Reasoning and Identifying Relevant Matches for XML Keyword Search [*]

Ziyang Liu   Yi Chen
Arizona State University
{ziyang.liu, yi}@asu.edu

ABSTRACT

Keyword search is a user-friendly mechanism for retrieving XML data in web and scientific applications. An intuitively compelling but vaguely defined goal is to identify matches to query keywords that are *relevant* to the user. However, it is hard to directly evaluate the relevance of query results due to the inherent ambiguity of search semantics. In this work, we investigate an axiomatic framework that includes two intuitive and non-trivial properties that an XML keyword search technique should ideally satisfy: *monotonicity* and *consistency*, with respect to data and query. This is the first work that reasons about keyword search strategies from a formal perspective.

Then we propose a novel semantics for identifying relevant matches, which, to the best of our knowledge, is the only existing algorithm that satisfies both properties. An efficient algorithm is designed for realizing this semantics. Extensive experimental studies have verified the intuition of the properties and shown the effectiveness of the proposed algorithm.

1. INTRODUCTION

Keyword search provides a simple and user-friendly query interface to access XML data in web and scientific applications, where users may not know XPath/XQuery, or the data schema is unavailable, complex, or fast-evolving. As a result, keyword search has recently attracted more and more research interests [5, 6, 21, 14, 21, 8, 13, 10].

To identify relevant results for an XML keyword query, different systems use different underlying principles and heuristics, leading to different query results in general. How to guide the design and evaluate XML keyword search strategies is becoming a critical research problem. However, due to the inherent ambiguity of search semantics, it is hard, if not impossible, to directly assess the relevance of query results and reason about various strategies.

Interestingly, we discover that by examining query results produced by the same XML keyword search strategy for similar queries or on similar documents, sometimes abnormal behaviors can be clearly observed, which exhibit the pitfalls that a good search strategy should avoid. Let us start with some examples.

[*]This material is based on work partially supported by NSF grant IIS-0740129 and IIS-0612273.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

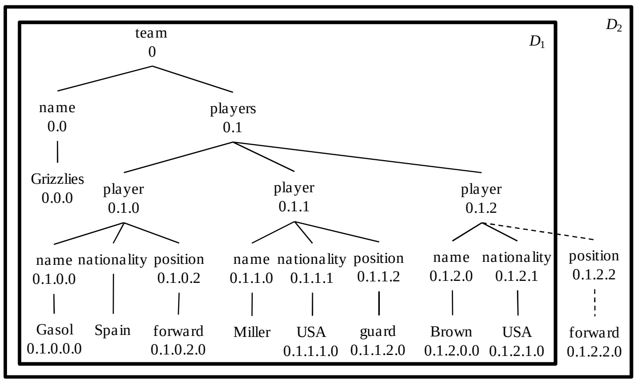


Figure 1: Sample XML Document

Q_1	Gasol, position
Q_2	Grizzlies, Gasol, position
Q_3	Grizzlies, Gasol, Brown, position
Q_4	forward, name
Q_5	forward, USA, name

Figure 2: Sample Keyword Searches

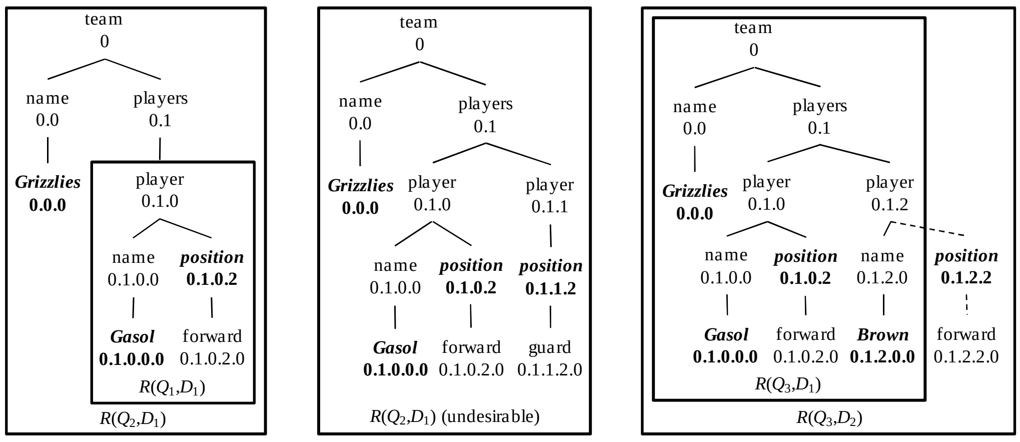
Example 1.1: Consider XML tree D_1 in Figure 1, where each node is associated with a unique ID (some IDs are not shown), and the keyword searches listed in Figure 2.

$\underline{[Q_1, D_1]}$:Processing query Q_1 (*Gasol, position*) which searches for the position of Gasol on D_1, a reasonable result is shown in the box annotated with $R(Q_1, D_1)$ in Figure 3(a), where the matches in the subtree rooted at the *player* node with ID 0.1.0 are included in the result, but not the match node *position* 0.1.1.2 of player Miller. Such a query result will be produced by many existing XML keyword search systems [21, 6, 14, 5, 13, 15].

$\underline{[Q_2, D_1]}$: To search for the position of Gasol within team Grizzlies, the user would issue Q_2 (*Grizzlies, Gasol, position*), containing one more keyword *Grizzlies* than Q_1. A query result as produced by [21, 6, 5, 15] is shown in Figure 3(b). One difference between these two results is: the *position* node 0.1.1.2 of player Miller is now included, which is unlikely to be justifiable.

A reasonable result would be the one shown in the box annotated with $R(Q_2, D_1)$ in Figure 3(a). ■

Example 1.2: $\underline{[Q_3, D_1]}$: Consider Q_3 (*Grizzlies, Gasol, Brown, position*) on XML tree D_1, searching for the position of player Gasol and Brown within Grizzlies. A reasonable query result is shown in the box annotated with $R(Q_3, D_1)$ in Figure 3(c), as pro-



(a) Results of Q_1 and Q_2 on D_1 (b) Undesirable Result of Q_2 on D_1 (c) Results of Q_3 on D_1 and D_2

Figure 3: Sample Query Results

duced by many existing XML keyword search systems [21, 6, 14, 5, 15].

$[Q_3, D_2]$: Suppose the *position* information of player *Brown* is now inserted to the XML tree D_1 as represented by the dotted line, which results in an XML tree D_2, shown in Figure 1. Now D_2 has position information for *both* players, as requested by Q_3.

An empty result, as produced by [14, 13, 5], is unlikely to be desirable. When *AND* semantics is considered, a keyword search is a *positive query*. A search strategy that outputs one query result for Q_3 but outputs nothing when a new data node is inserted is abnormal. A more reasonable query result is shown in the box annotated with $R(Q_3, D_2)$ in Figure 3(c). ■

From these examples we can observe that there should be some logical correlation of the query results generated by a desirable XML keyword search engine for two similar queries on the same document or the same query on two similar documents. Since abnormal behaviors can be more easily identified when checking query pairs or document pairs than considering a single query on a single document, we attempt to assess the quality of XML keyword search engines from a new angle: capturing desirable *changes* to a query result upon a *change* to the query or data in a general framework.

Indeed, the approach that formalizes broad intuitions as a collection of simple axioms and evaluates various solutions based on the axioms has been successfully used in many areas, such as mathematical economics [17],[1] clustering [11], discrete location theory [7], and collaborative filtering [18].

In light of the success of an axiomatic approach in those areas, we initiate an investigation of a formal axiomatic framework to express valid changes to a query result upon an addition to the user query or to the data.[2] This is, nevertheless, very challenging. As we can see, when a new keyword is added to a query or when a new data node is inserted, some keyword matches should become relevant and be added to the query result, such as *Grizzlies* 0.0.0 for $[Q_2, D_1]$ and *position* 0.1.2.2 for $[Q_3, D_2]$; but not all the matches, such as *position* 0.1.1.2 for $[Q_2, D_1]$. Some keyword matches should become irrelevant and be removed from the query result, such as the position of a player Gasol in a team other than Grizzlies (not shown in the figure); but not all the matches, such as the case in $[Q_3, D_2]$.

After an in-depth analysis of valid changes to query results, independently of any particular algorithm, we identify two intuitive and elegant properties that an XML keyword search algorithm should ideally possess: *data monotonicity*, *query monotonicity*, *data consistency* and *query consistency*. An algorithm that shows the abnormal behaviors illustrated in the above examples violates at least one of the properties, as will be analyzed in Example 2.2 through 2.6.[3]

After reviewing the existing strategies on XML keyword search, we find that, surprisingly, none of them satisfies all these properties. We then design a novel XML keyword search strategy, MaxMatch, which possesses all these properties and efficiently processes user queries.

The contributions and outline of this paper are:

1. To the best of our knowledge, this is the first work that reasons about keyword search strategies from a formal perspective, and proposes the desirable properties that an XML keyword search strategy should ideally satisfy.
2. Four properties, *data monotonicity*, *query monotonicity*, *data consistency* and *query consistency* are presented in Section 2, capturing the reasonable connection between an original query result and a new query result obtained after an update to the query or to the data. These properties are non-trivial, non-redundant, and satisfiable.
3. We have reasoned about existing XML keyword search approaches for identifying relevant matches with respect to these properties in Section 3.
4. We have designed and implemented a novel technique, MaxMatch, to identify relevant keyword matches (Section 4). To the best of our knowledge, this is the only algorithm that satisfies all properties.
5. Experimental evaluation have verified our intuition about the properties, and demonstrated the improved precision and recall of MaxMatch over existing approaches as well as its efficiency, as presented in Section 5.

