standard for representing web data, effective and efficient methods to query XML data has become an increasingly important problem. Traditionally, a structured query language, such as XPath [7] and XQuery [1], is used to search XML data, which can convey complex semantic meanings and therefore retrieve precisely the desired results. Nevertheless, there are many situations where keyword search on XML documents is highly desirable. For example, a user may not know the data schema, or the schema is very complex such that a query can not be easily for-

mulated, or a user does not know how to express a search using a structured query language, as typically found in web applications.

Due to the lack of expressivity and inherent ambiguity, there are two main challenges in interpreting the semantics when performing keyword search on XML data.

- First, unlike a structured query where the connection among the data nodes matching the query is specified precisely in the “where” clause (in XQuery or SQL) and/or as variable bindings (in XQuery), we need to automatically connect the match nodes in a meaningful way.
- Second, unlike a structured query where the return nodes are specified using either a “return” clause (in XQuery) or “select” clause (in SQL), we should effectively identify the desired return information.

Several recent attempts have been made on supporting keyword search on XML documents by addressing the first challenge, such as XRank [12], XSearch [8], XKeyword [21], and [17]. For example, consider the XML document in Figure 1(a) about NBA sport teams, where each node is associated with a unique ID. Consider a query “Mutombo, position” ( $Q_3$  in Figure 2). Though there are many XML data nodes matching keyword `position`, only the one with ID 0.2.4.0.1 should be considered as closely related to the match of `Mutombo 0.2.4.0.0.0` since they refer to the same player. This is achieved using a variant of lowest common ancestor concepts, named as *VLCA* in this paper, such as *SLCA* [21], *MLCA* [17], *interconnection* [8], etc.

However, none of the above work has addressed the second challenge: what are appropriate data nodes to be returned after we establish the connection between the matches. There are two baseline approaches for determining return nodes adopted in the existing works. One is to return the subtrees rooted at the VLCA nodes [12, 8, 21], named as *Subtree Return* in this paper. Alternatively, we can return the paths in the XML tree from each VLCA node to its descendants that match an input keyword, as described in [5, 13], named as *Path Return* in the paper. However, neither approach is effective in identifying return nodes.

Let us look at some sample queries listed in Figure 2. For  $Q_1$ , it is likely that the user is interested in the information about `Rockets`. Both *Subtree Return* and *Path Return* first compute the VLCA of the keyword matches, which is the node with ID 0.2.0.0, then output the node `Rockets` itself. However, to echo print the user input without any additional information is not informative. Ideally, we would like to return the subtree rooted at the `team` node with ID 0.2 for information about `Rockets`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 11–14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...\$5.00

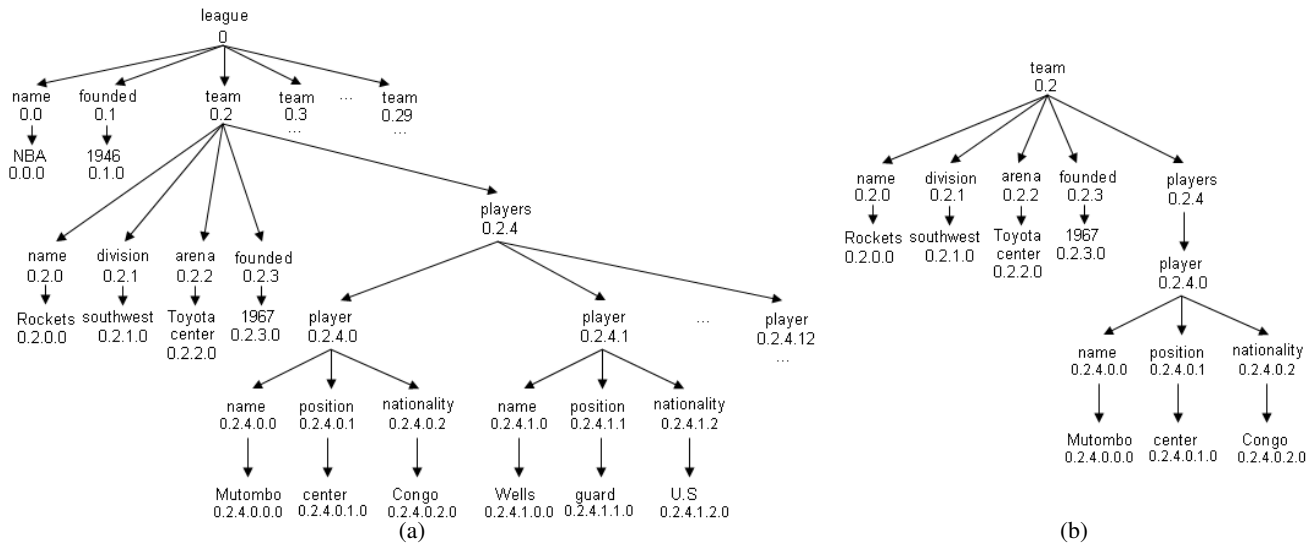


Figure 1: Sample XML Document(a) and Search Result for  $Q_4$ (b)

$Q_1$	Rockets
$Q_2$	Mutombo, center
$Q_3$	Mutombo, position
$Q_4$	team, Rockets, center
$Q_5$	Rockets, players

Figure 2: Sample Keyword Searches

Now let's consider  $Q_2$  and  $Q_3$ . By issuing  $Q_2$ , the user is likely to be interested in information about the player whose name is Mutombo and who is a center in the team. Therefore the subtree rooted at the `player` node with ID 0.2.4.0 is a desired output. In contrast,  $Q_3$  indicates that the user is interested in a particular piece of information: the position of Mutombo.

As we can see, an input keyword can specify a *predicate* for the search, or specify a desired *return* node. However, existing approaches fail to differentiate these two types of keywords. In particular, *Path Return* approach returns the paths from the VLCA `player` node (0.2.4.0) to Mutombo and to center for  $Q_2$ , and the path from `player` to Mutombo and `position` for  $Q_3$ , respectively. On the other hand, since  $Q_2$  and  $Q_3$  have the same VLCA node `player` 0.2.4.0, *Subtree Return* outputs the subtree rooted at this node for both queries, though the user indicates that only the `position` information is of interest in  $Q_3$ .

Now let's look at a more complex query  $Q_4$ , intending to find information about the player who is a center in the team Rockets. The desired query result is shown in Figure 1(b). *Subtree Return* approach outputs the whole tree rooted at `team` (0.2), and requires the user him/herself to search the relevant player information in this big tree. On the other hand, *Path Return* approach outputs the path from `team` to Rockets and to center, without providing any additional information about the `player` (0.2.4.0) node.

Finally, if the input keywords match a big subtree, displaying the whole tree at once can be overwhelming to the user. It is more desirable to output the most relevant information first, and then provide *expansion links*, so that the user can click and browse detailed information. For example, to process  $Q_5$ , we output the name of

`players` and provide a link to its `player` children which, upon click, provides information about all the players in the team. The question is how to identify the information to be displayed at the beginning, and subsequently, at each expansion step.

As we can see from the above sample queries, existing approaches fail to effectively identify relevant return nodes. Sometimes they suffer low precision, such that users need to browse and screen relevant information from large output themselves, which can be time consuming. Sometimes they have low recall, such that users are not able to get informative results.

The only work that has considered the problem of identifying return nodes is [15, 16]. Both of them require schema information. In addition, [15] requires a system administrator to split the schema graph into pieces, called *Target Schema Segments (TSS)* for search result presentation. [16] requires users or a system administrator to specify a weight of each edge in the schema graph, and then each user needs to specify a *degree constraint* and *cardinality constraint* in the schema to determine the return nodes.

