Exploiting and Maintaining Materialized Views for XML Keyword Queries

ZIYANG LIU and YI CHEN, Arizona State University

Keyword query is a user-friendly mechanism for retrieving useful information from XML data in Web and scientific applications. Inspired by the performance benefits of exploiting materialized views when processing structured queries, we investigate the feasibility and present a general framework for answering XML keyword queries using materialized views. Then we develop an XML keyword search engine that leverages materialized views for query evaluation and maintains materialized views incrementally upon XML data update. Experimental evaluation demonstrates the significance and efficiency of our approach.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: XML, keyword search, materialized view

ACM Reference Format:
Liu, Z. and Chen, Y. 2012. Exploiting and maintaining materialized views for XML keyword queries. ACM Trans. Internet Technol. 12, 2, Article 6 (December 2012), 27 pages.
DOI = 10.1145/2390209.2390212 http://doi.acm.org/10.1145/2390209.2390212

1. INTRODUCTION

Web data are becoming increasingly heterogeneous due to the huge amount of data published on the Web from various sources. As a result, XML gains more popularity as a Web data representation format, thanks to the characteristics of semistructured data. Recently, keyword queries on XML data have attracted more and more research interest, as it is virtually the only way to query XML if the user has no knowledge of a structured query language, or the data schema is not available. Existing works on XML keyword queries focus on exploiting the mark-ups in XML data and generating meaningful and fine-grained query results, rather than returning the whole XML documents as a text search engine would do [Bao et al. 2009; Cohen et al. 2003; Guo et al. 2003; Hristidis et al. 2006; Li et al. 2004, 2007a, 2007b; Liu and Chen 2007, 2008b; Sun et al. 2007; Xu and Papakonstantinou 2005]. However, an important problem of how to efficiently evaluate and optimize XML keyword queries has not yet been well addressed.

Materialized views/semantic caching are queries whose results have been computed and are immediately available. Caching is used in many text search engines to speed up query processing by storing the results for selected queries. Traditionally, cached

This material is based on work partially supported by NSF CAREER Award IIS-0845647, IIS-0915438, an IBM Faculty Award and a Google Research Award.
Z. Liu is currently affiliated with NEC Laboratories, Cupertino, CA.
Authors' addresses: Z. Liu, NEC Laboratories, Cupertino, CA; email: ziyang@nec-labs.com; Y. Chen (corresponding author), The School of Computing, Informatics and Decision Systems Engineering, Arizona State University, AZ; email: yi@asu.edu.

DOI 10.1145/2390209.2390212 http://doi.acm.org/10.1145/2390209.2390212

query results can only be used to answer exactly the same query. Semantic caching allows to answer new queries using cached results of existing queries, if the new query is totally or partially answerable by the cache. It further reduces network accesses, and has been proved successful for performance optimization in evaluating Web queries as well as structured queries on XML and databases [Arion et al. 2007; Balmin et al. 2004; Feng et al. 2007; Mandhani and Suciu 2005; Onose et al. 2006; Tang et al. 2008; Xu and Ozsoyoglu 2005], which are used to store data for many Web sites.

A natural question is whether we can exploit semantic cache/materialized views for keyword queries on XML data. In this article, we study two related open problems. First, how should we leverage materialized views of previous keyword query results for efficient evaluation of new keyword queries on XML? Second, when the source data is updated, how should we incrementally maintain materialized views of XML keyword queries to keep them fresh? Due to the differences of syntax, semantics, and query processing techniques between XPath/XQuery and keyword queries, answering these questions for XML keyword queries presents a new set of challenges. Compared with structured queries, a major difference of keyword queries is that no two distinct keyword queries have containment relationship (i.e., the results of one query are a subset of that of another query) according to query result definitions in the literature. This makes answering queries using views completely different for structured queries and keyword queries.

Note that it is possible to convert a keyword query into an XQuery and then apply XQuery query processing and optimization techniques. However, as evaluated in the literature [Cohen et al. 2003; Li et al. 2004], such an approach incurs excessively high time complexity and demands the existence of data schema. Thus all existing XML keyword search engines opt to design specific techniques tailored for evaluating keyword queries rather than converting them into XQuery queries, including this work. Next we present the unique challenges of exploiting and maintaining materialized views for XML keyword queries.

—*How to exploit materialized views for query evaluation.* Suppose we have a set of materialized views of previous XML keyword query results. Upon the arrival of a keyword query, we would like to utilize materialized views as much as possible in order to gain performance improvements. The first question is how to determine which views are *relevant* to a given query. If a relevant view itself cannot answer the query, can we find a set of views which together can answer the query, that is, an *answering set*? If there are several answering sets, we need to efficiently find the optimal one for query answering. Furthermore, techniques need to be developed to answer the query with an answering set.

One immediate question is whether we can apply existing cache techniques for keyword queries on text documents to handle keyword queries on XML data. For keyword queries on text documents, the result granularity is fixed to a document. We can simply do a merge join on the result of subqueries in the cache to obtain the result of the user query, where the cost function is proportional to the total number of matched documents to query keywords. However, the results of XML keyword queries are dynamically selected *fragments* of XML documents based on the XML tree structure and the position of the keyword matches, as will be shown in Section 4. This makes the process of using materialized views completely different between XML and text documents, from view selection algorithms, cost function and effectiveness analysis, to the algorithms for answering queries using views.

Example 1.1. Consider a keyword query (*mid-size, sedan, 2006, CNN, award*) on the entire XML tree in Figure 1. Intuitively, the subtree rooted at *award* (10) is the

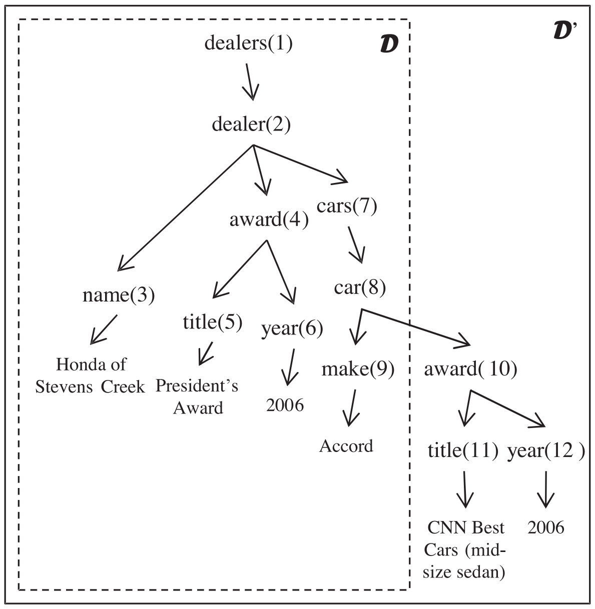


Fig. 1. XML trees $\mathcal{D}$ and $\mathcal{D}'$.

result of this query, as it is the smallest subtree that contains all the query keywords, indicating a close relationship of the keywords. Indeed, many existing XML search engines [Hristidis et al. 2006; Li et al. 2004; Liu and Chen 2007, 2008b; Sun et al. 2007; Xu and Papakonstantinou 2005] return *award* (10) as the result. Suppose we have a set of materialized views for this XML data, each of which is a query whose results are materialized. For example, the materialized view for query "mid-size, sedan, award" is the results of this query, for example, the subtree rooted at node award (10). Given this set of materialized views, we first need to identify which ones are relevant. Suppose the relevant views are: V_1: (*mid-size, sedan, award*), V_2: (*sedan, CNN*), V_3: (*2006, award*), and V_4: (*mid-size, 2006, award*). Then how should we maximally exploit relevant views to minimize query processing cost? Assuming we find two sets of views to answer the query: $S_1 = \{V_1, V_2, V_3\}$, $S_2 = \{V_2, V_4\}$, we need to efficiently select the best answering set for answering queries using views.

Furthermore, techniques differing from answering keyword queries using cache on text documents need to be developed to use the selected answering set for query processing. For example, suppose we choose S_1 to answer the query; the results of V_1, V_2 are the subtrees rooted at *award* (10) and *title* (11), respectively, and the results of V_3 are the subtrees rooted at *award* (4) and *award* (10). As we can see, the query result, which is the subtree rooted at *award* (10), is neither the intersection nor the union of the materialized views.

—*How to incrementally maintain materialized views.* For a materialized view to be useful for evaluating queries, it must be up-to-date with respect to dynamic source data. Given an update to the source data, which materialized views are affected? Can we maintain these materialized views incrementally using the original views and a small fragment of the source data without recomputing the views on the updated data from scratch?