We close by discussing related work in Section 6 and future directions in Section 7.

[1]A striking case is the axioms on social choice functions [3], which can not be simultaneously satisfied by any solution, proposed by Kenneth Arrow, a co-recipient of the 1972 Nobel Prize in Economics.

[2]The desirable behaviors of an algorithm are symmetric for deletions, whose details are omitted for space reasons.

[3]However, the proposed properties may not be complete, i.e. a system that satisfies these properties is not necessary a perfect system.

2. PROPERTIES FOR IDENTIFYING RELEVANT MATCHES

2.1 Data Model and Query

We model XML data as a rooted, labeled, unordered tree. Every internal node in the tree has a name, and every leaf node has a data value. XML attributes are treated as subelements, i.e., children of the associated element nodes. A user query is expressed as a set of keywords, each of which may match name and/or value nodes in the XML tree.

In this paper, we consider *AND* semantics in defining query results, similar as existing approaches [21, 14, 9, 8, 6]. Evaluating a keyword search on an XML tree returns a set of subtrees, each of which contains at least one *match* to each keyword in the query.

Definition 2.1:[Matches] If a keyword k is contained in the name or value of a node m, then m is a *match* to k. ■

Definition 2.2:[Query Results] Processing query Q on XML data D returns *a set of query results*, denoted as $R(Q, D)$. Each query result is a tree defined by a pair $r = (t, M)$, where t is the root, M is a subset of matches in the tree, consisting of all the matches in the tree that are considered as relevant to Q. Every keyword in Q has at least one match in M. A *query result* is a tree consisting of the paths in D that connect t to each match in M (as well as its value child, if any). The number of query results, $|R(Q, D)|$, is the number of (t, M) pairs. ■

Note that one query result should not be *subsumed* by another, therefore the root nodes t should not have ancestor-descendant relationship.

Example 2.1: $\underline{[Q_2, D_1]}$: Consider Q_2 (*Grizzlies, Gasol, position*) on D_1 in Figure 1, which searches for the position of Gasol who is a player in team Grizzlies. There is only one meaningful query result: *t= team*, M = { *Grizzlies*, *Gasol*, *position* (0.1.0.2) }. Note that match node *position* (0.1.1.2) in the subtree rooted at *team* should be considered as irrelevant and not be included in M, as it is the position of player *Miller*, not *Gasol*. The query result is a tree consisting of the paths from *team* to each node in M, including their value children, as shown in the box of $R(Q_2, D_1)$ in Figure 3(a). ■

In this paper we reason about identifying relevant matches to generate a query result. Instead of directly assessing the relevance of match nodes, we propose an axiomatic framework that characterizes the valid connections between the original query results and the new query results generated by the same algorithm when an update is performed on the query and/or the data. Specifically, property *monotonicity* captures reasonable changes to the number of query results (i.e., $|R(Q, D)|$); and *consistency* captures reasonable changes to the content of query results (i.e., M sets in $R(Q, D)$).

Note that the our approach is independent of any particular algorithm, therefore in this section we present some meaningful query results to illustrate the proposed properties, without considering *how to design an algorithm* that generates those results.

2.2 Monotonicity

Monotonicity describes the desirable change to the number of query results with respect to data updates and query updates.

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. Similarly, for keyword search on XML trees, adding a node to the data may enable an XML subtree that is not originally a query result to be a new query result. Let us look at an example before defining data monotonicity formally.

Example 2.2: Adding a new data node may increase the number of query results. $\underline{[Q_4, D_1]}$: Consider Q_4 (*forward, name*) on XML data D_1 shown in Figure 1, which searches for the name of a forward. Ideally, there should be one query result, rooted at *player* (0.1.0) with the matches in its subtree: *name* (0.1.0.0) and *forward* (0.1.0.2.0), and the paths connecting them. $\underline{[Q_4, D_2]}$: Now consider an insertion of a *position* node (0.1.2.2) and its value *forward* (0.1.2.2.0) to D_1, which results in XML tree D_2. Ideally, we should have one more query result: a tree rooted at *player* (0.1.2), the matches in its subtree *name* (0.1.2.0) and *forward* (0.1.2.2.0), and the paths connecting them.

The number of query results may also stay the same after a data insertion. Consider Q_3 (*Grizzlies, Gasol, Brown, position*), searching for the position of Gasol and Brown in Grizzlies, on D_1 and D_2, respectively. For each document there should be a single query result tree, rooted at *team*, as this subtree contains at least one relevant match to each keyword. Though the set of relevant matches in the subtree rooted at the *team* node are different for D_1 and D_2 as shown in Figure 3(c),[4] yet the number of query result is one for both XML documents. On the other hand, if $R(Q_3, D_2)$ has an empty set of query results as discussed in Example 1.2, it violates data monotonicity. This is undesirable as the positions of both Gasol and Brown in Grizzlies indeed present in D_2. ■

Definition 2.3:[Data Monotonicity] An algorithm satisfies *data monotonicity* if for a query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$, the number of query results on document D' is no less than that on D, i.e., $|R(Q, D)| \leq |R(Q, D')|$. ■

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. Similarly, for XML keyword search, adding a new keyword can disqualify a query result of the original query if it is far away from any match to the new keyword.

Example 2.3: Adding a new keyword may decrease the number of query results. $\underline{[Q_4, D_2]}$: Recall that there are two query results when processing Q_4 on the XML data D_2, rooted at *player* (0.1.0) and *player* (0.1.2) respectively. $\underline{[Q_5, D_2]}$: Now suppose we add one more keyword *USA* to Q_4, which results in Q_5 in Figure 2, searching for the name related to forward and USA. The query result rooted at node *player* (0.1.2) is still a relevant query result. However, the one rooted at *player* (0.1.0) becomes invalid, as it does not contain any match to *USA* in its subtree. To satisfy the *AND* semantics of the query, one would think of replacing the query result rooted at *player* (0.1.0) with the one rooted at *players* (0.1) which contains at least one match to each keyword in its subtree. However, since match node *USA* (0.1.2.1.0) belongs to a different player than match nodes *forward* (0.1.0.2.0) and *name* (0.1.0.0), they are unlikely to be meaningfully related to define a relevant query result. Therefore, the number of query results of Q_5 is reduced to one.

The number of query results may stay the same after a query insertion. Consider Q_1 and Q_2 on D_1. The query result $R(Q_1, D_1)$

[4]We discuss the desirable changes of relevant matches in consistency property.

would be *player* (0.1.0) along with the match nodes in the subtree; the query result $R(Q_2, D_1)$ would be *team* (0) along with the match nodes *Grizzlies*, *Gasol*, and *position* (0.1.0.2), as shown in Figure 3(a). Though the results of Q_1 and Q_2 are different, both queries have the same number of query result: one. ■

Definition 2.4:[Query Monotonicity] An algorithm satisfies *query monotonicity* if for two queries Q and Q' and an XML document D, $Q' = Q \cup \{k\}$, where k is a keyword, $k \notin Q$, the number of query results of Q' is no more than that of Q, i.e., $|R(Q, D)| \geq |R(Q', D)|$. ■

2.3 Consistency

Monotonicity describes how the number of query results should change upon an update to the data or query. Consistency describes how the content of query results should change upon an update to the data or query. Intuitively, the delta of two sets of query results can be defined as the biggest subtrees that are in one set of query results but not in the other, named as *delta result trees*.

Example 2.4: Consider $R(Q_3, D_1)$ and $R(Q_3, D_2)$ in Figure 3(c). The subtree rooted at *position* (0.1.2.2) is a biggest subtree that is in $R(Q_3, D_2)$ but not in $R(Q_3, D_1)$, i.e., a delta result tree. Indeed, every node in this subtree is in $R(Q_3, D_2)$, and none of them is in $R(Q_3, D_1)$. On the other hand, the subtree rooted at its parent node *player* (0.1.2) is not a delta result tree, as some nodes, e.g. *player* (0.1.2), *name* (0.1.2.0) are in $R(Q_3, D_1)$. ■

Now we formally define a *delta result tree* in XML keyword search as the subtree that newly becomes part of the set of query results upon an insertion to the data or query. Note that a delta result tree could be a query result itself, or could be part of a query result.

Definition 2.5:[Delta Result Tree (δ)] Let R be a set of query results of processing query Q on data D, and R' be the set of updated query results after an insertion to Q or D. A subtree rooted at a node n in a query result tree $r' \in R'$ is a *delta result tree* if $desc\text{-}or\text{-}self(n, r') \cap R = \emptyset$ and $desc\text{-}or\text{-}self(parent(n, r'), r') \cap R \neq \emptyset$, where $parent(n, r')$ and $desc\text{-}or\text{-}self(n, r')$ denotes the parent, and the set of descendant-or-self nodes of node n in a tree r', respectively. The set of all delta result trees is denoted as $\delta(R, R')$. ■

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. Similarly, for keyword search on XML trees, after we add a new node to the XML data, if there exists a delta result tree in the new query result, then this delta result tree should contain the newly inserted node to be qualified (because otherwise, this sub-tree should not be part of the new query result in order to be consistent with the original query result). Let us look an example before defining data consistency formally.