There are several desirable features for determining the return nodes that an XML keyword search system should achieve. First, though schema information can be used whenever possible, its presence should be optional, since XML data may not have an associated schema. Second, it is important for the system to automatically infer return nodes without eliciting preference from users and system administrators due to two reasons: the users who do not issue structured queries are probably unwilling or unable to specify the output schema; and it is hard for a system administrator to specify return nodes that reflect individual user needs. Furthermore, return node identification should consider not only data structure, but also keyword match patterns, as shown in  $Q_2$  and  $Q_3$ .

In this paper, we present a system XSeek that allows users to search information in XML documents by keywords, and identifies meaningful return nodes as exemplified in the above sample data and queries without user solicitation. In this paper, return nodes refer to the names (types) of the data nodes that are the goal of user

searches. To achieve this, we analyze both XML data structure and keyword match patterns. We differentiate three types of information represented in XML data: entities in the real world, attributes of entities, and connection nodes. We also categorize input keywords into two types: the ones that specify search predicates, and the ones that indicate return information that the user is seeking. Based on data and keyword analysis, XSeek generates return nodes, which can be explicitly inferred from keywords, or dynamically constructed according to the entities in the data that are relevant to the search. Finally, data nodes that match predicates and return nodes are output as query results with optional expansion links.

The contributions of our work include:

1. To the best of our knowledge, this is the first work that addresses the problem of automatically inferring desirable return nodes for XML keyword search.
2. We identify and generate return nodes of two types: return nodes that can be inferred explicitly by analyzing keyword match patterns; and return nodes that can be inferred implicitly by considering both keyword match patterns and XML data structure.
3. XSeek outputs the data nodes that match search predicates and return nodes according to node categories: entities, attributes or connection nodes.
4. We have designed and developed an XML keyword search system XSeek that realizes these heuristics in determining the desired search results.
5. Experiments show that XSeek generates results with improved precision and recall over those produced by *Subtree Return* and *Path Return* approaches with good scalability.

The rest of the paper is organized as follows. Section 2 presents the data model and query syntax of the XSeek system. Section 3 discusses the intuition of semantics inference and return node generation. Based on these inferences, we present the algorithms of XSeek in Section 4. The experimental studies are presented in Section 5. Section 6 discusses the related work and Section 7 summarizes the paper and discusses future directions.

## 2. BACKGROUND

**Data Model.** We model XML data as a rooted, labeled, unordered tree, such as the one shown in Figure 1(a). Every internal node in the tree has a node name, and every leaf node has a data value<sup>1</sup>. We model XML attribute nodes as children of the associated element node, and do not distinguish them from element nodes.

Each node in the XML tree is assigned a *Dewey* label [19] as a unique ID. We record the relative position of a node among its siblings, and then concatenate these positions using dot ‘.’ starting from the root to compose the Dewey ID for the node. For example, node with Dewey ID 0.2.3 is the 4-th child of its parent node 0.2. Dewey ID can be used to detect the order, sibling and ancestor information of a node.

1. The start tag of a node  $u$  appears before that of node  $v$  in an XML document if and only if  $Dewey(u)$  is smaller than  $Dewey(v)$  by comparing each component in order.

<sup>1</sup>We do not consider mixed content in this paper, but it can be easily supported by adding a dummy parent node with name “value”.

```
<!ELEMENT league (name, founded, team*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT founded (#PCDATA)>
<!ELEMENT team (name, division, arena, founded, players)>
<!ELEMENT players (player*)>
```

**Figure 3: Sample DTD Fragment**

2. A node  $u$  is an ancestor of node  $v$  if and only if  $Dewey(u)$  is a prefix of  $Dewey(v)$ .
3. A node  $u$  is a sibling of node  $v$  if and only if  $Dewey(u)$  differs from  $Dewey(v)$  only in the last component.

We can also infer that given two nodes  $u$  and  $v$ , their lowest common ancestor (LCA) has a Dewey ID that is equal to the longest common prefix of  $Dewey(u)$  and  $Dewey(v)$ .

**Schema of XML data.** An XML document can optionally have a schema. Besides XML Schema, Document Type Description (DTD) is a commonly used method to describe the structure of an XML document and acts like a schema. For each XML node, it specifies the names of its sub-elements and attributes using regular expressions with operators  $*$  (a set of zero or more elements),  $+$  (a set of one or more elements),  $?$  (optional), and  $|$  (or). For example, Figure 3 shows a fragment of the DTD for the XML document in Figure 1(a). “ELEMENT league (name, founded, team\*)” indicates that a league element should have exactly one child name, one child founded, and zero or more child team. “ELEMENT name (#PCDATA)” specifies that name node has a value child. We refer the nodes that can have siblings of the same name as *\*-node*, as they are followed by a “\*” in the DTD, such as the team and player nodes.

**Keyword Search.** The user input is a set of keywords, each of which may match name and/or value nodes in the XML tree. Since we use an unordered model, query (Mutombo, Congo) and query (Congo, Mutombo) have the same effect.

**Definition 2.1:** If a keyword  $k$  is the same as the name of a node  $u$ , or is contained in the value of a node  $u$ , we say that  $u$  is a *match* to  $k$ . ■

For example, consider the XML tree in Figure 1(a) and  $Q_3$  in Figure 2. The node 0.2.4.0.0.0 is a match to keyword Mutombo, and the node 0.2.4.0.1 is a match to position.

## 3. INFERRING QUERY SEMANTICS

As we have discussed in Section 1, there are two challenges in supporting keyword searches: identifying search predicates by connecting matches, and inferring desirable return nodes.

To address the first challenge, we follow the approach [21] to compute VLCA nodes. An XML node is a VLCA node if its subtree contains at least one match to every keyword, and it does not have a descendant whose subtree contains at least one match to every keyword. For example, consider  $Q_3$  on XML data in Figure 1(a). First we find the set of matches to each keyword:  $\{0.2.4.0.0.0\}$  matches Mutombo,  $\{0.2.4.0.1, 0.2.4.1.1\}$  matches position. Then we compute the VLCA of these matches. Note that the players node 0.2.4 is an LCA of matches 0.2.4.0.0.0 and 0.2.4.1.1. However, it is not considered as a VLCA since it has a descendant

with ID 0.2.4.0 whose subtree contains at least one match to each keyword. In this example, the `player` node 0.2.4.0 is the only VLCA. Alternatively, other approaches to compute VLCA nodes such as MLCA [17], Interconnection [8] can be incorporated into the XSeek system for connecting keyword matches.

Then we group the matches according to their ancestor VLCA nodes, such that each group corresponds to one VLCA node and contains at least one match to each keyword. The nodes in the same group are considered to be closely related. Note that by definition, a VLCA does not have another VLCA as its ancestor, therefore each match belongs to at most one group. For  $Q_3$ , we have one group, consisting of `Mutombo` (0.2.4.0.0.0) and `position` (0.2.4.0.1).

Now we focus the discussion on how to address the second challenge: given a group of keyword matches and their VLCA, how can we identify meaningful return nodes?

### 3.1 Analyzing XML Data Structure

To decide what information should be returned, we need to understand the roles and relationships of nodes in the data. The information in XML documents can be recognized as a set of real world entities, each of which has attributes<sup>2</sup>, and interacts with other entities through relationships. This mimics the *Entity-Relationship* model in relational databases.

For example, for the XML data in Figure 1(a), conceptually we can recognize two types of entities: `team` and `player`. Each type of entity has certain attributes. `team` has `name`, `division`, `arena` and `founded` as attributes. `player` has attributes `name`, `position` and `nationality`. The relationships between entities are represented by the paths connecting them. For example, a `team` has one or more `player` nodes.

We believe that by issuing a query a user would like to find out information about entities along with their relationships in a document. Therefore to determine the search result, we should consider the entities in the document that are related to the input keywords.

For example, consider query  $Q_1$  that searches `Rockets`. Very likely the user would like to find out the information about the real world entity that `Rockets` corresponds to. Therefore, we first identify that the `team` node with ID 0.2 is the corresponding entity of `Rockets`. Then, we output the information of this `team` entity. The attributes associated with the `team` entity are considered as the most important and relevant information, and are output at the first place. The relationship between `team` and `player` is considered to be of secondary importance. A link is generated for the relationship to `players`, which allows users to click and browse the interacting entities.