These questions pose unique challenges for maintaining materialized views of XML keyword queries compared with structured queries and keyword queries on text documents. When considering conjunctive structured queries or conjunctive keyword queries on text documents, insertion (deletion) to the data will not cause deletion (insertion) in the materialized views. However, this does not hold for keyword queries on XML data. Due to the ambiguity of keyword queries, existing search engines follow the best-effort approach to determine the semantics based on the current XML data and user input keywords [Cohen et al. 2003; Guo et al. 2003; Li et al. 2004, 2007a, 2007b; Liu and Chen 2007, 2008b; Xu and Papakonstantinou 2005]. When new knowledge is added to the data, the inferred query semantics may be refined. Such a query result definition does not have the monotonicity properties, as shown in Example 1.2, and brings new challenges for incremental view maintenance.

Example 1.2. Consider a materialized view V of keyword query Q (*Accord, award*) on the XML tree $\mathcal{D}$ in Figure 1. It is reasonable that the keyword matches in the subtree rooted at node *dealer* (2) constitute query result V. Now we insert a subtree rooted at *award* (10) into $\mathcal{D}$ and obtain the new XML tree $\mathcal{D}'$. Leveraging the information of the newly inserted data, a better semantics of Q would be: searching for the award of Accord cars. To be consistent with this semantics, the result of Q should now be the subtree rooted at *car* (8), thus the materialized view should be updated [Bao et al. 2009; Guo et al. 2003; Li et al. 2004; Liu and Chen 2007, 2008b; Xu and Papakonstantinou 2005].

To address these challenges, we present a general framework for exploiting materialized views for XML keyword queries. We first identify the relevant materialized views that potentially can be used to answer a user query. We prove that given a set of relevant materialized views $\mathcal{V}$ and a user query Q, the decision problem of finding the minimal answering set in $\mathcal{V}$ is NP-complete. These results are based on query results defined as a variation of the *Lowest Common Ancestor* (LCA) semantics, such as SLCA (Smallest LCA) [Chen and Papakonstantinou 2010; Liu and Chen 2007, 2008b; Sun et al. 2007; Xu and Papakonstantinou 2005], MLCA (Meaningful LCA) [Li et al. 2004, 2007a], CVLCA (Compact Valuable LCA) [Li et al. 2007a], ELCA (Exclusive LCA) [Chen and Papakonstantinou 2010; Guo et al. 2003; Xu and Papakonstantinou 2008], XSEarch [Cohen et al. 2003], etc.

We then design and implement an XML keyword search engine that can answer queries using materialized views. A polynomial-time $2(\ln|Q| + 1)$ approximation algorithm is proposed that finds the optimal answering set of materialized views to evaluate the input query, where $|Q|$ is the number of keywords in Q. In order to keep the materialized views fresh, they are incrementally maintained upon XML data insertion or deletion. The realization of these functionalities depends on how query results are defined. Our keyword search engine adopts a commonly used semantics to define query results for XML keyword queries, the SLCA semantics, which will be reviewed in Section 4. SLCA semantics is used in many XML keyword query systems,[1] including XKSearch [Xu and Papakonstantinou 2005], XSeek [Liu and Chen 2007], MaxMatch [Liu and Chen 2008b], and Sun et al. [2007]. The techniques that we propose for answering queries using views and maintaining materialized views of XML keyword queries can be incorporated into these systems.

Note that the focus of this article is to utilize and maintain the views given the materialized views. How the materialized views are selected is orthogonal to the problems to be studied in this article. Some simple yet effective ways of selecting views include

[1]Some systems perform additional node filtering after SLCA computation is done.

using recent queries, most frequent queries, frequent small queries, etc. Whether there are even better ways for view selection is out of this article's scope.

Our technical contributions include the following.

— In this article we study a new research challenge that has not been explored in the literature: how to exploit and maintain materialized views for XML keyword queries.
— In Section 3 we analyze the problem of identifying relevant materialized views to a query, and show that the decision problem of finding the minimal answering set to a query on any LCA semantics is NP-complete.
— We develop the first XML keyword search engine that can answer queries using materialized views and maintain materialized views incrementally upon XML data update, as presented in Sections 4 and 5.
— Section 6 presents experimental evaluation, which shows significant performance improvements of answering queries using views and the efficiency of maintaining views incrementally.

Related works are reviewed in Section 2. We conclude the article with a discussion of future work in Section 7.

2. RELATED WORK

Processing Keyword Queries on XML Data. Keyword queries on XML has attracted a lot of research interest in the database community. Due to the inherent ambiguity of keyword queries, not all keyword match nodes are relevant. To address this, most of the existing work on XML keyword queries focuses on identifying relevant keyword matches and defining meaningful query results [Bao et al. 2009; Chen and Papakonstantinou 2010; Cohen et al. 2003; Guo et al. 2003; Hristidis et al. 2006; Li et al. 2004, 2007a, 2007b, 2008; Liu and Chen 2007, 2008b; Liu et al. 2010b; Sun et al. 2007; Xu and Papakonstantinou 2005]. Most of these approaches adopt LCA-based semantics, such as SLCA [Chen and Papakonstantinou 2010; Hristidis et al. 2006; Liu and Chen 2007, 2008b; Sun et al. 2007; Xu and Papakonstantinou 2005], MLCA [Li et al. 2004, 2007b], ELCA [Chen and Papakonstantinou 2010; Guo et al. 2003; Xu and Papakonstantinou 2008], CVLCA [Li et al. 2007a], etc. XSeek [Liu and Chen 2007, 2010] further proposes strategies to identify relevant nodes to be included in a query result even though they do not match keywords. XReal [Bao et al. 2009] addresses two keyword ambiguity problems for XML keyword queries using IR ideas, and proposes an IR-based relevance-oriented ranking scheme. AND semantics of keyword queries is adopted in most of the work including this article. Chen and Papakonstantinou [2010] study techniques for top-k result generation for SLCA and ELCA semantics. There are also works on result analysis, such as snippet generation [Huang et al. 2008; Liu et al. 2010a], result comparison [Liu and Chen 2012], result clustering [Liu and Chen 2010], and query suggestion [Liu et al. 2011]. Please refer to the surveys of keyword search on structured data [Chen et al. 2009, 2011; Liu and Chen 2011].

Exploiting and Maintaining XPath/XQuery Views. There is a lot of work on evaluating queries using materialized views for XPath, XQuery, and their subsets [Arion et al. 2007; Balmin et al. 2004; Mandhani and Suciu 2005; Onose et al. 2006; Tang et al. 2008; Xu and Ozsoyoglu 2005]. Most of the work focuses on queries with child/descendant axes, wildcards, XPath predicates, equality and inequality comparisons, nested FLWR blocks and joins. Recently, Tang et al. [2008] studied the strategy of selecting multiple materialized views for rewriting XPath queries. Materialized view maintenance for XPath/XQuery and their subsets has also been well studied [Sawires et al. 2005, 2006].

Fan et al. [2007] and Shao et al. [2007] address the problem of processing keyword queries on virtual views defined by XQuery or annotated DTD without materializing them. The keyword queries, though posed on views (e.g., for user privilege reasons), are evaluated on source XML data. The technical challenge there is query rewriting.

Handling materialized views of XML keyword queries requires a completely different set of techniques than those of XPath/XQuery. Existing work on answering XPath/XQuery using materialized view focuses on inferring query containment relationships from the logical relationships among different axes, query structure, query node name test, and the logical relationship among predicates. However, as stated before, keyword queries do not have such containment relationships; there are no two distinct keyword queries, such that the result of one query is always the subset of the results of the other query on any data. As we show later, finding the best view set to answer a keyword query is an NP-hard problem.

A brief description of this work is presented in a poster paper [Liu and Chen 2008a]. In comparison, this article presents techniques for incremental view maintenance, which is not discussed in Liu and Chen [2008a]. The general framework in Section 3 can apply on any LCA semantics, beyond SLCA semantics. Furthermore, for answering queries using views with SLCA semantics, Liu and Chen [2008a] finds the answering set with the smallest number of views, while the algorithm in Section 4 finds the answering set with the *smallest cost* to answer a query.