Example 2.5: $\underline{[Q_4, D_1]}$: Consider Q_4 (*forward, name*) on XML data D_1 shown in Figure 1, which searches for the name of a forward. There is one query result, consisting of *player* (0.1.0) and the matches in its subtree. $\underline{[Q_4, D_2]}$: Now consider D_2 obtained after an insertion of a *position* (0.1.2.2) node along with its value *forward* to D_1. This insertion qualifies a new query result in $R(Q_4, D_2)$, consisting of *player* (0.1.2) and the matches in its subtree. This new query result is a delta result tree, as it is the biggest subtree that is in $R(Q_4, D_2)$, but not in $R(Q_4, D_1)$. It is valid with respect to data consistency since the delta result tree contains the newly inserted match node *forward* (0.1.2.2.0).

Consider Q_1 (*Gasol, position*) on D_1 and D_2, searching for the position of Gasol. Although the newly inserted node *position* (0.1.2.2) is a match of Q_1, the query result should not change. Intuitively, this match refers to the *position* of a player other than *Gasol*, and therefore is irrelevant. In this case, there does not exist a delta result tree, and data consistency holds trivially.

It is easy to verify that the changes to the query results in the above examples also satisfy data monotonicity. ■

Definition 2.6:[Data Consistency] An algorithm satisfies *data consistency* if for query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$, if $\delta(R(Q, D), R(Q, D'))$ is not empty, then every delta result tree contains n (so there can only be one delta result tree). ■

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 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. Similarly, for keyword search on XML trees, after we add a new keyword to the query, if there exists a delta result tree in the new query result, then this delta result tree must contain at least one match to the new keyword (because otherwise, this sub-tree should not be part of the new query result in order to be consistent with the original query result).

Example 2.6: $\underline{[Q_4, D_2]}$: Consider again Q_4 on XML data D_2 in Figure 1. We have two query results: *player* (0.1.0) and the matches in its subtree; *player* (0.1.2) and the matches in its subtrees. $\underline{[Q_5, D_2]}$: If we add one more keyword *USA* to Q_4, which results in Q_5, then *player* (0.1.0) should no longer be a query result (which satisfies query monotonicity). On the other hand, the query result related to *player* (0.1.2) should add the subtree rooted at *nationality* (0.1.2.1), which is a delta result tree. This is valid with respect to query consistency since this subtree contains a node 0.1.2.1.0 matching the new keyword *USA*.

Now consider Q_1 and Q_2 on D_1, where Q_2 has one new keyword *Grizzlies* compared with Q_1, whose query results are shown in Figure 3(a). These are valid with respect to query consistency since the delta result tree rooted at *name* (0.0) contains a match (0.0.0) to the new keyword *Grizzlies*. On the other hand, if $R(Q_2, D_1)$ is the one shown in Figure 3(b) as discussed in Example 1.1, then there is another delta result tree compared to $R(Q_1, D_1)$ in Figure 3(a): the subtree rooted at *player* (0.1.1). However, since this delta result tree does not contain any match to the new keyword *Grizzlies*, it violates query consistency. This query result is indeed undesirable as *position* (0.1.1.2) is irrelevant to player Gasol. ■

Definition 2.7:[Query Consistency] An algorithm satisfies *query consistency* if for two queries Q and Q' and an XML document D, $Q' = Q \cup \{k\}$, where k is a keyword, $k \notin Q$, if $\delta(R(Q, D), R(Q', D))$ is not empty, then every delta result tree contains at least one match to k. ■

Monotonicity and consistency properties with respect to data and queries are non-trivial, non-redundant, and satisfiable. They are not trivial, as to the best of our knowledge, there is no existing XML keyword algorithm that satisfies all of them. They are not redundant since we can find algorithms that satisfy one property but fail another. Detailed analyses will be presented in Section 3. Furthermore, we show that these properties are satisfiable by proposing a keyword search semantics that satisfies all of them in Section 4.

3. ANALYZING EXISTING ALGORITHMS

Several approaches have been proposed for identifying relevant matches for XML keyword search, including XKSearch [21], XRank [6], XSEarch [5], Compact Valuable Lowest Common Ancestor (CVLCA) [13] and Meaningfully Lowest Common Ancestor (MLCA) [14]. In this section, we review and analyze these approaches in terms of monotonicity and consistency with respect to data and query.

XKSearch [21]. XKSearch proposes a concept of *Smallest Lowest Common Ancestor* (*SLCA*). For a query Q on data D, an XML node is an SLCA if it contains matches to all keywords in Q in its subtree, and none of its descendants does. For each SLCA, all its descendant matches are considered as relevant to Q.

However, not all such matches are necessarily relevant. For example, consider Q_2 (*Grizzlies, position, Gasol*) on D_1, where the SLCA node is *team*. Although node *position* (0.1.1.2) is a match in the subtree rooted at the SLCA node, it is irrelevant to the query as it is not the *position* of *Gasol*. This undesirable behavior can be detected by analyzing consistency property.

XKSearch does not satisfy query consistency. Consider Q_1 and Q_2 on D_1. The query results produced by XKSearch are shown in the box annotated with $R(Q_1, D_1)$ in Figure 3(a), and $R(Q_2, D_1)$ in Figure 3(b). As we can see, the subtree rooted at *player* (0.1.1) is a delta result tree in $\delta(R(Q_1, D_1), R(Q_2, D_1))$. However, it doesn't contain matches to the new keyword *Grizzlies*, and thus violates query consistency.

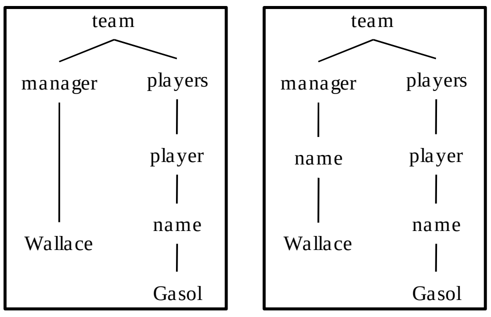
XRank [6]. In XRank, an XML node is the root of a query result if it contains at least one occurrence of each keyword in its subtree, after excluding the occurrences of the keywords in its descendants that already contain all the keywords. All descendant matches of such nodes are considered relevant.

XRank does not satisfy query consistency. For Q_1 and Q_2 on D_1, XRank produces the same results as XKSearch.

The following approaches, XSEarch, CVLCA, and MLCA, use a group of matches containing exactly one match to each keyword, referred as *pattern match*, to identify relevant matches. For a query Q on data D, these approaches find qualified pattern matches according to their specific metrics. In our analyses, we consider a match in a qualified pattern match as a relevant match. For example, consider Q_2 on D_1 in Figure 1, there are two pattern matches, $\{$*Grizzlies*, *Gasol*, *position* (0.1.0.2)$\}$, and $\{$*Grizzlies*, *Gasol*, *position* (0.1.1.2)$\}$. Suppose the first pattern match is identified to be qualified, but not the second. Then the matches in the first pattern match are considered as relevant.

XSEarch [5]. XSEarch defines more expressive search terms than keywords. There are three types of search terms. A node n satisfies term $l : k$ if n's name contains l and n has a descendant leaf node whose value contains k; a node n satisfies term l : if n's name contains l; a node n satisfies term : k if it has a child leaf node whose value contains k. In the analyses, we consider each word in a search term as a query keyword, and a node that satisfies a search term as a match.

To identify relevant matches, XSEarch defines *interconnection* relationship among two matches, and uses two semantics, namely *all-pair* semantics and *star* semantics, to identify relevant pattern matches. Two matches n and n' are interconnected if the shortest path between n and n' (through LCA(n, n')) does not have two distinct nodes with the same name, except n and n'. All-pair semantics considers a pattern match P to query Q on data D as qualified if any two nodes in P are interconnected. Star semantics considers a pattern match P as qualified if there is a node in P such that every other node in P is interconnected with it. As we can see, for a query containing two search terms, all-pairs and star semantics are equivalent.



(a) D_3 (b) D_4

Figure 4: D_3 **and** D_4

XSEarch may fail to recognize relevant matches. Consider query (*:Gasol, :Brown*) on D_1. The nodes that satisfy the search terms are *name* (0.1.0.0) and *name* (0.1.2.0), composing a pattern match. However, this pattern match is not qualified since the two match nodes are not interconnected: there are two distinct nodes (0.1.0, 0.1.2) with the same name *player* on the shortest path connecting them. Thus XSEarch gives an empty result for this query. This is unlikely to be desirable as the user may be interested in finding the relationship among two persons whose names are specified in the query. Such a behavior can be captured by analyzing query monotonicity.

XSEarch, either all-pair semantics or star semantics, does not satisfy query monotonicity. For query (*:Gasol, :Brown*) on D_1, XSEarch has an empty query result, as discussed above. Now consider query (*player:Gasol, player:Brown*), which has one more keyword than the previous query. There is one qualified pattern match: $\{$*player* (0.1.0), *player* (0.1.2)$\}$, which constitutes a query result. After adding a keyword *player*, the number of query results produced by XSEarch increases, thus it violates query monotonicity.