**The first guideline** of the XSeek system is to differentiate nodes representing entities from nodes representing attributes, and generate return nodes based on the entities related to the keyword matches.

However, since XML data may be designed and generated by autonomous sources, we do not necessarily know which nodes represent entities, and which represent attributes directly. Next we present heuristics for inferring entities and attributes in two situations: when the schema is available, and when it is absent.

<sup>2</sup>In the rest of the paper, *attribute* refers to the one defined in an ER-model, rather than the one defined in XML specification [20].

When the schema of an XML document is available, we classify nodes into entities and attributes according to their node relationships. If a node with name  $n_1$  has a one-to-many relationship with nodes with name  $n_2$ , then very likely  $n_2$  represents an entity rather than an attribute of  $n_1$ . On the other hand, a one-to-one relationship is more likely to introduce an attribute.

In general, we make the following **inferences on node categories**.

1. A node represents an *entity* if it corresponds to a \*-node in the DTD.
2. A node denotes an *attribute* if it does not correspond to a \*-node, and only has one child, which is a value.
3. A node is a *connection* node if it represents neither an entity nor an attribute. A connection node can have a child that is an entity, an attribute or another connection node.

For example, consider the DTD fragment in Figure 3 for the XML data in Figure 1(a). `team` is a \*-node, indicating a many-to-one relationship with its parent node `league`. We infer that `team` represents an entity that has a relationship with `league`, instead of an attribute of `league`. `name`, `division`, `arena`, and `founded` are considered as attributes of a `team` entity. On the other hand, `players` is not a \*-node and it doesn't have a value child, therefore it is considered as a connection node.

Though these inferences do not always hold (for example, one person may have multiple attributes of phone number, and the `league` node should be considered as an entity instead of a connection node), they provide good heuristics in the absence of the E-R model.

When the schema information is not available, we infer the schema based on data summarization, similar as [9, 22]. Then we identify the entities and attributes in the data according to the inferred schema. For example, since the `player` node can occur more than once within its parent `players` in the data, it is considered as a \*-node.

### 3.2 Analyzing Keyword Match Patterns

Besides studying the structure of XML data and inferring inherent entities and attributes presented in the data, we also analyze the pattern of the keyword matches to infer search predicate and return node specifications.

Recall  $Q_2$  and  $Q_3$  in Figure 2, which have the same set of VLCA nodes and keyword match groups. Existing approaches do not differentiate these two queries while generating results. However, different patterns of these two queries imply different user intentions.  $Q_2$  searches information about `Mutombo` who is a `center`.  $Q_3$  searches the `position` information of `Mutombo`.

**The second guideline** of the XSeek system is to take keyword match patterns into consideration when generating search results, by classifying keywords into two categories: search predicates and return nodes.

1. Some keywords indicate *predicates* that restrict the search, corresponding to the *where* clause in XQuery or SQL.
2. Some keywords specify *return nodes* as the desired output type, corresponding to the *return* clause in XQuery or the *select* clause in SQL.

Since keyword queries do not have any structure, the immediate question is how we should infer predicates and return nodes. To achieve this, we first differentiate data *types* from data *values*. If schema information is available, we can obtain type information directly. Otherwise, we use node *names* to indicate data types. Recall that in a structured query language such as XQuery or SQL, typically a predicate consists of a pair of type and value, while a return clause only specifies data types without value information (whose values are expected to be query results). For example, consider an SQL query: `select position from DB where name = "Mutombo"`.

Based on this observation, we make the following **inferences on keyword categories** for each match group.

1. If an input keyword  $k_1$  matches a node name (type)  $u$ , and there does not exist an input keyword  $k_2$  matching a node value  $v$ , such that  $u$  is an ancestor of  $v$ , then  $k_1$  specifies a *return node*.
2. A keyword that does not indicate a return node is treated as a *predicate* specification. In other words, if a keyword matches a node value, or it matches a node name (type) that has a value descendant matching another keyword, then this keyword specifies a predicate.

For example, in  $Q_2$ , `center` is considered as a predicate since it matches a value (0.2.4.0.1.0). Similarly, `Mutombo` in both  $Q_2$  and  $Q_3$  are considered as a predicate. `team` in  $Q_4$  is also inferred as a predicate since it matches a name (0.2) which has a descendant value node (0.2.0.0) that matches another keyword `Rockets`. On the other hand, `position` in  $Q_3$  is considered as a return node since it matches the name of two nodes (0.2.4.0.1, 0.2.4.1.1), neither of which has any descendant value node matching another keyword in  $Q_3$ . Similarly, `players` in  $Q_4$  is also treated as a return node.

### 3.3 Generating Search Results

Once we have identified the inherent entities and attributes in the data according to Section 3.1 and the keyword match patterns according to Section 3.2, we generate search results accordingly.

**The third guideline** of XSeek is to output data nodes that match query predicates and return nodes as search results.

**Outputting Predicate Matches.** One difference between performing keyword search and processing a structured query is that, besides outputting the data matching the potential return nodes, we should also output the data matching predicates. Since in face of the inherent ambiguity of keyword search, the user often would like to check the predicate matches and make sure that the predicates are satisfied in a meaningful way. Therefore, the paths from a VLCA node<sup>3</sup> to each of its descendant matches will be output as part of search results, indicating how the keywords are matched and how the matches are connected to each other. For example, for  $Q_3$ , even though the user is only interested in the `position` information, we also output the path from the VLCA `player` (0.2.4.0) node to the predicate match `Mutombo` (0.2.4.0.0.0).

**Identifying Return Nodes.** Return nodes can be *explicit* or *implicit*. For some queries, we can infer explicit return nodes from

<sup>3</sup>In fact, the paths start from the master entities as defined later.

keyword matches as we have discussed in Section 3.2. For example, `position` is an explicit return node in  $Q_3$ . For other queries, all the input keywords are considered as predicates, and no return nodes can be inferred from the keywords themselves, such as  $Q_2$ . In this case, we believe that the user is interested in the general information about the entities related to the search. We define master entity and relevant entity in the following, and consider them as implicit return nodes when the input keywords do not have explicit return nodes specified.

**Definition 3.1:** If an entity  $e$  is the lowest ancestor-or-self of the VLCA node in a group  $g$ , then  $e$  is named as the *master entity* of group  $g$ . If such an entity  $e$  can not be found, the root node of the XML tree is considered as the master entity. ■

Since VLCA is the lowest common ancestor of all the matches in a group, master entity is the lowest common entity ancestor of them. All the information that is considered to be relevant to the query is in the subtrees rooted at master entities.

**Definition 3.2:** If an entity  $e$  is an ancestor-or-self of a keyword match  $v$  in a group  $g$ , and it is a descendant-or-self of the master entity, then  $e$  is a *relevant entity* with respect to  $v$  for the group  $g$ . ■

In  $Q_2$ , since both `Mutombo` and `center` are inferred as predicates, no explicit return nodes are specified in the keywords. We first identify the `player` node (0.2.4.0) as the VLCA node. It is in fact the master entity, and the only relevant entity in the group. Therefore `player` is treated as the implicit return node for  $Q_2$ .

**Outputting Return Nodes Based on Node Categories.** After we have identified explicit or implicit return nodes, the next question is how to display them appropriately. The data nodes that match return nodes will be displayed according to their categories: attributes, entities, and connection nodes. To output an attribute, we display its name and value. On the other hand, the subtrees rooted at entities and connection nodes can be big. Rather than outputting the whole subtree at once, it is often more user-friendly to display the most relevant information at the first stage with expansion links to less relevant information. Then the user may click links and browse for more details. The question is what information should be returned at the first stage. For an entity or a connection node, we first output its name. We additionally output all attribute children of an entity. Then we generate a link to each group of child entities that have the same name (type), and a link to each child connection node (except those that have descendants matching keywords, which will be output at the first stage).