3. EXPLOITING MATERIALIZED VIEWS FOR QUERY EVALUATION

3.1. Definitions

We consider an XML tree model, in which internal nodes denote elements and attribute names, and leaf nodes denote values. Each XML node n has a *Dewey* label [Tatarinov et al. 2002] as a unique ID, referred to as $Dewey(n)$. 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. Dewey ID can be used to detect the relationship between nodes. A node n_1 is an ancestor of node n_2 if and only if $Dewey(n_1)$ is a prefix of $Dewey(n_2)$. This indicates that the LCA of nodes n_1 and n_2 has the Dewey ID which is the longest common prefix of $Dewey(n_1)$ and $Dewey(n_2)$. Besides, we say $Dewey(n_1)$ is smaller than $Dewey(n_2)$ if n_1 appears before n_2 in document order.[2]

A query issued by the user is a set of keywords. We take the AND semantics of XML keyword query and define the query result of XML keyword query as follows.

Definition 3.1. The *query result* of evaluating a keyword query Q on XML data $\mathcal{D}$, denoted as $Q(\mathcal{D})$, consists of the node identifiers of *selected* Lowest Common Ancestors (LCA) of keyword matches of Q. According to the definition of LCA, each $s \in Q(\mathcal{D})$ satisfies: the set of keyword matches in *s*'s subtree contains at least one match to each keyword in Q; and this set is not the same as the set of keyword matches in the subtree rooted at any child of *s*.

Note that in practice, most XML search engines return subtrees rooted at LCA nodes, rather than just the LCA nodes themselves, as query results. Different search engines return different parts of the subtrees rooted at LCA nodes, but most of them have a commonality: they first compute LCA nodes, then compute the subtrees to be returned. Therefore, the focus of this article is to use materialized views to efficiently

[2]Note that upon a data update, the Dewey labels of some nodes may be affected. Efficient maintenance of Dewey labeling is an orthogonal to this article. It has been investigated in the literature and can be achieved without relabeling [Cohen et al. 2002; Li et al. 2006; O'Neil et al. 2004; Xu et al. 2009].

compute LCA nodes (rather than computing the full results), so that it can be applied to a number of existing works. Therefore, in the remainder of this article, we consider query results as LCA node identifiers, as defined in Definition 3.1.

As we can see, Definition 3.1 specifies a necessary condition that each XML keyword query result should satisfy. For example, consider query (*Accord, award*) on XML tree $\mathcal{D}$ in Figure 1. The subtree rooted at *dealers*(1) contains exactly the same set of keyword matches as that in the subtree rooted at its child *dealer*(2). Intuitively, the subtree rooted at *dealer*(1) is larger but does not provide additional information compared to the subtree rooted at *dealer*(2) with respect to the query, and therefore should not be considered as a desirable query result. There are many variations in the literature of defining query results for XML keyword query involving AND semantics [Cohen et al. 2003; Guo et al. 2003; Hristidis et al. 2006; Li et al. 2004, 2007a, 2007b; Liu and Chen 2007; Xu and Papakonstantinou 2005], all of which are special cases of Definition 3.1.

A *materialized view* is a query whose result is materialized. Queries and views are used interchangeably in this article. The size of a query Q, denoted as $|Q|$, is the number of keywords in Q. The size of a query result (or materialized view) on data $\mathcal{D}$, denoted as $|Q(\mathcal{D})|$, is the number of data nodes it has.

Note that given a query Q and a set of materialized views $\mathcal{V}$, if a keyword in Q is not in any view in $\mathcal{V}$, then $\mathcal{V}$ itself is not sufficient to answer the query. In this case, we additionally access the data index to retrieve the matches to keywords that are in Q but not in $\mathcal{V}$, which together with $\mathcal{V}$ can answer Q. A *full-text inverted index* is commonly used in keyword search engines, which maps a word to a sorted list of its occurrences in the data. In this article, we address the general problem of answering queries in the presence of both the inverted index and materialized views. For simplicity of presentation, we consider the invert index as a set of materialized views, each of which contains a single keyword, and present techniques for optimizing query evaluation using (these generalized) materialized views.

The relationship between two keyword queries with respect to the set of keywords they contain is defined as follows.

Definition 3.2. Let Q_1 and Q_2 be two keyword queries. We say Q_2 is a *subquery* of Q_1, or Q_1 is a *superquery* of Q_2, denoted as $Q_2 \trianglelefteq Q_1$, if every keyword in Q_2 is in Q_1. Q_2 is a *proper subquery* of Q_1, denoted as $Q_2 \,)\, Q_1$, if $Q_2 \trianglelefteq Q_1$, and there exists a keyword k, $k \in Q_1$, $k \notin Q_2$.

3.2. Analysis

Given the general definition of XML keyword query results, we now discuss how to identify relevant views, answering sets, and the minimal answering set of a query, in order to leverage materialized views for query answering.

Definition 3.3. A set of views $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$ *can answer query* Q, if there exists an algorithm $\mathcal{A}$ such that for any data $\mathcal{D}$, $Q(\mathcal{D}) = \mathcal{A}(\mathcal{V}(\mathcal{D}))$, where $\mathcal{V}(\mathcal{D})$ is the set of materialized views, $\mathcal{V}(\mathcal{D}) = \bigcup_{V \in \mathcal{V}} \{V(\mathcal{D})\}$. If $\mathcal{V}$ can be used to answer Q, it is named as an *answering set* of Q.

Next we discuss how to exploit materialized views in the context of XML keyword queries. The following proposition shows that the views that are subqueries of a query Q are helpful for evaluating Q.

PROPOSITION 3.1. *Consider a universe of keywords* $K = \{k_1, k_2, \ldots, k_n\}$*, a set of materialized views* $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$*,* $V_i \trianglelefteq K$*,* $1 \le i \le m$*, and a query* $Q \trianglelefteq K$*. Let* $\mathcal{V}'$ *be the set of subqueries of* Q *in* $\mathcal{V}$*:* $\mathcal{V}' = \{V | V \in \mathcal{V} \wedge V \trianglelefteq Q\}$*. We claim that* Q *can be*

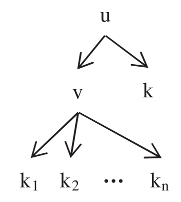


Fig. 2. Proposition 3.2.

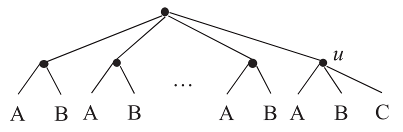


Fig. 3. LCA of a superquery of Q is not helpful for finding the LCA of Q.

answered by $\mathcal{V}$ if and only if Q can be answered by $\mathcal{V}'$. If a view is in $\mathcal{V}'$, it is called a relevant view *to Q.*

PROOF. $\Leftarrow$: If Q can be answered by $\mathcal{V}'$, since $\mathcal{V}' \subseteq \mathcal{V}$, Q can be answered by $\mathcal{V}$.

$\Rightarrow$: Suppose (i) Q can be answered by $\mathcal{V}$, (ii) but cannot be answered by $\mathcal{V}'$. Let $\mathcal{A}$ be an algorithm that can answer Q using $\mathcal{V}$ on any document. By assumption (ii), there exists a document $\mathcal{D}$, such that $\mathcal{A}$ cannot answer Q on $\mathcal{D}$ using $\mathcal{V}'$ (i.e., $\mathcal{A}(\mathcal{V}'(\mathcal{D})) \neq Q(\mathcal{D})$). Now for every distinct word in tag names or values t in $\mathcal{D}$, $t \notin Q$, we replace all the occurrences of t with t', where t' is a word that does not appear in $\mathcal{D}$, and denote the new XML tree to be $\mathcal{D}'$. Since $\mathcal{D}'$ and $\mathcal{D}$ only differ in nonkeyword words, $Q(\mathcal{D}) = Q(\mathcal{D}')$, $\mathcal{V}'(\mathcal{D}) = \mathcal{V}'(\mathcal{D}')$. It is easy to see that for any view V_i that contains keywords that are not in Q (i.e., $V_i \in \mathcal{V}$, $V_i \notin \mathcal{V}'$), we have $V_i(\mathcal{D}') = \emptyset$. Therefore $\mathcal{V}(\mathcal{D}') = \mathcal{V}'(\mathcal{D}')$. According to assumption (ii), we have $\mathcal{A}(\mathcal{V}(\mathcal{D}')) \neq Q(\mathcal{D}')$. However, this means that $\mathcal{A}$ cannot answer Q on $\mathcal{D}'$ using $\mathcal{V}$, which conflicts with assumption (i). □

Proposition 3.1 indicates that views of superqueries of Q are unnecessary for answering Q. Besides, we observe that it is highly unlikely that a materialized view of a superquery of a query Q is useful for answering Q. Consider the example in Figure 3, and the query "*A, B, C*". As we can see, The LCA of (A, B, C) (node u) merely tells us that there are LCA nodes of (A, B) in u's subtree. However, first, it does not tell us where they are. There is no way to identify the locations of the LCA of (A, B) within u's subtree (it could be located anywhere within the subtree), other than scanning the subtree or building an index for the subtree, both of which are costly. Second, there may be LCA nodes of (A, B) *not* in the subtree of u. Therefore, the materialized view of (A, B, C) won't help us answer query (A, B) for most LCA-based semantics. Whether views of superqueries can be used for query optimization for a certain LCA-based semantics is one of our future works. In the remainder of this article, we only consider views of subqueries of Q when answering Q.