XSEarch star semantics does not satisfy query consistency. Consider query (*:Gasol, position:*) (corresponding to Q_1) on D_1. *position* (0.1.1.2) is not considered as a relevant match, as it is not interconnected with node *name* (0.1.0.0) that matches term *:Gasol*. Now if we add one more search term to form a new query (*:Grizzlies, :Gasol, position:*) (corresponding Q_2), then *position* (0.1.1.2) is considered as relevant. This is because there is a qualified pattern match $\{$*name* (0.0), *name* (0.1.0.0), *position* (0.1.1.2)$\}$, where *name* (0.0) is interconnected with the other two nodes. Note that the relevant matches identified by XSEarch for this query is the same as those identified by XKSearch and XRank. As discussed, such behavior is undesirable and violates query consistency.

Compact Valuable LCA (CVLCA) [13]. This approach proposes the concept of *Compact Valuable LCA*. For a keyword query Q on data D, a node u is considered a valuable LCA (VLCA) if there is a pattern match P that satisfies XSEarch all-pair semantics, and u is the LCA of the nodes in P. A node u is a CVLCA if it is a VLCA of pattern match P, and dominates every node in P. u dominates a node v in P if for any other pattern match P' that contains v, the LCA of nodes in P' is an ancestor-or-self of u.

CVLCA does not satisfy data monotonicity. Consider query (*Wallace, Gasol*) on D_3 in Figure 4. Pattern match (*Wallace, Gasol*) is qualified as the two nodes are interconnected. Now we insert a *name* node between nodes *manager* and *Wallace*, resulting in the XML tree D_4. Nodes *Wallace* and *Gasol* are no longer interconnected, thus the query result on D_4 is empty. Since this data insertion results in fewer query results, it violates data monotonicity. Such a behavior is indeed unintuitive. If *Wallace* and *Gasol* are considered to be relevant to each other in D_3, adding the description that *Wallace* is a *name* in D_4 should not disqualify their relevance.

Meaningfully LCA (MLCA) [14]. The concept of MLCA is proposed as part of Schema-free XQuery which allows users to query XML with arbitrary knowledge of the underlying schema [5]. MLCA can be used to identify qualified pattern matches and thus relevant matches in XML keyword search. Two XML nodes n_1 and n_2 that match keywords k_1 and k_2 are meaningfully related if there does not exist n_1' ad n_2' that match k_1 and k_2, such that LCA(n_1, n_2) is an ancestor of LCA(n_1', n_2'). For a keyword search Q on data D, a pattern match P is qualified if every two nodes in P are meaningfully related. If P qualifies, the LCA of all nodes in P is defined as an MLCA of Q on D, and the matches in a qualified pattern match are considered relevant.

MLCA does not satisfy data monotonicity. Consider Q_3 on D_1 and D_2. The query result produced by MLCA on D_1 is shown in the box annotated with $R(Q_3, D_1)$ in Figure 3(c). The query result $R(Q_3, D_2)$ produced by MLCA is empty, since we are not able to find a pattern match in which every pair of nodes is meaningfully related. No matter which *position* match we choose in a pattern match, it is not related to at least one other node. For instance, if we choose *position* (0.1.0.2), it is not related to *Brown* (0.1.2.0.0). This is because they have an LCA node *players* (0.1). However, the LCA node of *position* (0.1.2.2) and *Brown* (0.1.2.0.0) is *player* (0.1.2), which is a descendant of *players* (0.1). Similarly, we cannot choose the other two *position* matches to compose a qualified pattern match. Thus MLCA violates data monotonicity as the number of query results decreases when we add a new node to the data. Such a behavior is not desirable. Q_3 is likely to search the *position* of both *Gasol* and *Brown* in team *Grizzlies*. Since this information is present in the data, the query result should not be empty.

In summary, none of the existing approaches for identifying relevant matches in XML keyword search satisfies all four properties.

4. MAXMATCH

In this section, we show that the properties proposed in Section 2 are satisfiable by presenting MaxMatch, an effective and efficient XML keyword search technique. We first introduce the semantics of MaxMatch for identifying relevant matches, then propose an efficient algorithm to achieve it.

4.1 Definitions

Recall that a query result tree is defined by a pair, $r = (t, M)$, where t is the root, M is the set of matches in the tree that are considered as relevant to Q, and every keyword in Q has at least one match in M (Definition 2.2). We adopt a commonly used approach in the literature [14, 21, 8], namely *SLCA*, to identify t, as to be reviewed in this section. We address the challenge of identifying relevant matches M within t in Section 4.2 and 4.4.

The intuition of SLCA is that only the matches in a *smallest subtree* in the XML data that contains matches to every keyword in a query are possibly relevant. A tree rooted at node n_1 is smaller than the one rooted at node n_2 if n_1 is a descendant of n_2. Let us look at an example.

Example 4.1: $\underline{[Q_1, D_1]}$:. For Q_1 (*Gasol, position*), node *position* (0.1.0.2) and *Gasol* (0.1.0.0.0) are relevant matches since they refer to the same player. Indeed they are in the subtree rooted at *player* (0.1.0), which is a smallest subtree that contains matches to both keywords. On the other hand, irrelevant match *position* (0.1.1.2) can be detected since the subtree that contains this match and a match to *Gasol* is rooted at *players* (0.1), which is not a smallest subtree that contains matches to both keywords. ■

As can be seen from the example, matches that are not in a smallest subtree that contains matches to all keywords are unlikely to be relevant, such as *position* (0.1.1.2). Choosing such smallest subtrees can prune some irrelevant matches.[6]

Given the intuition of such smallest subtrees, now we formally define *Descendant Matches* and *SLCA* [21].

Definition 4.1:[Descendant Matches] For query Q on XML data D, the *descendant matches* of a node $n \in D$, denoted by $dMatch(n)$, is a set of keywords in Q, each of which has at least one match in the subtree rooted at n. ■

Definition 4.2:[SLCA] A set of *smallest lowest common ancestor (SLCA)* of matches to Q on D, denoted by $SLCA(Q, D)$, consists of nodes $t \in D$ that satisfy the following: (i) the set of descendant matches of t is Q, i.e., $dMatch(t) = Q$, and (ii) there does not exist a node t' which is a descendant of t, such that $dMatch(t') = Q$. ■

Example 4.2: $\underline{[Q_1, D_1]}$: Continuing the previous example, there are three nodes in D_1 that contain matches to both keywords in Q_1 in their subtrees, $dMatch(0.1.0) = dMatch(0.1) = dMatch(0) = Q$. Therefore we have $SLCA(Q_1, D_1)$={0.1.0}. ■

In MaxMatch, a query result tree is identified as $r = (t, M)$, where $t \in SLCA(Q, D)$ is the root. Next we discuss how to select relevant matches M in the subtree rooted at t, which satisfies both monotonicity and consistency.

4.2 Semantics of Selecting Relevant Matches

Not all matches in the subtrees rooted at SLCA nodes are relevant. Recall $\underline{[Q_2, D_1]}$: in Example 2.1, where match node *position* (0.1.1.2) in the subtree rooted at the SLCA node *team* (0) in D_1 is irrelevant to Q_2 (*Grizzlies, Gasol, position*) since it corresponds to player *Miller*.

To identify irrelevant matches, we observe that not every descendant of an SLCA node is equally important in contributing to query results. A node in the subtree rooted at an SLCA node may provide strictly less information than its sibling nodes. Continuing the example $\underline{[Q_2, D_1]}$, *player* (0.1.1) provides strictly less information than its sibling node *player* (0.1.0), since the set of descendant matches of *player* (0.1.1) ({*position*}), is a proper subset of that of *player* (0.1.0) ({*Gasol*, *position*}). Therefore *player* (0.1.1) is considered to be inferior than *player* (0.1.0) and the matches in its subtree are considered as irrelevant.

Definition 4.3:[Contributor] For an XML tree D and a query Q, a node n in D is a *contributor* to Q if (i) n has an ancestor-or-self n_1 in the SLCA set, $n_1 \in SLCA(D, Q)$, and (ii) n does not have a sibling n_2, such that $dMatch(n_2) \supset dMatch(n)$, where $dMatch(n)$ is the set of descendant matches of node n (Definition 4.1). ■

By Definition 4.3, an SLCA node is a contributor. Furthermore, there is at least one contributor among sibling nodes. Now we define relevant matches based on contributors.

[5]Note that Schema-free XQuery [14] is an XQuery like engine, not a keyword search engine. We are not directly using and comparing with the query engine, but instead applying the MLCA semantics to keyword search.

[6]Additional irrelevant matches need to be pruned as will be discussed in Section 4.2 and 4.4.

Definition 4.4:[Relevant Match] For an XML tree D and a query Q, a match node m in D is *relevant* to Q if (i) m has an ancestor-or-self n, $n \in SLCA(D, Q)$, and (ii) every node on the path from n to m is a contributor to Q. ■

In the above example $\underline{[Q_2, D_1]}$:, every node on the path from SLCA *team* to *Gasol* is a contributor, therefore *Gasol* is a relevant match. Similarly, *Grizzlies* and *position* (0.1.0.2) are relevant matches.