For example, consider  $Q_1$ . We infer `team` as an implicit return node since it is a relevant entity and no return node is specified in the keywords. We display its name `team`, the names and values of its attributes `name`, `division`, `arena` and `founded`. Then we generate an expansion link to its connection child `players`. On the other hand, for  $Q_5$ , a connection node `players` is inferred as an explicit return node from the input keywords. We display its name `players`, and a link to its child entities `player`. If the `players` node has another set of children named as `former_player`, we also output an expansion link for `former_player`.

In summary, we make the following **inferences on search result generation**.

1. We infer return nodes either explicitly from keywords by analyzing keyword match patterns, or implicitly by considering both keyword matches and relevant entities in the data.
2. The data nodes that match return nodes are output based on their node categories: attributes, entities and connection nodes.
3. Besides outputting the matches to return nodes, data nodes that match search predicates are also output such that the user can verify the meaning of the matches.

## 4. ALGORITHMS

After we have discussed the semantics of XSeek, we present the algorithms that process keyword searches on XML data and achieve the semantics efficiently.

### 4.1 Data Processing and Index Construction

XSeek parses the input XML documents and categorizes each node as an entity, an attribute, or a connection node by leveraging the schema information if available or by data summarization, according to the inferences discussed in Section 3.1.

To speed up query processing, three indexes are built in XSeek.

*Name index* retrieves the Dewey IDs of the nodes in document order whose names match the input keyword. It supports the operation:

- $Name2ID(name)$ .

*Value index* retrieves the Dewey IDs of the nodes in document order whose values contain the input keyword, similar as the inverted index in information retrieval. It supports the operation:

- $Value2ID(value)$ .

We also have a *Dewey index*, which retrieves the XML node entry for a given Dewey ID. Each node entry contains the information about node name, number of its children, pointers to child entries and parent entry. Furthermore, we also record the inferred node category, i.e. whether the node is an entity, attribute or connection node. The index supports the following operations:

- $ID2Node(ID)$  returns the node entry in the Dewey index for a given Dewey ID.
- $isEntity(ID)$  returns *true* if the node with the given Dewey ID is inferred as an entity, and *false* otherwise.
- $isAttribute(ID)$  returns *true* if the node with the given Dewey ID is inferred as an attribute, and *false* otherwise.

Name index and value index are implemented as hash tables. Dewey index is implemented as a B+ tree clustered by Dewey ID. Since we often need to traverse nodes with consecutive Dewey numbers, B+ tree can improve access locality and achieve efficiency.

### 4.2 Query Processing

The algorithm *KeywordSearch* is presented in Algorithm 1 and 2, which identifies and outputs meaningful return information for XML keyword search. There are four stages in the process. First, the *findMatch* procedure retrieves all the XML nodes matching the input keywords. Then the *computeVLCA* procedure, adopted from [21], computes the VLCA nodes of the matches. Based on the VLCA nodes, the *groupMatch* procedure groups the matches such that each group corresponds to a VLCA node and contains at least one match to every keyword. Finally, the *genResult* procedure generates search results for each group. Next we will discuss

each stage in detail using  $Q_4$  in Figure 2 as a running example, which intends to search for the player who is a center in the team `Rockets`.

**Matching Keywords.** For a set of input keywords  $keyword[n]$ , we start with the procedure *findMatch* that retrieves the list of data nodes  $KWmatch[i]$  that match  $keyword[i]$ ,  $1 \leq i \leq n$ . The node lists are obtained by accessing the *name index* and *value index* using  $Name2ID$  and  $Value2ID$  operations, respectively. For each match, we record whether it is a name or a value.

**Example 4.1:** We start with retrieving the list of nodes matching each keyword in  $Q_4$ : `team`: (0.2, 0.3, ..., 0.29), `Rockets`: (0.2.0.0), and `center`: (0.2.4.0.1.0), respectively. ■

**Computing VLCA.** The procedure *computeVLCA* computes the VLCA nodes from  $KWmatch$  according to the algorithm proposed in [21]. Recall that an XML node is a VLCA node if its subtree contains at least one match to every keyword, and it does not have a descendant whose subtree contains at least one match to every keyword. Due to space constraints, we direct readers to [21] for computation details.

**Example 4.2:** Consider a match node 0.2 to keyword `team`, a match node 0.2.0.0 to `Rockets`, and a match node 0.2.4.0.1.0 to `center`. Their LCA node is the `team` node 0.2. Whereas for the `Rockets` match 0.2.0.0, `team` match 0.3, and `center` match, say 0.3.4.0.1.0 (not shown in Figure 1(a)), their LCA is the `league` node 0. By definition, the second LCA is disqualified as an VLCA node. In this example, there is only one VLCA node: the `team` node 0.2. ■

**Grouping Matches.** Then the *groupMatch* procedure groups the keyword matches  $KWmatch$  based on their ancestor VLCA node (if exists). For each group, it infers whether a keyword represents a predicate or an explicit return node, according to the inferences discussed in Section 3.2.

First we merge the list  $KWmatch[i]$ ,  $1 \leq i \leq n$ , ordered by Dewey ID to produce a list *Allmatch* ordered by Dewey ID.

Then we differentiate two types of keywords, the ones that represent predicates, and the ones that represent return nodes, as discussed in Section 3.2. We observe that keyword matches can have ancestor-descendant relationship. An ancestor match restricts the descendant match(es) and therefore is considered to be a predicate. If we remove the matches that are ancestors of others, then a remaining match is considered as a predicate if it is a value node, otherwise (the match is a name node) it is considered as an explicit return node. As discussed in Section 3.3, since the path from the master entity to each remaining match will be output, the information of the removed matches will be output to users without losing information. The procedure of removing ancestor matches can be efficiently achieved by removing every match  $u$  that is an ancestor of the next match  $v$  in the *Allmatch* list. This guarantees that all the ancestor matches will be removed. To see this, suppose  $u$  is an ancestor of a set of matches, among which  $v$  is the one that occurs first in document order, then  $v$  must be the node immediately after  $u$  in the *Allmatch* list, as *Allmatch* is sorted by Dewey ID.

---

**Algorithm 1** Matching Keywords and Grouping Match Nodes

---

```
KeywordSearch (keyword[n], indexes)
1: KWmatch = findMatch(keyword)
2: VLCA = computeVLCA(KWmatch) {adopted from [21]}
3: group, retSpecified = groupMatch(KWmatch, VLCA)
4: for j = 1 to m do
5:   currMatch = group[j][1]
6:   genResult(findEntity(VLCA[j]))
7: end for
findMatch (keyword[n])
1: for i = 1 to n do
2:   KWmatch[i] = Name2ID(keyword[i])      ∪
   Value2ID(keyword[i])
3: end for
groupMatch (KWmatch[n], VLCA[m])
1: {merge KWmatch[i], 1 ≤ i ≤ n, in Dewey ID order to one list
   Allmatch in Dewey ID order}
2: Allmatch[l] = merge(KWmatch[1], ..., KWmatch[n])
3: {remove ancestor matches}
4: for i = 1 to l - 1 do
5:   if isAncestor(Allmatch[i], Allmatch[i + 1]) then
6:     remove Allmatch[i]
7:   end if
8: end for
9: {group matches}
10: j = k = 1
11: while (j ≠ m) or (k ≠ l) do
12:   if isAncestor(VLCA[j], Allmatch[k]) then
13:     group[j] = group[j] ∪ Allmatch[k]
14:     k ++
15:     if isName(Allmatch[k]) then
16:       retSpecified[j] = true
17:     end if
18:   else if group[j] ≠ ∅ then
19:     j ++
20:   else
21:     k ++
22:   end if
23: end while
findEntity (v)
1: for each node u along the path from v to the root node do
2:   if isEntity(u) then
3:     return u
4:   end if
5: end for
```