Proposition 3.2 shows that a set of views $\mathcal{V}$ that are subqueries of Q is able to answer Q only if the union of keywords in $\mathcal{V}$ are the same as the set of keywords in Q.

PROPOSITION 3.2. *Given a query Q and a set of relevant materialized views $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$, each view is a subquery of Q, that is, $V_i \trianglelefteq Q$, $1 \leq i \leq m$. If Q can be answered by $\mathcal{V}$, that is, $\mathcal{V}$ is an answering set of Q, then $\bigcup_{V \in \mathcal{V}}(V) = Q$.*

PROOF. Assume that Q can be answered by $\mathcal{V}$, but $\bigcup_{V \in \mathcal{V}}(V) \neq Q$. Since $V_i \trianglelefteq Q$, there exists a keyword k such that $k \in Q$, and $k \notin \bigcup_{V \in \mathcal{V}}(V)$. Let the set of keywords $\bigcup_{V \in \mathcal{V}}(V)$ be $\{k_1, k_2, ..., k_n\}$. We can construct an XML tree $\mathcal{D}$ as illustrated in Figure 2.

Note that we must have the knowledge of where the matches to k locate, in order to find the result of Q. However, none of the views in $\mathcal{V}$ provides such information according to Definition 3.1. Therefore, Q cannot be answered by $\mathcal{V}$. $\square$

One interesting observation of Propositions 3.1 and 3.2 is that relevant views and answering sets for XML keyword queries can be identified in the same way as those for keyword queries on text documents, despite that the former has an LCA-based query result definition and the latter has the whole document as a query result. However, the proofs are significantly different.

If for a given query, there are multiple answering sets in a set of materialized views, we need to find out the best one. Intuitively, the fewer number of views in an answering set, the more efficiently it can be used to answer the query due to less computational cost and smaller intermediate query results.[3] However, finding such an answering set is hard.

Definition 3.4. Consider a universal keyword set K, a query Q, $Q \subseteq K$, and a set of materialized views $\mathcal{V}$, $\forall V \in \mathcal{V}$, $V \subseteq K$. The *minimal answering set problem* is the following: given an integer i, can we find an answering set $\mathcal{V}'$ of Q, that is, $\mathcal{V}' \subseteq \mathcal{V}$, $\forall V \in \mathcal{V}'$, $V \trianglelefteq Q$, $\bigcup_{V \in \mathcal{V}'}(V) = Q$, such that $|\mathcal{V}'| \leq i$?

It is easy to see that the minimal answering set problem is NP-complete as it is equivalent as the minimum set cover problem.

Having defined relevant materialized views and answering sets and analyzed the complexity of finding the minimal answering set for XML keyword queries, the next question is how to use answering sets for query evaluation. In the article, we do not discuss the trivial case where only the materialized view that is *exactly the same as* a query can be used in query evaluation, but focus the discussion on the general case where the answering set of a query can be any set of materialized views satisfying Proposition 3.2. To this end, we must find an algorithm that can compute the query result of Q from the results of Q's (proper) subqueries. However, whether this is achievable depends on the specific definition of the keyword query result.

Fortunately, we discover that a common query result definition of XML keyword queries, Smallest Lowest Common Ancestor (SLCA), as used in Hristidis et al. [2006], Li et al. [2004], Liu and Chen [2007, 2008b], Xu and Papakonstantinou [2005], and Chen and Papakonstantinou [2010], allows the result of a query to be computed from the results of its subqueries, as will be analyzed in Section 4. It is an open question whether other XML keyword query result definitions in the literature allow query evaluation using subqueries.

Furthermore, given the query result definition and the algorithms of answering queries using views, in Section 4 we will also discuss how to find the *optimal answering set* with minimal cost. Section 5 will present how to efficiently keep the views up-to-date upon a change to the data.

4. HANDLING MATERIALIZED VIEWS IN AN XML KEYWORD SEARCH SYSTEM

In this section, we present an XML keyword search system that answers queries using an answering set of materialized views, adopting a commonly used query result definition: Smallest Lowest Common Ancestor (SLCA) [Xu and Papakonstantinou 2005]. Section 4.1 introduces the SLCA semantics. Section 4.2 presents the algorithm that

[3]To be more accurate, we should select the answering set that can evaluate the query with the minimal cost. However, the cost measurements depend on the complexity of the algorithms that answer a query using views, which in turn depends on the definition of query result. Therefore we defer the discussion of finding the optimal answering set with minimal cost in Section 4 when the query result definition is set.

answers a query using views and shows its complexity. In Section 4.3 we show that finding an answering set with the minimal cost using SLCA semantics is also an NP-hard problem. Finally we propose an approximation algorithm for selecting answering set, whose approximation ratio is proved to be 2(ln|Q|+1).

4.1. Search Semantics

We first present the definition of *Smallest Lowest Common Ancestor* (SLCA), which is a variant of the LCA semantics (Definition 3.1).

Definition 4.1. Let $S_1, S_2, ..., S_n$ be n sets of nodes in XML data $\mathcal{D}$. $SLCA(\mathcal{D}, S_1, S_2, ..., S_n)$ consists of all such nodes s such that:

(1) s contains at least one node in each S_i ($1 \leq i \leq n$) in its subtree;
(2) there does not exist a descendant of s that satisfies condition 1.

For a keyword query $Q = \{K_1, K_2, ..., K_n\}$ on XML document $\mathcal{D}$, the set of Smallest Lowest Common Ancestor $SLCA(\mathcal{D}, Q) = \{SLCA(\mathcal{D}, S_1, S_2, ..., S_n)|S_i$ is the set of nodes in $\mathcal{D}$ that match keyword K_i, $1 \leq i \leq n.\}$.

The query result of a keyword query Q on XML data $\mathcal{D}$ is the set of SLCA nodes, $Q(\mathcal{D}) = SLCA(\mathcal{D}, Q)$.

Example 4.1. Consider query Q = (*Accord, award*) on $\mathcal{D}'$ in Figure 1. We have $SLCA(\mathcal{D}', \{Accord, award\}) = SLCA(\mathcal{D}'$, matches to *Accord*, matches to *award*)={*car* (8)}. Node *dealer* (2) is not an SLCA node, even though it contains both keywords in its subtree. Intuitively, the keyword matches connected through node *car* (8) have a closer relationship compared with the ones connected through node *dealer* (2).

4.2. Answering Queries Using Materialized Views

In this section, we present the algorithm of answering queries using an answering set for the query result definition given in Definition 4.1.

Recall that we observed in Section 3 that the materialized view of a superquery of Q is likely unhelpful for answering Q. Therefore, we answer a query Q using Q's subqueries. Next we show the relationship between the result of a query Q and the results of its subqueries whose union is equal to Q.

PROPOSITION 4.2. *For two queries Q_1 and Q_2 on data $\mathcal{D}$, $SLCA(\mathcal{D}, Q1 \cup Q2) = SLCA(\mathcal{D}, SLCA(\mathcal{D}, Q1), SLCA(\mathcal{D}, Q2))$.*

PROOF. According to Definition 4.1, for each $s \in SLCA(\mathcal{D}, Q_1 \cup Q_2)$, s contains all the keywords in both Q_1 and Q_2. Therefore s must contain at least one node in $SLCA(\mathcal{D}, Q_1)$ and one node in $SLCA(\mathcal{D}, Q_2)$ in its subtree. Furthermore, none of s's descendants can contain a node in $SLCA(\mathcal{D}, Q_1)$ and a node in $SLCA(\mathcal{D}, Q_2)$ in its subtree (otherwise, such a descendant disqualifies s to be in $SLCA(\mathcal{D}, Q_1 \cup Q_2)$). Therefore, all such s nodes compose $SLCA(\mathcal{D}, SLCA(\mathcal{D}, Q_1), SLCA(\mathcal{D}, Q_2))$. □

COROLLARY 4.3. *For query $Q = Q_1 \cup Q_2 \cup ... \cup Q_k$, we have $SLCA(\mathcal{D}, Q) = SLCA(\mathcal{D}, SLCA(\mathcal{D}, Q_1), ..., SLCA(\mathcal{D}, Q_k))$.*

The algorithm of answering a query using views is presented in Algorithm 1. The answering set, $\mathcal{V}_{ans}$, is obtained using Algorithm 2. It computes the SLCAs from the materialized views in the answering set according to Corollary 4.3.