Note that we consider the partial order among sibling nodes induced by the $\subset$ relationship of the sets of their descendant matches, therefore a contributor is a *maximal* node among its siblings. A match is considered to be relevant if its ancestors are all maximal nodes among the siblings.

Proposition 4.3: (t, M) qualifies to be a query result according to Definition 2.2, where node $t \in SLCA(Q, D)$, M is the set of relevant matches in the subtree rooted at t.

PROOF. We need to prove that M contains at least one match to every keyword in Q. Let M' be the set of all the match nodes in the subtree rooted at t in D. According to the definition of SLCA (Definition 4.2), M' contains at least one match to each keyword in Q. Removing the irrelevant matches from M' results in set M. If an irrelevant match m_1 to keyword k is pruned, then it must have an ancestor-or-self n_1 that is not a contributor. That is, n_1 has a sibling contributor node n_2, $dMatch(n_1) \subset dMatch(n_2)$. Therefore $k \in dMatch(n_2)$. By induction, there still exists at least one match to keyword k in the subtree of t, for every keyword $k \in Q$, and finally M contains at least one match to each keyword. ■

Definition 4.5:[Query Results of MaxMatch] For an XML tree D and query Q, each query result generated by MaxMatch is defined by $r = (t, M)$ for every $t \in SLCA(Q, D)$, where M is the set of *relevant matches* to keywords in Q in the subtree rooted at t. A query result tree r consists of the paths from t to each relevant match in M. ■

4.3 Property Analysis

We prove that MaxMatch satisfies both monotonicity and consistency with respect to data and query.

Proposition 4.4: MaxMatch satisfies data monotonicity.

PROOF. Consider query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$. According to Definition 4.5, the number of query results generated by MaxMatch is equal to the number of SLCA nodes. For any $t \in SLCA(Q, D)$, there are two possibilities.

- $t \in SLCA(Q, D')$.
- $t \notin SLCA(Q, D')$. We still have $dMatch(t) = Q$, but t is no longer the root of a smallest subtree that contains matches to all keywords. Then there must exist at least one descendant of t that qualifies to be in $SLCA(Q, D')$.

Therefore, we have $|SLCA(Q, D')| \geq |SLCA(Q, D)|$, and $|R(Q, D')| \geq |R(Q, D)|$. ■

Proposition 4.5: MaxMatch satisfies query monotonicity.

PROOF. Consider two queries Q and Q' and an XML document D, $Q' = Q \cup \{k\}$, where k is a keyword, $k \notin Q$. For any $t \in SLCA(Q, D)$, there are two possibilities:

- $t \in SLCA(Q', D)$
- $t \notin SLCA(Q', D)$. This is because t no longer contains matches to all keywords in its subtree, i.e. $k \notin dMatch(t)$, $dMatch(t) \neq Q'$. It is not possible for a descendant of t to be in $SLCA(Q', D)$. At most one ancestor of t can be in $SLCA(Q', D)$, since SLCA nodes do not have ancestor-descendant relationship by Definition 4.2.

Therefore, we have $|SLCA(Q', D)| \leq |SLCA(Q, D)|$, and $|R(Q', D)| \leq |R(Q, D)|$. ■

Proposition 4.6: MaxMatch satisfies data consistency.

PROOF. Consider query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$. If $\delta(R(D, Q), R(D', Q))$ is empty, then MaxMatch trivially satisfies this property.

If $\delta(R(D, Q), R(D', Q))$ is not empty, let n_1 be the root of a delta result tree in $\delta(R(D, Q), R(D', Q))$, and n_2 be the parent of n_1. By Definition 2.5 and 4.3, n_2 is a contributor of both (D, Q) and (D', Q); n_1 is a contributor of (D', Q), but not a contributor of (D, Q) (because otherwise, since all ancestors of n_1 up to the SLCA node are contributors, we must have $n_1 \in R(D, Q)$). Therefore, there must exist a node n_3 which is a sibling of n_1, such that $dMatch(n_1) \subset dMatch(n_3)$ holds for D, but not for D'. This shows that the delta result tree rooted at n_1 must contain the newly inserted node n (which must match a keyword in Q). ■

Proposition 4.7: MaxMatch satisfies query consistency.

PROOF. Consider two queries Q and Q' and an XML document D, $Q' = Q \cup \{k\}$, where k is a keyword, $k \notin Q$. The proof is similar to that of Proposition 4.6. If $\delta(R(D, Q), R(D, Q'))$ does not exist, then MaxMatch trivially satisfies the property. Otherwise, for the root n_1 of each delta result tree in $\delta(R(D, Q), R(D, Q'))$, n_1 must contain at least one match to the new keyword k in its subtree. ■

4.4 Algorithm

The algorithm of realizing the semantics of MaxMatch is presented in Figure 5. There are four stages in the process. First, we retrieve the matches to each keyword in the query using procedure $findMatch$. Then we compute the set of SLCA nodes from the matches using procedure $findSLCA$. We group all keyword matches according to their SLCA ancestors using procedure $groupMatches$. Each group $group[i] = \{t, M\}$ consists of an SLCA node t, and the set of matches M that are descendants of t. Finally, the $pruneMatches$ procedure identifies and outputs contributors and relevant matches in M.

Next we illustrate each stage of the algorithm using Q_3 (*Grizzlies, Gasol, Brown, position*) on D_2 in Figure 1 as a running example, which searches for the position of player Gasol and Brown in team Grizzlies.

Matching Keywords. For a set of input keywords $keyword[w]$, procedure $findMatch$ retrieves the list of data nodes sorted in the order of their ID, $kwMatch[j]$, that match keyword $keyword[j]$, $1 \leq j \leq w$. To enable efficient retrieval, an inverted index is built from a word to the XML name or value nodes that contain this word.

Example 4.8: We start with retrieving the list of nodes matching each keyword in Q_3: *Grizzlies*, *Gasol*, *Brown*, *position* (0.1.0.2, 0.1.1.2, 0.1.2.2), respectively. ■

Computing SLCA. The procedure $findSLCA$ computes the SLCA nodes from $kwMatch$ according to the algorithm proposed in [21]. To facilitate the computation, each XML node is assigned a *Dewey*

```
MAXMATCH($keyword[w]$)
$kwMatch \leftarrow findMatch(keyword)$
$SLCA \leftarrow findSLCA(kwMatch)$ {adopted from [21]}
$group \leftarrow groupMatches(kwMatch, SLCA)$
for all $group[j]$ do
    $pruneMatches(group[j])$

GROUPMATCHES($kwMatch[w], SLCA[u]$)
{merge $kwMatch$ in Dewey ID order to a list $match$ in DeweyID order}
$match[v] \leftarrow merge(kwMatch[0], ..., kwMatch[w-1])$
{group matches}
$i \leftarrow 0, j \leftarrow 0$
while $(i \neq u)$ $or$ $(j \neq v)$ do
    $group[i].t \leftarrow SLCA[i]$
    if $isAncestor(group[i].t, match[j])$ then
        $group[i].M = group[i].M \cup match[j]$
        $j \leftarrow j+1$
    else if $group[i].M \neq \emptyset$ then
        $i \leftarrow i+1$
    else
        $j \leftarrow j+1$

PRUNEMATCHES($group = (t, M)$)
{identify relevant matches in $M$ and output query result trees}
$i \leftarrow M.size$
$start \leftarrow t$
{set $dMatch$ and $dMatchSet$ during a post-order tree traversal}
while $i \geq 0$ do
    for each node $n$ on the path from $M[i]$(exclusively) to $start$ do
        if $n$ matches $keyword[j]$ then
            set the $j^{th}$ bit of $n.dMatch$ to 1
        $n_p \leftarrow n.parent, n_c \leftarrow n.child$ on this path
        if $n_c \neq Null$ then
            $n.dMatch \leftarrow n.dMatch\ OR\ n_c.dMatch$
        $n.last \leftarrow i$ {record the last descendant match of $n$}
        $n_p.dMatchSet[num(n.dMatch)] \leftarrow true$ {$num$ is the
        function converting a binary number to a decimal number}
    $i \leftarrow i-1$
    $start \leftarrow LCA(M[i], M[i+1])$
{print query result during a pre-order tree traversal}
$i \leftarrow 0$
$start \leftarrow t$
while $i \leq M.size$ do
    for each node $n$ from $start$ to $M[i]$ do
        if $isContributor(n) = false$ then
            $i \leftarrow n.last + 1$ {skip the matches in the subtree rooted
            at $n$}
            $break$
        else
            output $n$
    $i \leftarrow i+1$
    $start \leftarrow LCA(M[i-1], M[i])$

ISCONTRIBUTOR($n$)
{return $true$ if $n$ is a contributor, otherwise $false$}
$n_p \leftarrow n.parent$
$i \leftarrow num(n.dMatch)$
for $j \leftarrow i+1$ to $2^w - 1$ do
    if $n_p.dMatchSet[j] = true$ && $AND(i,j) = i$ then
        return $false$
return $true$
```

Figure 5: Algorithm of MaxMatch

label [20] 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, the *player* node with Dewey ID 0.1.2 is the 3rd child of its parent node 0.1. Dewey ID can be used to determine the *lowest common ancestor (LCA)* of two XML nodes: the Dewey ID of nodes n_1 and n_2's LCA is equal to the longest common prefix of Dewey(n_1) and Dewey(n_2).