---

Finally, we group the keyword matches in *Allmatch* to *groups*. Each *VLCA*[*j*],  $1 \leq j \leq m$ , corresponds to a group *group*[*j*], such that the matches in *group*[*j*] are descendants of *VLCA*[*j*]. Since the VLCA nodes do not have ancestor-descendant relationship, any match can have at most one VLCA ancestor, and therefore belongs to at most one group. On the other hand, by definition every *VLCA*[*j*] must have one non-empty group associated with it. Due to these two properties and the fact that *VLCA* and *Allmatch* are sorted by Dewey ID, the grouping can be achieved by a single traversal of *VLCA* and *Allmatch*, during which matches that do not belong to any group (i.e. do not have a VLCA ancestor) are discarded. If the current node in *Allmatch*[*k*] is a descendant of *VLCA*[*j*], then it is added into *group*[*j*]. Otherwise, if *group*[*j*] is not empty, since it is possible that *Allmatch*[*k*] is a descendant of *VLCA*[*j* + 1], we move the cursor of *VLCA* and *group* (*j* ++). If *group*[*j*] is empty, then *Allmatch*[*k*] does not have any VLCA node as its ancestor and does not belong to any group, therefore it is discarded (*k* ++). If there is at least one keyword match identified as an explicit return node in *group*[*j*], we set the boolean variable *retSpecified*[*j*] to be *true*.

---

**Algorithm 2** Generating Results

---

```
genResult (v)
1: if (v.name = currMatch) then
2:   {v is an explicit return node}
3:   output Ret(v)
4: else if retSpecified[j] = false and isEntity(v) then
5:   {v is an implicit return node}
6:   output Ret(v)
7: else if isAttribute(v) then
8:   output v.name and v.value
9: else
10:  output v.name
11: end if
12: u = the first child of v that is an ancestor-or-self of currMatch
13: while (currMatch ≠ null) and (u ≠ null) do
14:   genResult(u)
15:   if u.name = currMatch or u.value = currMatch then
16:     currMatch ++ {move to the next match in the group}
17:   end if
18:   u = v's next child that is an ancestor-or-self of currMatch
19: end while
outputRet (v)
1: if isAttribute(v) then
2:   output v.name and v.value
3: else
4:   {v is an entity or connection node}
5:   output v.name
6:   if isEntity(v) then
7:     output v's attribute children that do not match a keyword
8:   end if
9:   for each of v's entity child with a distinct name and connection child
   w that does not have a descendant match do
10:    generate a link to w
11:   end for
12: end if
```

---

**Example 4.3:** Continuing our running example, we merge three lists of matches and produce the *Allmatch*: {0.2, 0.2.0.0, 0.2.4.0.1.0, 0.3, 0.4, ..., 0.29}. Since *team* node (0.2) is an ancestor of the next match node *Rockets* (0.2.0.0), it is removed from the list. Now the remaining nodes in *Allmatch* are: *Rockets* (0.2.0.0), *center* (0.2.4.0.1.0), *team* (0.3), *team* (0.4), ..., *team* (0.29).

Next we group *Allmatch* based on the VLCA {0.2}. We build one group *group*[0] for this VLCA. Since node 0.2.0.0 and 0.2.4.0.1.0 are descendants of this VLCA node, they are put into *group*[0]. On the other hand, the *team* nodes from 0.3 to 0.29 are discarded as they are not descendants of any VLCA node. In *group*[0], since both matches *Rockets* (0.2.0.0) and *center* (0.2.4.0.1.0) are value nodes, they are considered as predicates, no explicit return nodes can be inferred, and *retSpecified*[0] is set to be *false*. As discussed in Section 3.3, in this case we will infer implicit return nodes from relevant entities in the data. ■

**Generating Results.** After keyword matches are grouped according to their VLCA nodes, if no explicit return node are specified in a group, implicit return nodes will be inferred. Then XSeek generates search results by outputting data nodes that match search predicates and return nodes as discussed in Section 3.3.

We start by retrieving the master entity of a group using procedure *findEntity*, which accesses the ancestors of a VLCA node in order until an entity or the XML tree root is reached.

The *genResult* procedure navigates the paths from the master entity to each match in a group, identifies and outputs the matches to

predicates and explicit or implicit return nodes. Initially *genResult* is invoked on the master entity *v*, and then it is recursively invoked on the children of *v* that are on the path from *v* to a match in the group in order. If return nodes are not explicitly specified (*retSpecified = false*), then each entity node *v* on those paths (i.e. a relevant entity) is considered as an implicit return node. Eventually, *genResult* is invoked on one of the matches *v*. If the match is a node name (*v.name = currMatch*), then it is considered as an explicit return node (*retSpecified = true*). If a node *v* is an explicit or implicit return node, then we invoke procedure *outputRet(v)*, which display *v* according to its category. If *v* is an attribute, its name and value will be output. Otherwise (*v* is an entity or connection node), we output *v*'s name. Attribute children of an entity node will be output. Furthermore, a link to each distinct name (type) of *v*'s children that are entities or connection nodes and don't have descendant matches is displayed. After a match is processed, we move to the next match in the group.

**Example 4.4:** In the running example, since the VLCA node *team* (0.2) is itself an entity, it is the master entity and a relevant entity of this group. The *genResult* procedure is invoked on the master entity. The variable *currMatch* is initialized to be *Rockets* (0.2.0.0). Since there is no explicit return node in this group, *team* is considered as an implicit return node, and its name *child* that matches *currMatch*, its *division*, *arena* and *founded* attributes are output. *currMatch* is moved to the next match in the group, the *center* node (0.2.4.0.1.0). Then *team*'s child *players* that is on the path to *currMatch* is navigated and has its node name displayed. Recursively, we navigate the child of *players* that is on the path to *currMatch*: *player* (0.2.4.0). Since this *player* node is an entity, it is also considered as an implicit return node. We navigate and display its attribute *child* name, its child *position* that matches *currMatch*, and another attribute *nationality*. Since there is no other matches in the group, we have generated the search result for this group, as shown in Figure 1(b). ■

## 5. EXPERIMENTS

To evaluate the effectiveness of XSeek, we compare its performance with two search result generation approaches as introduced in Section 1. *Subtree Return* outputs the whole subtree rooted at each VLCA. *Path Return* outputs the paths from each VLCA to the matches in the subtree rooted at the VLCA. All three approaches adopt [21] for computing VLCA from keyword matches.

We have tested three metrics to compare these approaches: the *quality* of the search results measured by precision, recall and F-measure, the *speed*, and *scalability* upon an increase of document and query size, and a decrease of the height of VLCA.

### 5.1 Experimental Setup

The experiments were performed on a 3.60GHz Pentium 4 machine running Windows XP, with 2GB memory and one 160GB hard disk (7200rpm).

The experiments are performed on three XML data sets. The characteristics of the data sets and query sets are shown in Figure 4.

**Data Sets.** We have tested three XML data sets, Mondial, WSU<sup>4</sup>,

<sup>4</sup>Both Mondial and WSU are available at <http://www.cs.washington.edu/research/xml/datasets>. We repli-

WSU: 4.7MB	
<i>QW</i> <sub>1</sub>	course, title
<i>QW</i> <sub>2</sub>	course, title, days, crs, credit, sect
<i>QW</i> <sub>3</sub>	ECON, 572, place, times
<i>QW</i> <sub>4</sub>	CAC, 101
<i>QW</i> <sub>5</sub>	42879, title, days
<i>QW</i> <sub>6</sub>	42606, TU, TH
<i>QW</i> <sub>7</sub>	root, MILES, course
<i>QW</i> <sub>8</sub>	ECON
Mondial: 6.0MB	
<i>QM</i> <sub>1</sub>	organization, name, members
<i>QM</i> <sub>2</sub>	country, population
<i>QM</i> <sub>3</sub>	mondial, Africa
<i>QM</i> <sub>4</sub>	Belarus, population
<i>QM</i> <sub>5</sub>	mondial, country, Muslim
<i>QM</i> <sub>6</sub>	Croatia
<i>QM</i> <sub>7</sub>	Bulgaria, Serb
<i>QM</i> <sub>8</sub>	Group_of_77, members
Auction: 24.5MB	
<i>QA</i> <sub>1</sub>	closed_auction, price
<i>QA</i> <sub>2</sub>	closed_auction, price, date, itemref, quantity, type, seller, buyer
<i>QA</i> <sub>3</sub>	open_auction, person257
<i>QA</i> <sub>4</sub>	person0, address
<i>QA</i> <sub>5</sub>	closed_auction, buyer, person133
<i>QA</i> <sub>6</sub>	person257, person133
<i>QA</i> <sub>7</sub>	seller, person179, buyer, price, date
<i>QA</i> <sub>8</sub>	seller, 04/02/1999

Figure 4: Data and Query Sets

and Auction<sup>5</sup>. Mondial is a world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources. WSU is a course description database. Auction is a synthetic benchmark data set generated by the XML Generator from XMark using the default DTD.