Example 4.4. Consider evaluating $Q = \{A, B, C, D, E\}$ given a set of materialized views $\mathcal{V}$: $V_1 = \{A, B\}$, $V_2 = \{A, B, C\}$, $V_3 = \{D\}$, $V_4 = \{B, D\}$, $V_5 = \{E, F\}$. $V_1, \cdots, V_4$ are subqueries of Q and therefore are relevant to Q. A candidate answering set can be

Algorithm 1: Answering Query Using Views

```
Input: keyword query $Q$, data $\mathcal{D}$, answering set $\mathcal{V}_{ans}$ obtained using Algorithm 2
Output: The results of $Q$ on $\mathcal{D}$: $SLCA(\mathcal{D}, Q)$
1: $Mview$ = the materialized view of $\mathcal{V}_{ans}[1]$
2: $SLCA(\mathcal{D}, Q) = Mview$
3: for $j = 2$ to $\mathcal{V}_{ans}.size$ do
4:     $Mview$ = the materialized view of $\mathcal{V}_{ans}[j]$
5:     $SLCA(\mathcal{D}, Q) = computeSLCA(SLCA(\mathcal{D}, Q),$
       $SLCA(\mathcal{D}, Mview))$ {We use the algorithm in XKSearch to implement function
       $computeSLCA$.}
6: return $SLCA(\mathcal{D}, Q)$
```

$\{V_2, V_4, V_E\}$ where V_E is a view of a single keyword E. Note that although V_E is not a materialized view, its result can be retrieved from the inverted index (as introduced in Section 3.1), and is thus equivalent as a materialized view.

According to Corollary 4.3, to compute the SLCA of a query Q using a set of views, we can use the same procedure which computes SLCA of Q from the data, by considering each materialized view as the set of matches to each keyword in Q. In this article we adopt the SLCA computation algorithm proposed in XKSearch[4], whose complexity is $O(S_{min} \sum_{Si} \log(Si))$[5], where S_{min} is the number of matches to the most selective keyword, and S_i is the number of matches to the ith keyword. Therefore, the cost of answering Q on data $\mathcal{D}$ with a set of materialized views $\mathcal{V}$ using Algorithm 1 is

$$cost_{SLCA}(Q, \mathcal{D}, \mathcal{V}) = O(|V_{min}(\mathcal{D})| \cdot \sum_{V_i \in \mathcal{V}} \log|V_i(\mathcal{D})|), \tag{1}$$

which is bounded by $O(|V_{min}(\mathcal{D})||\mathcal{V}|\log|V_{max}(\mathcal{D})|)$, where V_{min} and V_{max} are the smallest set and biggest set, respectively, among all materialized views in the answering set together with the sets of keyword matches retrieved from the full-text index. In other words, the cost of SLCA computation using a set of views is proportional to the product of the size (i.e., number of SLCAs) of the smallest view and the summation of the logarithm of the sizes of the remaining views. This is achievable because the number of SLCAs is bounded by the size of the smallest view.

4.3. Finding Answering Set

In Section 3 we have discussed the problem of selecting the minimal answering set with respect to a general query result definition for XML keyword queries, which is NP-complete. Now we discuss for SLCA semantics, whose time complexity of answering queries using views is given in Eq. (1), how to select the *optimal* answering set of a query which has *the minimal cost of processing the query* among all answering sets. It is easy to see that this problem is also NP-complete, as in the special case when each materialized view has the same size, the goal becomes finding the smallest number of views, which is the same as the minimal answering set problem (Definition 3.4).

Since this problem bears similarity with weighted set cover, a natural idea is to use an algorithm that is similar to the greedy algorithm of weighted set cover, that is, always selecting the subset with the highest benefit-cost ratio. The greedy algorithm of

[4]Specifically, our implementation adopts the Indexed Lookup Eager algorithm, as it generally outperforms Scan Eager and stack algorithms.
[5]Assuming XML tree depth is a constant.

set cover has an approximation ratio of ln$n$+1 (n is the number of items to be covered), which is the best possible approximation ratio unless P = NP.

However, the cost functions of these two problems are different: for weighted set cover, the cost is simply the summation of the costs of all selected subsets, while for the optimal answering set problem, the cost of computing SLCAs is the product of the size of smallest view and the summation of the logarithm of the sizes of the remaining views (Eq. (1)). Is it possible to design an algorithm for the optimal answering set problem that achieves a similar approximation ratio as the set cover problem?

In fact, despite the different cost functions, there is a greedy algorithm for selecting the answering set, which is shown (with nontrivial proofs) to have an approximation ratio of $2(\ln|Q|+1)$.

First, we present the algorithm of selecting the optimal answering set from a set of materialized views $\mathcal{V}$ for query Q on data $\mathcal{D}$, as shown in Algorithm 2.[6] Let $\mathcal{V}_Q$ be the set of relevant materialized views of Q in $\mathcal{V}$, *answerSet* be the set of views selected from $\mathcal{V}_Q$ to form an answering set, *currQ* record the remaining keywords in Q that are not in *answerSet*. We set $V.cost = \log|V(\mathcal{D})|$, where $|V(\mathcal{D})|$ denotes the size of the materialized view V.

As shown in Eq. (1), the cost of answering Q on data $\mathcal{D}$ with a set of materialized views $\mathcal{V}$ is dominated by the size of the minimal view, so it is crucial to first pick the relevant materialized view $V \in \mathcal{V}_Q$ whose size $|V(\mathcal{D})|$ is the smallest, that is, $V.cost$ is minimal among all relevant views to minimize the processing cost (line 5).

We then use a greedy algorithm to select the rest of the views, which is similar as the greedy algorithm for weighted set cover: always selecting the view with the highest ratio of the number of new keywords covered and the cost of the view (lines 12–21). Finally, for each keyword k that cannot be covered by any subquery of Q in $\mathcal{V}$, we add a view containing only this keyword into the answering set (lines 22–24). The result of this view can be retrieved from the inverted index.

After selecting a set of relevant views *answerSet*, we invoke Algorithm 1 to compute the results of Q using views and full-text index if necessary.

Example 4.5. Continuing Example 4.4, we have $\mathcal{V}_Q$ as $\{V_1, V_2, V_3, V_4\}$, and initialize $currQ = \{A, B, C, D, E\}$, and $answerSet = \emptyset$. Suppose their costs (the size of the materialized views) are 100, 60, 80, and 20, respectively. Since V_4 has the minimal cost, it is selected by updating $answerSet = \{V_4\}$. Meanwhile, we refresh the $currQ$ and $cover$ value for other views by setting $currQ = \{A, C, E\}$, $V_3.cover = V_4.cover = 0$, $V_1.cover = 1$, and $V_2.cover = 2$. Next step, we choose the view V_2 since it has the largest $benefit$: $V_2.cover/V_2.cost = 1/30$, and update $answerSet = \{V_2, V_4\}$, $currQ = \{E\}$. Now, no query in $\mathcal{V}_Q$ covers any keyword in $currQ$. Since keyword E has not been covered, we add a view V_E which contains a single keyword E into *answerSet*. The final answering set contains three views $\{V_2, V_4, V_E\}$.

To compute $SLCA(\mathcal{D}, Q)$, we invoke Algorithm 1. We first compute $SLCA(\mathcal{D}, Q) = SLCA(\mathcal{D}, SLCA(\mathcal{D}, V_2), SLCA(\mathcal{D}, V_4))$. Then we access the data for the matches to keyword E in $currQ$, and update $SLCA(\mathcal{D}, Q) = SLCA(\mathcal{D}, SLCA(\mathcal{D}, Q)$, matches of $E)$.

Note that although the algorithm looks similar to the greedy algorithm for the weighted set cover problem, the approximation ratio of the greedy algorithm for weighted set cover, ln$n$+1, does *not* apply to Algorithm 2 due to their different cost functions. Now an immediate question is whether Algorithm 2 has an approximation bound. We prove that the cost of answering a query Q using the answering set

[6]Although the processing cost in Eq. (1) involves both materialized views and keyword match lists, we only need to discuss how to select views to compose an answering set since keyword match lists are determined directly by $\bigcup(V \mid V \in \mathcal{V}) \cap Q$.