To efficiently retrieve the information of a node by its Dewey ID, we build a Dewey index using B-tree structure, clustered by Dewey ID.

Example 4.9: In our example, the SLCA node is *team* (0), as this is a lowest (in fact, the only) node that contains matches to all the keywords. ■

Grouping Matches. Then the $groupMatches$ procedure groups keyword matches $kwMatch$, such that the matches in each group are descendants of the same SLCA node.

First we merge $kwMatch[j]$ ordered by Dewey ID, $1 \leq j \leq w$, to produce a $match$ list ordered by Dewey ID. Then we build groups according to $match$ and the SLCA nodes, such that for each $t \in SLCA$, we have $group[i] = (t, M)$, M is the set of matches in the subtree rooted at t. According to Definition 4.2, $M \neq \emptyset$. Furthermore, SLCA nodes do not have ancestor-descendant relationship, a match can have at most one SLCA ancestor, and therefore belong to at most one group. Due to these two properties and the fact that $SLCA$ and $match$ are sorted by Dewey ID, the grouping can be achieved by a single traversal of $SLCA$ and $match$, during which matches that do not belong to any group (i.e. do not have an $SLCA$ ancestor) are discarded. We set the cursors to the beginning of $SLCA$, $group$, and $match$ ($i = 0$, $j = 0$). For each $SLCA[i]$, we set $group[i].t = SLCA[i]$. If the current node $match[j]$ is a descendant of $group[i].t$, then it is added into $group[i].M$. Otherwise, if $group[i].M$ is not empty, then $match[j]$ is either a descendant of $SLCA[i+1]$ or does not have an SLCA ancestor. In other words, we have already processed all the descendant matches of $SLCA[i]$, and thus move the cursor of $SLCA$ and $group$ (i.e. $i = i+1$). If $group[i].M$ is empty, then $match[j]$ does not have an SLCA node as its ancestor and does not belong to any group, therefore it is discarded.

Example 4.10: Continuing our running example, we merge four lists of $kwMatch$ and produce $match$: *Grizzlies* (0.0.0), *Gasol* (0.1.0.0.0), *position* (0.1.0.2), *position* (0.1.1.2), *Brown* (0.1.2.0.0), *position* (0.1.2.2). We group $match$ based on the SLCA nodes. In this example, there is only one SLCA node: *team* (0), therefore we have $group[0]$, where $group[0].t$ is the SLCA node, and $group[0].M$ is equal to $match$. ■

Pruning Matches. Each $group = (t, M)$ defines a tree T_g composed of the nodes on the paths from t to each match in M. Procedure $pruneMatches$ identifies and outputs the *contributors* and *relevant matches* in T_g as query results.

According to Definition 4.3, a node n is a contributor if n does not have any sibling whose descendant match set is a proper superset of that of n. We use a boolean array $n.dMatch$ of size w to record the set of descendant matches of node n with respect to query $keyword[w]$. This array is represented as a binary number that has 1 at position j if and only if $keyword[j]$ has a match in the subtree rooted at n, $1 \leq j \leq w$. Let $num(n.dMatch)$ be the decimal value of $n.dMatch$.

Example 4.11: In the running example of processing Q_3 (*Grizzlies*, *Gasol*, *Brown*, *position*), node *player* (0.1.0) contains matches to *Gasol* and *position* in its subtree, therefore its $dMatch$ is 0101.

Similarly, the $dMatch$ of *player* (0.1.1), *player* (0.1.2) and *name* (0.0) are 0001, 0011, 1000, respectively. ■

Instead of checking $n.dMatch$ with respect to each of its siblings, we associate its parent node n_p a boolean array $dMatchSet$ of size 2^w to record the $dMatch$ information of n_p's children. Specifically, $n_p.dMatchSet[i] = true$ if and only if n_p has a child n, such that $num(n.dMatch) = i$.

If the values of $dMatch$ and $dMatchSet$ are set, procedure $isContributor$ determines that a node n is a not a contributor if there exists a number j that *subsumes* i, and we have $n_p.dMatchSet[j]$ = $true$. A number j subsumes i if bitwise $AND(i, j) = i, j \neq i$.

Example 4.12: Continuing our running example, let n be the node *player* (0.1.1), $n.dMatch$ = 0001. Let n_p be its parent: *players* (0.1). Since the $dMatch$ of n_p's three children are 0101, 0001 and 0011, respectively, we set $n_p.dMatchSet[1]$ = $n_p.dMatchSet[3]$ = $n_p.dMatchSet[5]$ = $true$. Since 0011 subsumes 0001 and $n_p.dMatchSet[3]$ = $true$, n is not a contributor. On the other hand, both *player* nodes 0.1.0 and 0.1.2 are contributors. ■

To set the values of $dMatch$ and $dMatchSet$ for each $group = (t, M)$, the procedure $pruneMatches$ performs a post-order traversal of the tree T_g. For each node $n \in T_g$, if n matches a keyword $keyword[j]$, $1 \leq j \leq w$, then we set the j^{th} position of $n.dMatch$ to be 1. If n has children, we further set $n.dMatch$ according to the bitwise OR of the $dMatch$ of n's children. Then we set $dMatchSet(n_p)[num(dMatch(n))] = true$, where n_p is the parent of n.

To identify relevant matches, recall that a match in M is relevant if and only if all its ancestors up to SLCA node t are contributors (Definition 4.4). Equivalently, if a node is disqualified as a contributor, then none of the matches in its subtree can be relevant. The *pruneMatches* procedure performs a pre-order traversal of tree T_g in identifying relevant matches. If a node reached is not a contributor, then we skip its subtree, discarding all its descendant-or-self matches. The matches that remain in the traversal are considered to be relevant and are then output.

Example 4.13: In the running example, tree T_g consists of the paths in D_2 from $group[0].t$, *team* (0), to each match in $group[0].M$. We perform a pre-order traversal on T_g. We start with outputting the root *team* (0). Then since *name* (0.0) is a contributor, it is output. So does its child, a relevant match *Grizzlies* (0.0.0). Similarly we output the nodes on the path from *players* (0.1) to relevant matches *Gasol* (0.1.0.0.0) and *position* (0.1.0.2) (along with its value child). When we visit *player* (0.1.1), since it is not a contributor, we move to the next match in $group[0].M$ that is not a descendant of *player* (0.1.1): *Brown* (0.1.2.0.0). Then we reach *player* (0.1.2), which is a contributor. Finally, we output the nodes on the path from *player* (0.1.2) to relevant matches *Brown* (0.1.2.0.0) and *position* (0.1.2.2) (along with its value node), as shown in box annotated with $R(Q_3, D_2)$ in Figure 3 (c). ■

Now we analyze the complexity of MaxMatch. We use M_{max} and M_{min} to denote the maximum and minimum number of matches to a keyword, and M, $|D|$, d, w to denote the total number of keyword matches, the XML tree size, the depth of the XML tree and the number of keywords, respectively. Procedure $findMatch$ retrieves keyword matches from the inverted index, with cost bounded by $O(M)$. We adopt [21] for procedure $findSLCA$ with cost $O(M_{min}wd\log M_{max})$. Procedure $groupMatches$ merges the lists of matches in a merge-sort manner, which takes $O(M\log w)$ time. Then it traverses the match list and the SLCA list to group them, costing $O(Md)$. Finally, procedure $pruneMatches$ traverses the tree T_g consisting of the paths from SLCA to each match twice, one post-order and one pre-order, which takes $O(\min\{|D|, Md\} \cdot 2^w)$ time. 2^w is the cost of the *for* loop in $isContributor$. Therefore, the overall complexity of MaxMatch is $O(\min\{\min\{|D|, Md\} \cdot 2^w), M_{min}wd\log M_{max}\})$.

Baseball	
QB_1	Jim, Abbott, Outfield
QB_2	Jim, Abbott, James, Baldwin, Starting Pitcher
QB_3	player, Abbott, Baldwin
QB_4*	Tigers, Starting Pitcher, surname
QB_4	Tigers, Starting Pitcher, surname
QB_5	Tigers, Starting Pitcher, Outfield, surname
QB_6	Cordero, First Base
QB_7	Cordero, First Base, Tigers
QB_8	1998 Abbott Team
Mondial	
QM_1	United States, Birmingham, Population
QM_2*	United States, United Kingdom, Birmingham, Population
QM_2	United States, United Kingdom, Birmingham, Population
QM_3	Tasmania, Sardinia, Gotland, Area
QM_4	Ethnicgroups, Chinese, Indian, Capital
QM_5	Mondial, Country, Muslim
QM_6	Country, Muslim
QM_7	Asia, China, Government
QM_8	Organization, Name, Member

Figure 6: Part of Query Sets (36 Queries in Total)

5. EXPERIMENTS

To evaluate the effectiveness of MaxMatch, we tested three metrics: *search quality* measured by precision, recall and F-measure compared with the relevant query results obtained from user studies, *processing time* and *scalability*.