**Query Sets.** We have tested eight queries for each data set. Among them, the first two only involve node names without values, therefore have relatively low selectivity (and large query result size); the rest six involve node values.

### 5.2 Search Quality

To measure the search quality, we use *precision*, *recall*, and *F-measure*, defined as follows:

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

where *Rel* is the set of relevant nodes (i.e. desired search results), *Ret* is the set of nodes returned by a system, and we use  $|S|$  to denote the number of elements in a set *S*. Precision measures the percentage of the output nodes that are desired, recall measures the percentage of the desired nodes that are output.

F-measure is the weighted harmonic mean of precision and recall, and can be computed as:

$$F = \frac{(1 + \alpha) \times precision \times recall}{\alpha \times precision + recall}$$

When  $\alpha = 1$ , precision and recall are evenly weighted. Another two commonly used F-measures are  $\alpha = 0.5$ , assigning a larger weight to precision, and  $\alpha = 2$ , assigning a larger weight to recall.

cated the WSU and Mondial data sets three times to get a larger data size.

<sup>5</sup><http://monetdb.cwi.nl/xml/>

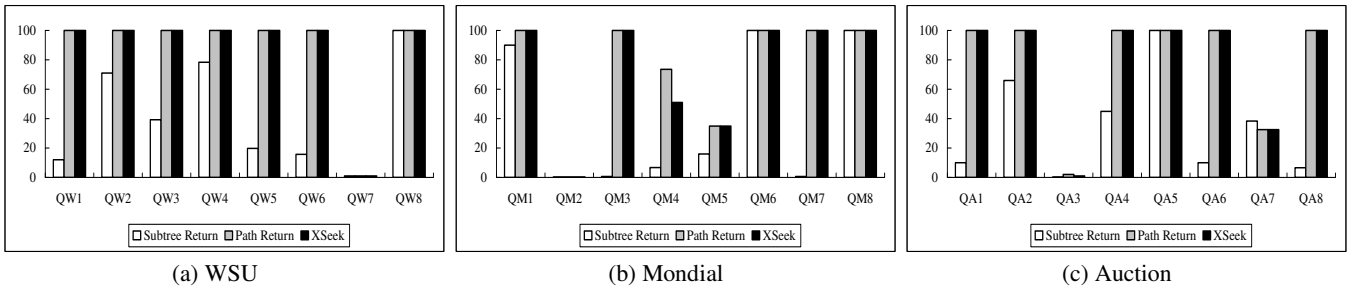


Figure 5: Precision Measurement

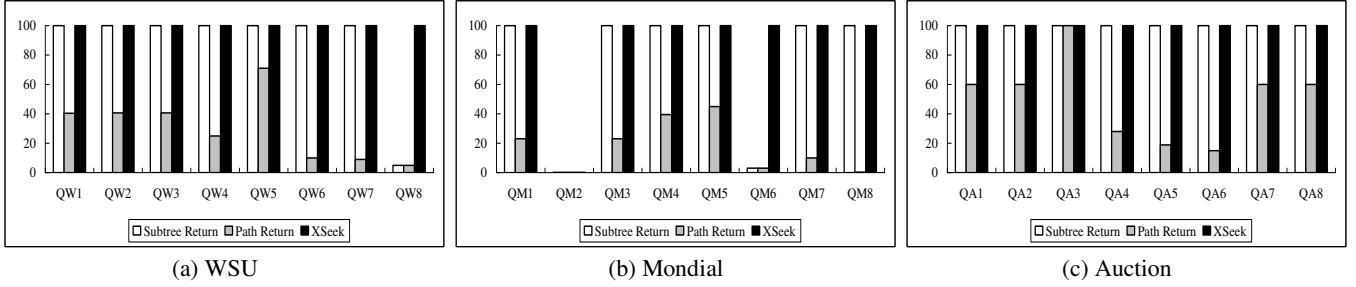


Figure 6: Recall Measurement

Each keyword search is expressed as an English sentence, according to which XML fragments are extracted from the original document and are set as ground truth, *Rel*. Then, we exam the output generated by each system, *Ret*, and count how many nodes in *Ret* appear in *Rel*.

The precision and recall of three approaches on each test data and query set are shown in Figure 5 and Figure 6, respectively.

As we can see, *Subtree Return* usually has a perfect recall as the whole subtree rooted at each VLCA is returned. The only exceptions are *QM<sub>6</sub>* Croatia and *QW<sub>8</sub>* ECON, both of which consist of a single value, which is in turn returned as the search result. The precision of *subtree return* is usually low as not all the nodes in the subtree rooted at each VLCA are relevant. In particular, consider *QM<sub>3</sub>* mondial, Africa, the whole document tree rooted at mondial is returned, even though the user is only interested in the information about Africa in the mondial document.

On the other hand, *Path Return* has the best precision, even perfect precision in many cases, since the matches to predicates and the paths connecting these matches are almost always considered to be relevant. However, it often has a low recall. Consider *QM<sub>3</sub>* again, no information about Africa will be returned except the input keyword themselves.

XSeek has in general a high precision and recall across different queries on the data sets. Its precision is almost the same as *Path Return*, and its recall is almost the same as *Subtree Return*, or even better for some queries.

However, there are several cases that XSeek needs to be improved. For example, *QM<sub>2</sub>* country population intends to search the population of each country. However, XSeek returns the population of all the cities instead of the population of countries, there-

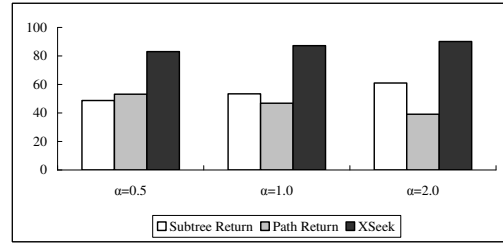


Figure 7: F-measure of All Test Queries

fore suffers a zero precision and a zero recall. This is because in Mondial document, each `country` node has `city` children, and each `city` node has an attribute node named `country`. Both `country` and `city` element nodes have a child `population`. According to the definition of VLCA that these three approaches adopt from [21], `city` instead of `country` is the VLCA. The element nodes matching `country` are not descendants of any VLCA nodes and are discarded, and all three approaches fail. A similar problem exists for *QM<sub>4</sub>*. The only difference is that the VLCA node is the `country` element node that has a descendant `Belarus`, and therefore the population of the country will be output along with city populations.

For *QW<sub>7</sub>*, all three systems suffer a low precision. This is because the `root` node is considered as the VLCA, all the matches of `course` and `MILES` are in the same group and are returned even though the course instructor is not `MILES`. Similar situation occurs for *QM<sub>5</sub>* where the root node `mondial` is searched.