Algorithm 2: Find answering set for a Query

```
Input: keyword query $Q$, set of materialized views $\mathcal{V}$
Output: answering set of $Q$
 1: $currQ$ = the set of keywords in $Q$ {$currQ$ records the set of uncovered keywords in $Q$}
 2: $answerSet = \emptyset$ {$answerSet$ records the set of selected materialized views}
 3: $\mathcal{V}_Q$ is a subset of $\mathcal{V}$ consisting of subqueries of $Q$
 4: for $V \in \mathcal{V}_Q$, set $V.cover = |V|$, $V.cost = log|V(\mathcal{D})|$, $V.benefit = V.cover/V.cost$
 5: select $V$ such that $V.cost$ is minimal over all views in $\mathcal{V}_Q$
 6: for each keyword $k \in V$ do
 7:    for each view $V' \in \mathcal{V}_Q$ that contains $k$ do
 8:       $V'.cover = V'.cover - 1$
 9:       $V'.benefit = V'.cover/V'.cost$
10: $answerSet = answerSet \cup Q'$
11: $currQ = currQ$ - $V$
12: while $currQ \neq \emptyset$ do
13:    select $V$ such that $V.benefit$ is maximal over all views in $\mathcal{V}_Q$
14:    if $V.benefit$=0 then
15:       break
16:    for each keyword $k \in currQ \cap V$ do
17:       for each view $V' \in \mathcal{V}_Q$ that contains $k$ do
18:          $V'.cover = V'.cover$-1
19:          $V'.benefit = V'.cover/V'.cost$
20:    $answerSet = answerSet \cup V$
21:    $currQ = currQ$ - $V$
22: for each keyword $k$ in $currQ$ do
23:    $V$ = a view containing a single keyword $k$
24:    $answerSet = answerSet \cup V$
25: return $answerSet$
```

returned by Algorithm 2 is no more than 2(ln$|Q|$+1) times of the cost of answering Q using the optimal answering set.

THEOREM 4.6. *Algorithm 2 finds the optimal answering set with an approximation ratio of* $2(ln|Q|+1)$*, where* $|Q|$ *is the number of keywords in* Q.

PROOF. We use OPT to denote the cost of answering the query using the optimal answering set $\mathcal{V}_{opt}$, and APP to denote the cost of answering the query using the answering set $\mathcal{V}_Q$ found by Algorithm 2. Suppose $\mathcal{V}_{opt}$ consists of p views: $S = \{V_1...V_p\}$, where V_1 is the smallest materialized view in S; and $\mathcal{V}_Q$ has q views: $\mathcal{V} = \{V'_1...V'_q\}$, where V'_1 is the smallest materialized view in $\mathcal{V}_Q$. So we have $OPT = |V_1| \cdot \sum_{i=2}^{p} \log|V_i|$, and $APP = |V'_1| \cdot \sum_{i=2}^{q} \log|V'_i|$.

Consider an instance Ins of the Weighted Set Cover (WSC) problem: a universe $\mathcal{U}$ consists of all keywords in Q, and a collection $\mathcal{S}$ consists of subsets of $\mathcal{U}$, $\forall S \in \mathcal{S}$, $S \subseteq \mathcal{U}$. Each $S \in \mathcal{S}$ is associated with a cost c_S which is a positive real number. In Ins, each S is a view $V \in \mathcal{V}_Q$, and $c_V = V(\mathcal{D})$. WSC is the problem of finding a collection $\mathcal{S}' \subseteq \mathcal{S}$, such that $\bigcup(S \mid S \in \mathcal{S}') = \mathcal{U}$ and $\sum_{S \in \mathcal{S}'} c_S \leq l$. Suppose for Ins, the optimal cost is OPT_{Ins}.

Let Ins' be another instance of WSC, in which each keyword $k \in V_1'$ is removed from the universal set $\mathcal{U}$, and all other settings are the same as Ins. Let the optimal cost of Ins' be $OPT_{Ins'}$.

Since V_1 is the smallest materialized view in $\mathcal{V}_{opt}$, we have

$$\sum_{i=2}^{p} \log|V_i| \geq \frac{p-1}{p} \cdot \sum_{i=1}^{p} \log|V_i| \geq \frac{p-1}{p} OPT_{Ins}$$

which means

$$OPT \geq |V_1| \cdot \frac{p-1}{p} \cdot OPT_{Ins}. \tag{2}$$

On the other hand, after we choose the smallest materialized view V_1' from $\mathcal{V}_Q$ in Algorithm 2, the problem has become exactly WSC' described before. The procedure of choosing $V_2' ... V_q'$ in Algorithm 2 has an approximation ratio of $\ln|Q|+1$ [Cormen et al. 2001]. Therefore,

$$\begin{aligned}\sum_{i=2}^{q} \log|V_i'| &\leq OPT_{Ins'} \cdot (\ln|Q| + 1) \\ &\leq OPT_{Ins} \cdot (\ln|Q| + 1)\end{aligned}$$

which means

$$APP \leq |V_1'| \cdot OPT_{Ins} \cdot (\ln|Q| + 1). \tag{3}$$

According to Algorithm 2, V_1' is the smallest materialized view in $\mathcal{V}_Q$, we have $|V_1| \geq |V_1'|$. When $p = 1$, there is only one relevant view and Algorithm 2 finds it as the optimal solution. When $p \geq 2$, from Eq. (2) and Eq. (3) we have

$$\frac{APP}{OPT} \leq 2(\ln|Q| + 1).$$

□

To improve the performance of Algorithm 2, we find that there are several common operations, including finding whether a view is materialized, and finding all the views that cover a given keyword. We build a *view inverted index* that maps a word to a sorted list of IDs of the views that contain this word. A *view existence index* maps a query to a boolean value, denoting whether the result of the query is materialized or not.

Now we analyze the complexity of Algorithm 2. In line 3, we find all subqueries of Q that are materialized, which takes $O(\min\{|\mathcal{V}|, 2^{|Q|}\})$, where $|\mathcal{V}|$ is the number of materialized views. The while-loop in lines 12–21 is similar to the greedy algorithm for the weighted set cover problem [Cormen et al. 2001], which takes $O(\min\{|\mathcal{V}|, 2^{|Q|}\}|Q|^2)$. Lines 22–24 take $O(|Q|)$ time. The total time complexity of Algorithm 2 is therefore $O(\min\{|\mathcal{V}|, 2^{|Q|}\}|Q|^2)$.

5. INCREMENTALLY MAINTAINING VIEWS

For materialized views to be useful, they need to keep fresh upon data updates. On text documents, this is a relatively straightforward task, as newly added/removed documents can simply be added/removed from the views. On the other hand, it is much more difficult for XML. As to be shown in this section, when a new subtree is inserted into the original data, not only can new SLCA emerge from the inserted subtree, but some original XML nodes can become new SLCA or be disqualified as new SLCA. Similarly, when a subtree is removed from the data, some original nodes may become new

SLCA and some old SLCA may no longer qualify. It is challenging to design algorithms for efficient view maintenance for XML keyword queries that significantly outperform recomputation from scratch. In this section, we discuss how to incrementally maintain materialized views of XML keyword queries upon insertion and deletion of a subtree with respect to the SLCA-based query result definition.

Two update operations on XML data are supported. It is easy to see that any update to the data can be accomplished by a sequence of these primitive update operations.

(1) $Insert(n, n', \mathcal{T})$ denotes an insertion of a tree $\mathcal{T}$ rooted at node n as a child of node n'.
(2) $Delete(n)$ denotes a deletion of node n along with its subtree.

In the following, we use *delta tree* to refer to the inserted or deleted subtree.

5.1. Identifying Affected Views

Not all the views are affected by an XML data update. The first task is to efficiently identify affected views which will be maintained. Recall that a view in SLCA semantics is defined based on the nodes that match keywords, thus a view may be affected by an update if the delta tree contains at least one keyword in the view definition. For a given update, we traverse the delta tree and find all the words that it contains. For each word, we use the view inverted index (introduced in Section 4.3) to find out the list of views whose definitions contain this word, that is, the views that are affected by this update.

Next we discuss how to maintain an affected view for data insertion (Section 5.2) and data deletion (Section 5.3), respectively.

5.2. Incremental View Maintenance upon Insertion

Consider the insertion of a subtree rooted at node n to a node n' in an XML tree, $Insert(n, n', \mathcal{T})$. Suppose the original data is $\mathcal{D}_1$, the updated data is $\mathcal{D}_2$, and the inserted subtree is $\mathcal{T}$. As discussed in Section 1, we must address a unique challenge of maintaining materialized views for XML (conjunctive) keyword queries: a data insertion can result in an insertion and/or *a deletion* in the view.