5.1 Experimental Setup

Equipment. The experiments are performed on a 3.0GHz AMD Athlon (TM) dual-core CPU running Microsoft Windows Server 2008 Enterprise operating system with 4.0GB memory. The algorithms are implemented in Microsoft Visual C++ 8.0. We use Oracle Berkeley DB [1] as the tool for creating inverted index and Dewey index.

The test data and part of the query sets are shown in Figure 6.

Data Set. We have tested two data sets: Baseball and Mondial. Baseball is a data set about the teams and players of North American baseball league.[7] Mondial is a world geographic data set.[8]

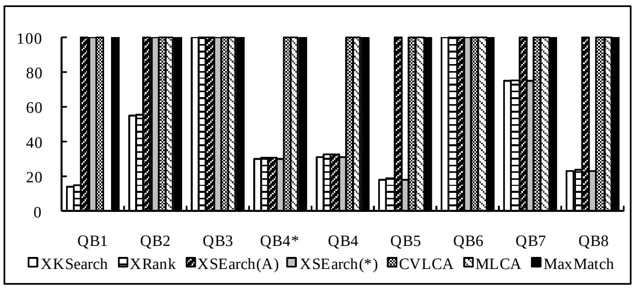
Query Set. Our query set consists of two parts with 36 queries in total. First we pick eight distinct queries for each data set, which are shown in Figure 6. These queries are chosen to represent a variety of cases, where both tag names and values are used.

To test the validity of data monotonicity and data consistency, the queries in Figure 6 with a '$*$' in their names are evaluated on the modified data sets. QB_4* is evaluated on the Baseball data set after removing a *Starting Pitcher* node within team *Tigers*. QM_2* is evaluated on the Mondial data set after removing the *Birmingham* node of United Kingdom.

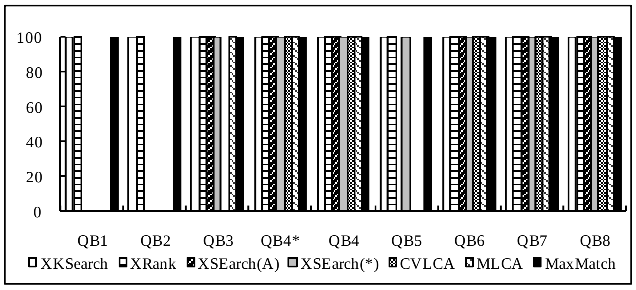
To test the validity of query monotonicity and query consistency, we design some query pairs, such that one contains one more keyword than the other, including QB_4 and QB_5, QB_6 and QB_7, QM_1 and QM_2, QM_5 and QM_6.

In addition, ten test queries for each data set are issued by students who are not involved in this project, which are omitted due to limited space.

[7]http://www.ibiblio.org/xml/books/biblegold/examples/baseball/.
[8]http://www.cs.washington.edu/research/xmldatasets.

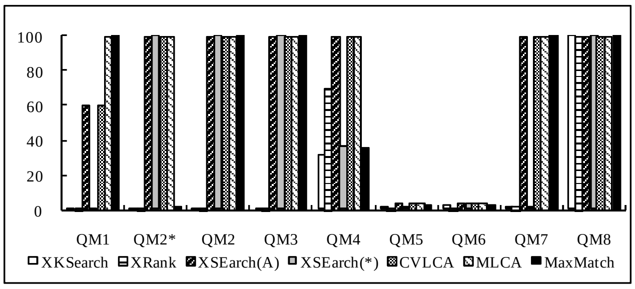


(a) Precision

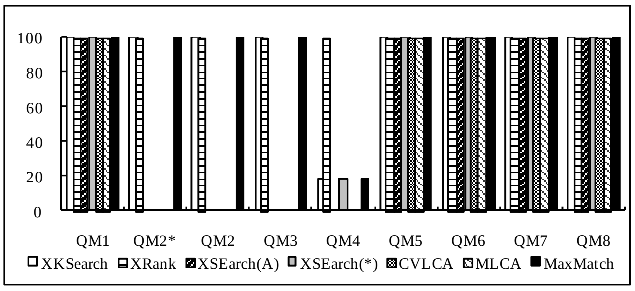


(b) Recall

Figure 7: Precision and Recall on Baseball Data Set



(a) Precision



(b) Recall

Figure 8: Precision and Recall on Mondial Data Set

5.2 Search Quality

To measure search quality, we need to assess the relevance of query results. We have conducted user surveys on the test data and queries to set the ground truth of relevant matches of each query. Thirteen students participated in the survey. Each participant was asked to specify relevant matches for each query. The ground truth of relevant matches are the ones selected by at least seven out of the thirteen users.

Perception of the Proposed Properties. For the queries designed to test the proposed properties, user study results confirm our intuition. Whenever we add a new keyword to a query, the number of relevant query results should not increase; and if a delta result tree exists, it should contain at least one match to the new keyword. Whenever we add a new data node, the number of relevant query results should not decrease; and if a delta result tree exists, it should contain the new data node.

System Analysis. We compared the search quality of MaxMatch with XKSearch [21], XRank [6], XSEarch [5] (including all-pair semantics and star semantics), CVLCA [13] and MLCA [14] based on the semantics described in those papers.

To measure the search quality, we use *precision*, *recall*, and *F-measure*. 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 is computed as:

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

The precision and recall of each approach on test queries in Figure 6 are shown in Figure 7 and Figure 8 respectively.

Now we analyze each approach. XKSearch always has a perfect recall except QM_4 (which will be discussed later), but generally

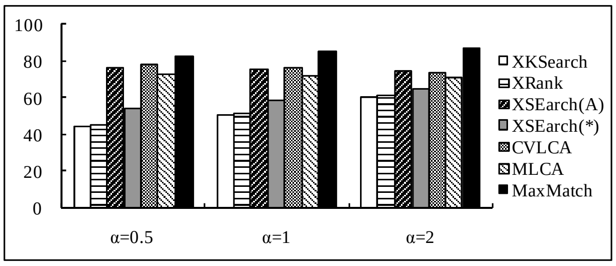
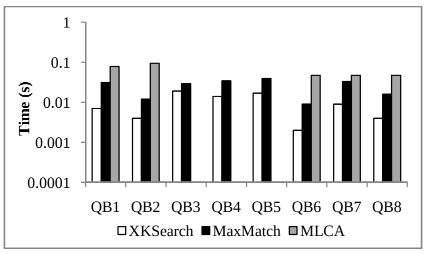


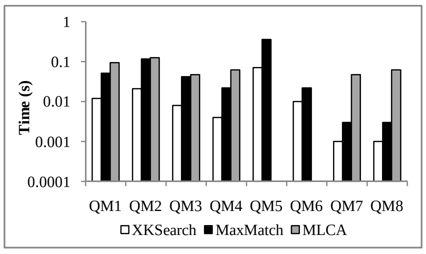
Figure 9: F-measure of All 36 Test Queries

has a very low precision as it outputs all the match nodes under each SLCA node, which are not necessarily relevant. Take QB_4 for example, XKSearch outputs the *surname* nodes of all the players, including the ones who are not *Starting Pitcher*s, and therefore has a low precision.

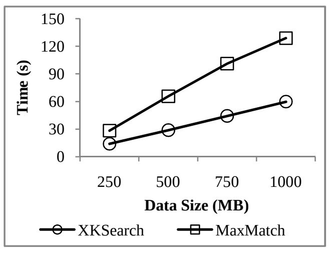
The query result of XRank, for a given query and document, is a superset of that of XKSearch. For many test queries, XRank has the same query results as XKSearch, such as QB_4, where the *surname* nodes of all the players (including the ones who are not *Starting Pitcher*s) are output. It has a different precision and recall from XKSearch for queries like QM_4. The semantics of QM_4 is to find capitals of the countries that have ethnicgroups Chinese or Indian. In the data, some countries have both Chinese and Indian, some have one of them, and some have neither. XKSearch outputs the countries that have both. On the other hand, besides outputting all such countries, XRank additionally considers the LCA that contains all query keywords of the remaining keyword matches, which is the document root. Therefore all keyword matches are considered to be relevant, and the information of the countries that have either ethnicgroups of Chinese or India are output, achieving a bet-



(a) Processing Time on Baseball



(b) Processing Time on Mondial



(c) Scalability

Missing bars or lines denote error reports during query evaluation.

Figure 10: Processing Time and Scalability

ter recall than XKSearch. Both approaches have a low precision as XKSearch outputs all *ethnicgroups* nodes within the countries that have both Chinese and Indian, and XRank outputs *capital* and *ethnicgroups* nodes that are not related to Chinese or Indian.

XSEarch star semantics has the same precision and recall as XKSearch on some queries such as QB_4 and QM_5. Take QB_4 for example. Keyword *Tigers* is a team name, and all *Starting Pitcher* and *surname* nodes within the team are interconnected with *Tigers*, and are considered relevant, which gives the same set of relevant matches as XKSearch. XSEarch star semantics has a zero recall on several queries, for example, QB_1. In the data, the player named Jim Abbott is not an outfield, and the semantics of the query is to find the players that play with Jim Abbott whose position is outfield. However, since the matches of *Jim* and *Abbott* are not interconnected with those of *Outfield*, XSEarch star semantics gives an empty result, leading to zero recall.