To process *QA<sub>3</sub>*, under the `open_auctions` node we find an `open_auction` node which has a descendant `person257`, so this `open_auction` node is a VLCA, and its subtree should be the desired output. Besides, under the `people` node, there's a per-

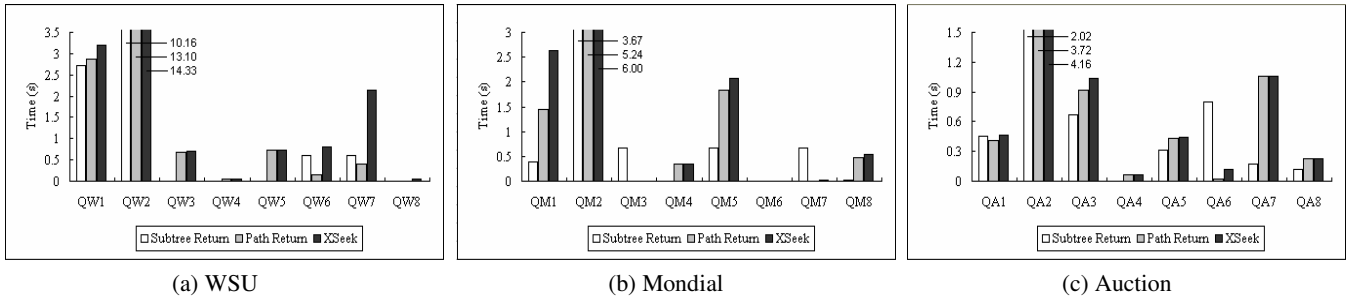


Figure 8: Processing Time

son named `person257` and there are a lot of `open_auction` nodes, but no `open_auction` node is associated with the person `person257`, so these nodes are not desired. However, since the `people` node is another VLCA of this query, all approaches output the corresponding information within the subtree rooted at `people` (which is very large), leading to a low precision.

In  $QA_7$ , the order of the keywords indicates that the user is interested in the buyer, price and date of the auction whose seller is `person179`. However, XSeek does not consider the order of keywords, and returns auctions whose buyer is `person179`, and therefore has a low precision.

Furthermore, we compute the F-measure of each approach according to the average precision and recall across all the test queries, with parameter  $\alpha = 0.5, 1$  and  $2$ , as presented in Figure 7. As we can see, XSeek significantly outperforms the *Subtree Return* and *Path Return* approaches.

### 5.3 Processing Time

The processing times of XSeek, *Subtree Return* and *Path Return* on the test data and query set are shown in Figure 8.

All three approaches need to search for keyword matches and compute VLCA nodes from the matches. Then *Subtree Return* requires time to output subtrees rooted at VLCA nodes. *Path Return* requires time to group keyword matches, and then output the paths from each VLCA to the matches in the corresponding group. XSeek also groups matches and accesses the path from the master entity to the matches in each group. Furthermore, XSeek needs to determine the predicates and explicit or implicit return nodes based on XML data structure and keyword match patterns, and output them accordingly as discussed in Section 3.3.

Now let us compare these three approaches. Compared with *Subtree Return*, both *Path Return* and XSeek additionally traverse all the keyword matches and group them according to their ancestor VLCAs, which has processing time proportional to the total number of matches in a document. On the other hand, *Subtree Return* outputs the subtree rooted at each VLCA directly without additional processing on keyword matches that do not have VLCA ancestors. Therefore, when there are a lot of matches that do not belong to any VLCA (e.g. for  $Q_4$  in Figure 2, the `team` nodes except the first one don't belong to any VLCA), *Path Return* and XSeek take longer processing times than *Subtree Return*, such as  $QW_3, QW_5, QM_1, QM_4, QM_8$  and  $QA_7$ . On the other hand, if the subtrees rooted at VLCA nodes are very big, *Subtree Return* is slower than *Path Return* and XSeek, such as  $QM_3, QM_7$  and  $QA_6$ .

The processing time of *Path Return* is a lower bound of that of XSeek. For most of the queries, XSeek takes almost the same amount of time as *Path Return*, indicating the efficiency of inferring keyword match patterns. On the other hand, for those queries that XSeek outputs a lot more information than *Path Return*, such as  $QW_6, QW_7, QM_1, QA_6$ , XSeek is slower. Indeed, *Path Return* fails to return relevant information other than keyword matches and suffers a low recall on these queries, as shown in Figure 6.

In summary, XSeek generates search results with improved quality and reasonable cost for most of the test queries.

### 5.4 Scalability

We tested the scalability of XSeek on the Auction data set over three parameters: document size, query size, and the depth of VLCA. Since the complexity and scalability of calculating VLCA were presented in [21], we only test the scalability of grouping matches and generating search results in this section.

**Document Size.** We replicated the Auction data set of size 4.8MB between 1 and 8 times to get increasingly large data sets. The processing time of queries  $QA_1$  and  $QA_2$  is shown in Figure 9. As we can see, the processing time of all three approaches increases linearly when the document size increases. In  $QA_1$ , there is only one return node, and the processing times of three approaches are close, while in  $QA_2$  where seven return nodes are inferred, *Subtree Return* is the fastest.

**Number of Keywords.** The experiments were performed on the Auction data set of size 24.5 MB for queries with an increasing number of keywords. We differentiate two different cases.

For queries with an increasing number of return nodes, a constant number of predicates, and a constant depth of VLCA nodes, the performance of three approaches are presented in Figure 10. Seven queries are tested, from  $QA_1$ : `closed_auction price` to  $QA_7$ : `closed_auction price date itemref quantity type seller buyer`, adding one keyword each time.

As we can see, since the set of VLCA nodes and the subtrees rooted at VLCAs remain the same across different queries, the result size and therefore the result generation time of *Subtree Return* remains the same across the queries. On the other hand, as the number of return nodes increases, the result size and therefore the result generation times of *Path Return* and XSeek increase linearly. Note that these two approaches have almost the same processing time as *Subtree Return* when there is only one return node in the query, but

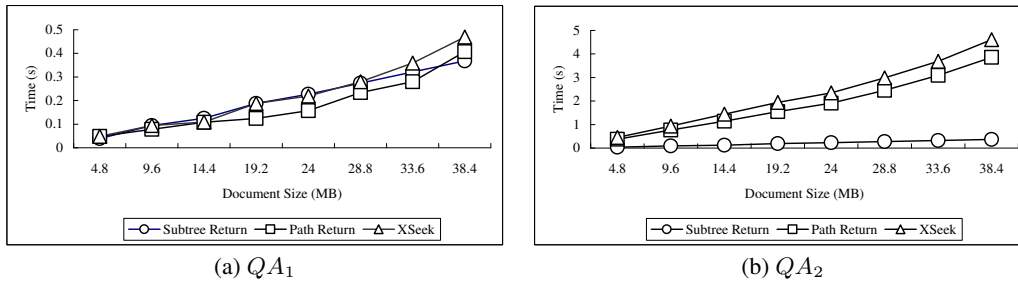


Figure 9: Processing Time with Increasing Document Size

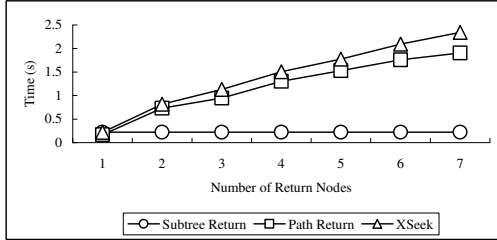


Figure 10: Processing Time with Increasing Return Nodes

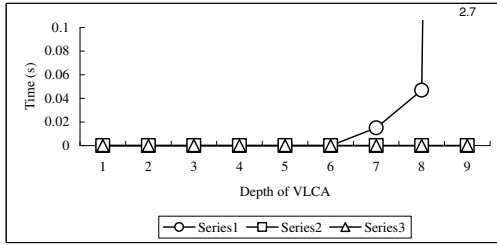


Figure 11: Processing Time with Decreasing VLCA Depth

require more time when the number of return nodes is larger than one.