The algorithm of maintaining the materialized views of keyword queries upon a subtree insertion is presented in Algorithm 3. We differentiate two cases.

Case 1. The inserted subtree $\mathcal{T}$ contains all keywords in a view definition Q (lines 5–10 of Algorithm 3). Since there must exist nodes in $\mathcal{T}$ that are the SLCA nodes of Q, such an insertion qualifies new SLCA nodes. We apply the algorithm proposed in XKSearch on $\mathcal{T}$ to compute $SLCA(\mathcal{T}, Q)$, and have $SLCA(\mathcal{T}, Q) \subseteq SLCA(\mathcal{D}_2, Q)$.

Example 5.1. Consider $\mathcal{D}_2$ in Figure 4, obtained after an insertion of a subtree $\mathcal{T}$ rooted at node $A(0.1.0.2)$ to the original tree $\mathcal{D}_1$. Let a query Q be $\{A, D\}$. We have node $SLCA(\mathcal{D}_1, Q) = \{0.1.0, 0.1.1\}$. Since $\mathcal{T}$ contains both keywords in Q, we have node $A(0.1.0.2) \in SLCA(\mathcal{D}_1, Q)$ as a new SLCA.

On the other hand, $SLCA(\mathcal{T}, Q)$ may disqualify some existing SLCA nodes (lines 7–8). Specifically, if a node in $SLCA(\mathcal{T}, Q)$ is a descendant of an SLCA node $s \in SLCA(\mathcal{D}_1, Q)$, then s is no longer an SLCA on the updated data $\mathcal{D}_2$. To find such s node (if exists), instead of checking every node in $SLCA(\mathcal{T}, Q)$ with respect to $SLCA(\mathcal{D}_1, Q)$, it is equivalent to check n. To see that, if n is a descendant of $s \in SLCA(\mathcal{D}_1, Q)$, then for every node $s_{\mathcal{T}} \in SLCA(\mathcal{T}, Q)$, $s_{\mathcal{T}}$ is a descendant of s. On the other hand, if node $s_{\mathcal{T}}$ is a descendant of s, then n must be a descendant of s, as s is not a node in the tree rooted at n.

Algorithm 3: View Maintenance upon Data Insertion

```
1: input: query $Q$, $SLCA(\mathcal{D}_1, Q)$, original data $\mathcal{D}_1$, $Insert(n, n', \mathcal{T})$, new data $\mathcal{D}_2$, delta tree $\mathcal{T}$
2: output: $SLCA(Q, \mathcal{D}_2)$
3: let $k_1, \ldots, k_p$ be keywords in $Q$ that are not contained in $\mathcal{T}$
4: $SLCA(\mathcal{D}_2, Q)$=$SLCA(\mathcal{D}_1, Q)$
5: if $\mathcal{T}$ contains all keywords then
6:     SLCAT = computeSLCA($\mathcal{T}$, $Q$)
7:     if $n$ is a descendant of $s \in SLCA(\mathcal{D}_1, Q)$ then
8:         $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) - \{s\}$
9:     $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) \cup SLCAT$
10:    return $SLCA(\mathcal{D}_2, Q)$
11: $currLCA = n$
12: for $j = 1$ to $p$ do
13:     $currLCA$ = the ancestor of $currLCA$ and $lowest(k_j, currLCA, \mathcal{D}_1)$
14: if $currLCA$ is not an ancestor of any $s \in SLCA(\mathcal{D}_1, Q)$ then
15:     $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) \cup currLCA$
16:     if $currLCA$ is a descendant of $s$ then
17:         $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) - \{s\}$
18: return $SLCA(\mathcal{D}_2, Q)$
```

lowest **($k_j, r, \mathcal{D}_1$)**

```
1: {$lowest(k_j, r, \mathcal{D}_1)$ is lowest ancestor-or-self of node $r$ in $\mathcal{D}_1$ that contains a match to keyword $k_j$.}
2: $lm = leftMatch(k_j, r, \mathcal{D}_1)$
3: $rm = rightMatch(k_j, r, \mathcal{D}_1)$
4: return the lower one of $LCA(lm, r)$ and $LCA(rm, r)$
```

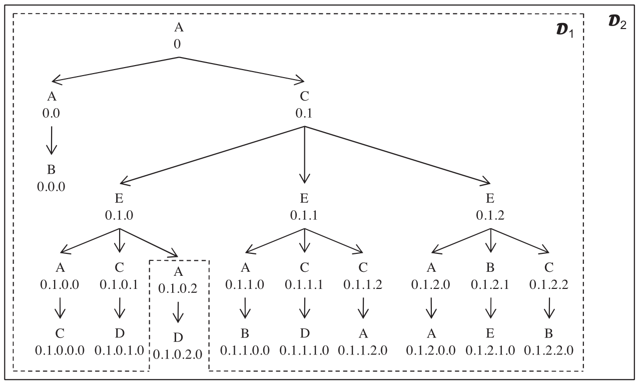


Fig. 4. XML trees $\mathcal{D}_1$ and $\mathcal{D}_2$.

To efficiently find the ancestor and descendant node relationships, we preprocess XML data and assign each XML node n a *Dewey* label [Tatarinov et al. 2002] as a unique ID, referred to as $Dewey(n)$. 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. Dewey ID can be used to detect the relationship between nodes. A node n_1 is an ancestor of node n_2 if and only if $Dewey(n_1)$ is a prefix of $Dewey(n_2)$. This indicates that the LCA of nodes n_1 and n_2 has the Dewey ID which is the longest common prefix of $Dewey(n_1)$ and $Dewey(n_2)$. Besides, we say $Dewey(n_1)$ is smaller than $Dewey(n_2)$ if n_1 appears before n_2 in document order.[7]

Since SLCA nodes do not have ancestor-descendant relationship, there is at most one ancestor of n in $SLCA(\mathcal{D}_1, Q)$, which can be found efficiently according to the following proposition.

PROPOSITION 5.2. *If there is a node $s \in SLCA(\mathcal{D}, Q)$ that is an ancestor of n, then s has the largest Dewey ID that is smaller than $Dewey(n)$ among all the nodes in $SLCA(\mathcal{D}, Q)$.*

PROOF. Suppose s does not have the largest Dewey ID that is smaller than $Dewey(n)$ among all nodes in $SLCA(\mathcal{D}, Q)$, that is, there is a node $s' \in SLCA(\mathcal{D}, Q)$, such that $Dewey(s) < Dewey(s') < Dewey(n)$. Since $Dewey(s)$ is a prefix of $Dewey(n)$, it is easy to see that $Dewey(s)$ must also be a prefix of $Dewey(s')$, and therefore s is an ancestor of s'. However, since SLCA nodes do not have ancestor-descendant relationship, this is impossible, thus s must have the largest Dewey ID that is smaller than $Dewey(n)$. □

Symmetrically, we have the following proposition which will be used later in this section.

PROPOSITION 5.3. *If there are nodes in $SLCA(\mathcal{D}, Q)$ that are descendants of n, then node s with the smallest Dewey ID that is larger than n among all the nodes in $SLCA(\mathcal{D}, Q)$ is a descendant of n.*

PROOF. According to the definition of Dewey ID, each node in n's subtree (i.e., a descendant of n) has a Dewey ID larger than n's Dewey ID. $\forall s, s'$, such that both s and s''s Dewey IDs are larger than n's Dewey ID, and s is a descendant of n while s' is not, then obviously s' is behind s in document order. Therefore, if n has a descendant in a set S of nodes, and s is the node in S that has the smallest Dewey ID larger than n, s must be a descendant of n. □

The propositions show that to find the ancestor (descendant) of a given node n in $SLCA(\mathcal{D}, Q)$, we only need to search node s with the largest Dewey ID that is smaller than $Dewey(n)$ (with the smallest Dewey ID that is larger than $Dewey(n)$), then check if s is an ancestor (descendant) of n. Since $SLCA(\mathcal{D}, Q)$ is sorted by Dewey ID, this can be done by a binary search.

Example 5.4. Continuing Example 5.1, for the newly found node in $SLCA(\mathcal{T}, Q)$: 0.1.0.2, we find the largest Dewey ID in $SLCA(\mathcal{D}_1, Q)$ that is smaller than 0.1.0.2: 0.1.0. Since node 0.1.0 is an ancestor of node 0.1.0.2, it no longer qualifies to be an SLCA. We remove it and have $SLCA(\mathcal{D}_2, Q) = \{0.1.0.2, 0.1.1\}$.