XSEarch all-pair semantics and CVLCA have zero recall on more queries than XSEarch star semantics, as they require that in a qualified pattern match, every two nodes must be interconnected. XSEarch all-pair and CVLCA have the same result for test queries except QB_3. For this query, XSEarch all-pair semantics correctly identifies relevant matches, as the match nodes are two *player* nodes which are interconnected. However, CVLCA generates empty result as the matches to *Abbott* and *Baldwin* are not interconnected.

The query result output by MLCA, for a given query and document, is a subset of that output by XKSearch. Some query results generated by MLCA are the same as XKSearch, such as QB_6 and QM_6. MLCA has a better precision on many queries such as QB_4, where irrelevant *surname* matches of players other than *Starting Pitcher* that are output by XKSearch are avoided. MLCA has a low recall for queries like QB_5, where an empty result is returned since there is no *surname* node that is meaningfully related with both *Starting Pitcher* and *Outfield*. In general MLCA has a high precision and a low recall.

MaxMatch has perfect precision and recall for most of the test queries, especially when the data structure is regular, such as the Baseball data set. However, there are queries where its performance can further be improved.

For QM_2*, since the *Birmingham* of United Kingdom is removed from the data, all *population* nodes under country United Kingdom is output by MaxMatch, leading to a low precision.

For QM_4, according to user study, the user would like to find the capitals of the countries that have ethnic group of Chinese or Indian. MaxMatch has a low recall because it only outputs the captials of the countries that have both Chinese and Indian ethnic groups. MaxMatch also has a low precision as it outputs not only the capital of each country, but also the capital of each province, which is irrelevant to the query.

For QM_5, the user would like to search the countries with religion *Muslim*. However, in the Mondial data set, a *Province* and a *City* node can have a child *Country*. All the approaches output these *Country* nodes of a *Muslim* country and thus have a low precision. In fact, for QM_5, XKSearch, XRank and XSEarch star semantics output all the *Country* nodes in the data set, which leads to a lower precision.

QM_6 searches for the country with Muslim. Since each province and city has a country child, and all these country nodes are output by all approaches, thus they all suffer from low precision.

Figure 9 shows the F-measure of all 36 test queries (including the 20 queries issued by users and 16 queries in Figure 6) with $\alpha = 0.5$, 1 and 2. As we can see, overall MaxMatch outperforms other approaches.

5.3 Processing Time and Scalability

We have implemented XKSearch/SLCA [21] and MaxMatch, both of which use the approach in [21] for computing SLCA of keyword matches. We use Timber [2] with default setting for identifying relevant matches under MLCA semantics. MLCA is proposed as a constructor of Schema-free XQuery, and can only be applied on tag names in Timber. To simulate its effect on keyword search, we convert the values in the test data into tag names with dummy value children (e.g., Frank is converted to <Frank/ >). Then we use the keywords as parameters of the MLCA function provided by Timber to obtain relevant matches under MLCA semantics. Since XML keyword search systems XSEarch, XRank, CVLCA are not available online, we compare MaxMatch with XKSearch and MLCA using the revised data set.

We use the Baseball data of size 1014KB, and a portion of the Mondial data of size 515KB. The processing times of XKSearch, MLCA, and MaxMatch over the queries on Baseball and Mondial data sets are shown in Figure 10 (a) and (b).

Both XKSearch and MaxMatch retrieve keyword matches and compute the SLCA nodes. XKSearch additionally needs to output all the matches in the trees rooted at SLCA nodes. MaxMatch additionally needs to group matches according to their SLCA ancestors, traverse subtrees rooted at SLCA nodes, identify relevant matches according to the descendant matches of each node in these subtrees. Therefore MaxMatch has processing overhead compared with XKSearch. For queries like QM_6, there are a lot of match nodes that are not descendants of any SLCA node, therefore the time for pruning irrelevant matches in the subtrees rooted at SLCA nodes is small, and the keyword match retrieval and SLCA computation time is the bottleneck. In this case, the processing time of XKSearch and MaxMatch are close. For queries where most of the keyword matches have an ancestor SLCA node, MaxMatch is slower than XKSearch. MLCA retrieves pattern matches to a query, and then checks whether the nodes in the pattern matches are pairwise meaningfully related, which is usually expensive.

We have tested the scalability of MaxMatch and XKSearch with respect to the increase of data size. The result is shown in Figure 10(c), which represents the processing of QB_1 on the Baseball data with size increased from 1MB to 1GB by replicating the original data set. The scalability of processing other queries are similar. It can be observed that the processing times of MaxMatch and XKSearch both increase linearly with the increase of the data size.

In summary, MaxMatch achieves improved search quality compared with existing XML keyword search engines in identifying relevant matches with efficiency.

6. RELATED WORK

We have reviewed the literature of XML keyword search on identifying relevant matches in Section 3. There are several orthogonal problems in XML keyword search.

Identifying Relevant Non-matches. Besides returning relevant matches as well as the paths connecting them, other data nodes that are not keyword matches may also be relevant to the query and should be returned. For example, consider keyword search *Gasol* on the XML data in Figure 1. This paper discusses which matches to *Gasol* are relevant and should be output. Besides matches, it would be desirable to output other nodes associated with this player, such as the *position* and *nationality* nodes, despite that they are not matches to keywords. To determine relevant non-matches, XKeyword [9] and Précis [12] allow a system administrator and/or users to specify them on a schema graph. [15] automatically infers relevant non-matches based on the characteristics of XML data and the pattern of input keywords.

Ranking Schemes. Ranking schemes have been studied for keyword search on XML documents. [9, 4] propose to rank query results according to the distance between different keyword matches in the document. In the presence of XML schema, efficient algorithms to compute top k results are presented in [9]. XSEarch [5] employs a ranking scheme in the flavor of information retrieval, considering factors like distance, term frequency, document frequency, etc. XRank [6] extends the page-rank hyperlink metric to XML.

This paper focuses on the problem of identifying relevant keyword matches. The techniques of determining relevant non-matching nodes and ranking query results are orthogonal issues and can be incorporated into MaxMatch.

Most of the work on XML keyword search including this paper considers *AND* semantics, except [5] and [19], which allow both *AND* and *OR* operators. Techniques for optimizing XML keyword search by exploiting materialized views have been studied [16].

7. CONCLUSIONS

This paper addresses an open problem of reasoning about XML keyword search algorithms. We take an axiomatic approach and have identified the properties that an XML keyword search algorithm should ideally possess in identifying relevant matches to keywords. Monotonicity states that data insertion (query keyword insertion) causes the number of query results to non-strictly monotonically increase (decrease). Consistency states that after data (query keyword) insertion, if an XML subtree becomes valid to be part of new query results, then it must contain the new data node (a match to the new query keyword). We have shown that these properties are non-trivial, non-redundant, and satisfiable. To the best of our knowledge, this is the first work that introduces properties to characterize reasonable behaviors of determining relevant matches for XML keyword search.

We have proposed MaxMatch, a novel semantics for identifying relevant matches and an efficient algorithm to realize this semantics, which is the only known algorithm that satisfies all properties. Experimental studies have verified the intuition of the properties and shown the effectiveness of our approach. MaxMatch is incorporated as part of the XSeek system [15] for XML keyword search, which intelligently identifies not only relevant matches to keywords but also relevant nodes that do not match keywords as query results.

This work initiates an investigation of reasoning about keyword search from a formal perspective. We have proposed four properties that are helpful to detect abnormal behaviors. They can serve as necessary conditions for a search engine to produce desirable query results, but may not guarantee a perfect search engine. For example, returning an entire XML database that contains matches to all the keywords in the query trivially satisfies these properties. The presence of additional desirable properties to further evaluate keyword search engines is an open problem. Consider another example, adding a new keyword *Tom* in a query *Cruise* may change the query semantics. A system that satisfies the proposed propensities will return information of star *Tom Cruise* and possibly the information about a tourist *Tom* in a *Cruise*. In this case, a good ranking scheme is important, which is part of our future work.

8. REFERENCES

[1] Oracle berkeley db. http://www.oracle.com/technology/products/berkeley-db/index.html.

[2] Timber project. http://www.eecs.umich.edu/db/timber/.

[3] K. Arrow. Social Choice and Individual Values. 1951.

[4] M. Barg and R. K. Wong. Structural proximity searching for large collections of semi-structured data. In *CIKM*, 2001.

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

[6] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD*, 2003.

[7] P. Hansen and F. S. Roberts. An impossibility result in axiomatic location theory. In *Mathematics of Operations Research*, 1996.

[8] 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.

[9] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *ICDE*, 2003.

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

[11] J. Kleinberg. An Impossibility Theorem for Clustering. In *NIPS*, 2002.

[12] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Précis: The essence of a query answer. In *ICDE*, 2006.

[13] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML Documents. In *CIKM*, 2007.

[14] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

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

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

[17] M. J. Osborne and A. Rubinstein. A Course in Game Theory. In *MIT Press*, 1994.

[18] D. M. Pennock, E. Horvitz, and C. L. Giles. An Impossibility Theorem for Clustering. In *AAAI*, 2000.

[19] C. Sun, C.-Y. Chan, and A. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *WWW*, 2007.

[20] V. Vesper. Let's Do Dewey. http://www.mtsu.edu/ vvesper/dewey.html.

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