We have also tested queries with an increasing number of predicates, a constant number of return nodes and a constant depth of VLCA nodes. The test queries are constructed by replacing the node names in the queries of the previous test to their corresponding values. The processing time of all three approaches can almost be neglected as there are few value nodes in the data that match the keywords, and therefore the experimental figure is omitted.

**Depth of VLCA.** The experiments were performed on the Auction data of size 38.4 MB for queries that have VLCAs of a decreasing depth in the XML data tree. Nine queries are tested, from the VLCAs of depth 9 to the document root, decreasing the depth by one each time. The number of match nodes are the same for all nine queries.

From the experimental results presented in Figure 11, we can see that when the depth of VLCA becomes smaller, the size of the subtree rooted at a VLCA increases exponentially, therefore the processing time of *Subtree Return* increases exponentially. On the other hand, the number of output nodes and therefore the result generation time of *Path Return* and *XSeek*, increase very slowly.

In summary, *XSeek* has an improved search quality compared with

*Subtree Return* and *Path Return* by analyzing XML data structure and keyword match patterns without eliciting user specifications. The processing overhead for return node inferences is reasonable. *XSeek* scales well when the document size and/or the query size increase, or the depth of the VLCAs decreases.

## 6. RELATED WORK

In this section, we review the literature on XML keyword search, and categorize them according to how they identify and connect keyword matches, generate return nodes, as well as design ranking schemes.

**Identifying and Connecting Keyword Matches.** A number of studies have been done on inferring the selection condition from keywords by connecting and grouping their matches in a meaningful way. *XRank* [12] connects keyword matches by the LCA nodes that contain at least one occurrence of all keywords after excluding the occurrences of keywords in their descendants that already contain all keywords. *XSearch* [8] introduces the concept of *interconnection*. Two matches are interconnected and therefore should be in the same group if there are no two distinct nodes with the same tag on the path between these two nodes (through their LCA), excluding themselves. [17] and *XKSearch* [21] group matches according to their *meaningful LCA (MLCA)* and *smallest LCA (SLCA)*, respectively. An *SLCA* is the root of a subtree containing matches to all keywords, and it does not have a descendant whose subtree contains all keywords [21]. Two nodes matching to different keywords are considered to be meaningfully related if their LCA is an *SLCA*; a set of nodes consisting of one match to each keyword is meaningfully related if every pair is meaningfully related, and a *MLCA* is defined as the LCA of these nodes [17].

**Outputting Return Information.** There are several typical approaches for determining the return information. The first approach is to return the entire documents that contain keyword matches [6].

Most of the existing approaches return the whole subtrees rooted at the LCA or its variants (*MLCA*, *SLCA*, etc) of keyword matches [12, 8, 21], named as *subtree return* in this paper.

Alternatively, a tree containing the paths from an LCA node to keyword matches can be returned [5], named as *path return* in the paper. Sometime there exist many *path return* subtrees, in order to output information concisely, [13] first reduces each path to an edge labeled with the path length, and then groups the isomorphic reduced subtrees into a *generalized tree*.

Finally, the return information can be specified by users or system administrators. As we discussed in Section 1, *XKeyword* [15] re-

quires a system administrator to split the schema graph into pieces, called *Target Schema Segments (TSS)*. It also uses presentation graphs with expansion links to present data with multi-value dependencies in a concise way. Précis [16] is a keyword search system for relational databases. It determines the schema of the output by requiring users or system administrator to specify a weight for each edge in the schema graph. Then each user further needs to specify *degree constraint* and *cardinality constraint* in the schema for the cut-off. [17] proposes a schema-free XQuery where the predicates in an XQuery are specified through the concept of MLCA, but return information is specified by users in the return clause of XQuery.

To the best of our knowledge, this paper is the first that addresses the problem of automatically inferring desirable return nodes for XML keyword search. Compared with existing approaches, XSeek has several advantages. First, it can automatically infer the most relevant return information without elicitation from users or system administrators. Second, the output information is determined by both the characteristics of XML data structure and the pattern of keyword matches. Third, it works well for data in the presence or in the absence of schema information.

**Ranking Search Results.** Ranking schemes have been studied for keyword search on XML documents. [15, 4] propose to rank query results according to the distance between different keyword matches in the original document. In the presence of XML schema, efficient algorithms to compute top  $k$  results are presented in [15]. XSEarch[8] employs a ranking scheme in the flavor of information retrieval, considering factors like distance, term frequency, document frequency, etc. XRank [12] extends the Page-rank hyperlink metric to XML. The ranking schemes can be orthogonal to retrieval and be incorporated into XSeek.

The problem of integrating keyword proximity search in a structured query language has been studied. We direct readers to [10, 11, 3] for details.

There is a lot of work supporting keyword search on a relational database, which is modeled as a graph of objects with connecting edges, DBXplorer [2], BANKS [5], DISCOVER [14], [18]. The focuses are how to efficiently compute (approximate) Steiner trees that contain all the keywords by building appropriate indexes, and how to rank query results effectively.

## 7. CONCLUSIONS

We present an XML keyword search engine XSeek that addresses an open problem of inferring desirable return nodes without elicitation of user preferences and has achieved promising results. We analyze both XML data structure and keyword match patterns, and generate meaningful return nodes accordingly. Data nodes that match search predicates and return nodes are output as query results with optional expansion links. Compared with existing approaches *Subtree Return* and *Path Return*, XSeek achieves improved precision and recall with reasonable cost and good scalability. In the future, we will investigate effective ranking schemes such that search results will be output in the order of their relevance.

## 8. ACKNOWLEDGMENTS

This research is partially supported by NSF IIS-0612273. We would like to thank Jeffrey Walker for his valuable comments.

## 9. REFERENCES

- [1] XQuery 1.0: An XML Query Language, June 2001. <http://www.w3.org/XML/Query>.
- [2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *Proceedings of ICDE*, pages 5–16, 2002.
- [3] S. Amer-Yahia, C. Botev, J. Dorre, and J. Shanmugasundaram. XQuery Full-Text extensions explained. In *IBM Systems Journal*, pages 335–352, 2006.
- [4] M. Barg and R. K. Wong. Structural Proximity Searching for Large Collections of Semi-Structured Data. In *Proceedings of CIKM*, pages 175–182, 2001.
- [5] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, 2002.
- [6] D. Carmel, Y. Maarek, Y. Mass, N. Efraty, and G. Landau. An Extension of the Vector Space Model for Querying XML Documents via XML Fragments. In *ACM SIGIR 2002: Workshop on XML and Information Retrieval*, 2002.
- [7] J. Clark and S. DeRose. XML Path Language (XPath) 1.0, November 1999. <http://www.w3.org/TR/xpath>.
- [8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML, 2003.
- [9] A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of SIGMOD*, pages 431–442, New York, NY, USA, 1999. ACM Press.
- [10] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):119–135, 2000.
- [11] N. Fuhr and K. GroBjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In *Proceedings of SIGIR*, pages 172–180, New York, NY, USA, 2001. ACM Press.
- [12] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of SIGMOD*, pages 16–27, 2003.
- [13] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Transactions on Knowledge and Data Engineering*, 18(4), 2006.
- [14] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *Procs. VLDB*, 2002.
- [15] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs, 2003. In *ICDE*.
- [16] G. Koutrika, A. Simitis, and Y. E. Ioannidis. Précis: The Essence of a Query Answer. In *ICDE*, page 69, 2006.
- [17] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.
- [18] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective Keyword Search in Relational Databases. In *Proceedings of SIGMOD*, pages 563–574, New York, NY, USA, 2006. ACM Press.
- [19] V. Vesper. Let's Do Dewey. <http://www.mtsu.edu/vvesper/dewey.html>.
- [20] Extensible Markup Language (XML) 1.0, 2004. <http://www.w3.org/TR/REC-xml/>.
- [21] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *Proceedings of SIGMOD*, pages 527–538, New York, NY, USA, 2005.
- [22] C. Yu and H. V. Jagadish. Schema Summarization. In *Proceedings of VLDB*, 2006.