Case 2. Now let us consider an inserted subtree $\mathcal{T}$ that does not contain all keywords in a view definition Q. According to the discussion in Section 5.1, we can find the keywords in Q that are not contained in $\mathcal{T}$, denoted as $k_1, \cdots, k_p$. Three steps need to be performed to compute $SLCA(\mathcal{D}_2, Q)$, as shown in Algorithm 3.

First, we find the lowest ancestor of n that contains matches to keywords $k_1, \cdots, k_p$ in its subtree, denoted as $currLCA$, which is a potential new SLCA (lines 12–13). We

[7]Note that upon a data update, the Dewey labels of some nodes may be affected. Efficient maintenance of Dewey labeling is orthogonal to this article. It has been investigated in the literature and can be achieved without relabeling [Cohen et al. 2002; Li et al. 2006; O'Neil et al. 2004; Xu et al. 2009].

define $lowest(k_j, n, \mathcal{D})$ as the lowest ancestor of n in XML data $\mathcal{D}$ that contains a match to keyword k_j. As shown in Xu and Papakonstantinou [2005], $lowest(k_j, n, \mathcal{D})$ must be either the LCA node of $leftMatch(k_j, n, \mathcal{D})$ and n, or the LCA node of $rightMatch(k_j, n, \mathcal{D})$ and n, where $leftMatch(k_j, n, \mathcal{D})$ is the match to k_j in $\mathcal{D}$ with the largest Dewey ID that is smaller than $Dewey(n)$, and $rightMatch(k_j, n, \mathcal{D})$ is the match to k_j with the smallest Dewey ID that is larger than $Dewey(n)$. Since the list of nodes matching a keyword are sorted by their Dewey ID in the full-text index, $leftMatch(k_j, n, \mathcal{D})$ and $rightMatch(k_j, n, \mathcal{D})$ can be found efficiently using binary search. Initially, we set $currLCA = n$. For each keyword k_j from k_1 to k_p, if $currLCA$ is a descendant of $lowest(k_j, n, \mathcal{D})$, then we set $currLCA = lowest(k_j, n, \mathcal{D})$. Finally, $currLCA$ is the lowest ancestor of n that contains matches to all the keywords.

Example 5.5. Consider $\mathcal{D}_1$ and $\mathcal{D}_2$ in Figure 4. Let query $Q = \{B, D, E\}$. $SLCA(\mathcal{D}_1, Q) = \{0.1.1\}$. Since $\mathcal{T}$ does not contain keywords B and E, we need to find the lowest ancestor of node 0.1.0.2 that contains keywords B and E in its subtree. The list of nodes in $\mathcal{D}_1$ that match keyword B is: {0.0.0, 0.1.1.0.0, 0.1.2.1}. We find $leftMatch(B, 0.1.0.2, \mathcal{D}_1)$ to be 0.0.0, and $rightMatch(B, 0.1.0.2, \mathcal{D}_1)$ to be 0.1.1.0.0. Therefore $lowest(B, 0.1.0.2, \mathcal{D}_1) = LCA(0.1.1.0.0, 0.1.0.2) = 0.1$. Similarly, we find the lowest ancestor of 0.1.0.2 that contains E, which is 0.1.0. The lowest ancestor of 0.1.0.2 that contains all the keywords in Q is therefore node 0.1.

Next, given $currLCA$, the lowest ancestor of n that contains matches to all keywords in Q, we need to check whether $currLCA$ qualifies to be a node in $SLCA(\mathcal{D}_2, Q)$ or not (line 14). Specifically, if there exists a node $s \in SLCA(\mathcal{D}_1, Q)$, such that $currLCA$ is an ancestor of s, then $currLCA$ is disqualified. This can be checked according to Proposition 5.3.

Example 5.6. Continuing Example 5.5, we check whether $currLCA = 0.1$ is an ancestor of a node in $SLCA(\mathcal{D}_1, Q) = \{0.1.1\}$, which is indeed the case. Therefore $currLCA$ is not a new SLCA, and $SLCA(\mathcal{D}_2, Q) = \{0.1.1\}$.

Finally, if $currLCA$ is identified as a node in $SLCA(\mathcal{D}_2, Q)$, then we need to further check whether any existing SLCA node should be removed (lines 16–17). This is done by checking whether any existing SLCA node is an ancestor of $currLCA$ using Proposition 5.2.

Now we analyze the complexity of Algorithm 3. If $\mathcal{T}$ contains all keywords, the time complexity is $M_{\mathcal{T}min}|Q|d\log M_{\mathcal{T}max}$, where $M_{\mathcal{T}min}$ and $M_{\mathcal{T}max}$ are the minimum and maximum number of matches to a keyword in delta tree $\mathcal{T}$, respectively, and d is the depth of $\mathcal{T}$. If $\mathcal{T}$ does not contain all keywords, then the running time of lines 12–13 of Algorithm 3 is $O(|Q|d\log M_{max})$, where M_{max} is the largest number of matches to a keyword of Q in $\mathcal{D}_1$. The running time of line 14 and 16 is $O(\log M_{min})$, where M_{min} is the minimum number of matches in the whole XML tree to a keyword in $\mathcal{D}_1$, which is the upper bound of $|SLCA(\mathcal{D}_1, Q)|$. So the time complexity when $\mathcal{T}$ does not contain all keywords in Q is $O(|Q|d\log M_{max})$. Therefore, the overall complexity of Algorithm 3 is $O(\max\{M_{Tmin}|Q|d\log M_{Tmax}, |Q|d\log M_{max}\})$.

5.3. Incremental View Maintenance upon Deletion

Now let us consider a deletion of a subtree $\mathcal{T}$ rooted at node n from the XML data, $Delete(n)$, which can result in an insertion and/or a deletion in a materialized view. Let $\mathcal{D}_1$ be the original data and $\mathcal{D}_2$ be the updated data. According to the discussion in Section 5.1, we can efficiently find the keywords $k_1, \cdots, k_p$ that are contained in the deleted tree $\mathcal{T}$.

Algorithm 4: View Maintenance upon Data Deletion

```
1: input: query $Q$, $SLCA(\mathcal{D}_1, Q)$, original data $\mathcal{D}_1$, $Delete(n)$, new data $\mathcal{D}_2$, delta tree $\mathcal{T}$
2: output: $SLCA(\mathcal{D}_2, Q)$
3: let $\mathcal{T}$ be the subtree rooted at $n$
4: $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_1, Q)$
5: if $n$ is an ancestor of $s \in SLCA(\mathcal{D}_1, Q)$ then
6:     $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) - \{s\}$
7: Let $k_1, ..., k_p$ be the keywords that are contained in $\mathcal{T}$
8: $currLCA = n$
9: for $j = 1$ to $p$ do
10:     $currLCA$ = the ancestor of $currLCA$ and $lowest(k_j, s, \mathcal{D}_2))$
11: if $currLCA$ is a descendant of $s \in SLCA(\mathcal{D}_2, Q)$ then
12:     $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) - \{s\}$
13: if $currLCA$ is not an ancestor of any $s \in SLCA(\mathcal{D}_2, Q)$ then
14:     $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) \cup currLCA$
15: return $SLCA(\mathcal{D}_2, Q)$
lowest $(k_j, r, \mathcal{D}_2)$
1: {$lowest(r, k_j)$ is the match node of $k_j$ in $\mathcal{D}_2$ that has the lowest LCA with $r$.}
2: $lm = leftMatch(k_j, r, \mathcal{D}_2)$
3: $rm = rightMatch(k_j, r, \mathcal{D}_2)$
4: return the lower one of $LCA(lm, r)$ and $LCA(rm, r)$
```

The algorithm for maintaining a materialized view Q upon a data deletion is presented in Algorithm 4. There are two possible cases considering the relationship of n and an existing SLCA node $s \in SLCA(\mathcal{D}_1, Q)$.

Case 1. If n is neither an ancestor nor a descendant-or-self of any $s \in SLCA(\mathcal{D}_1, Q)$, then the deletion will not affect the query result, that is, $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_1, Q)$.

Case 2. Otherwise, an ancestor of n may potentially become a new SLCA. First we check if n is an ancestor-or-self of a node $s \in SLCA(\mathcal{D}_1, Q)$. If so, we remove all such s from $SLCA(\mathcal{D}_2, Q)$ (lines 5–6). Then we search for the lowest ancestor of n that contains all keywords in Q, denoted as $currLCA$, which is a potential new SLCA. We check if $currLCA$ is a descendant of an existing SLCA in $s \in SLCA(\mathcal{D}_2, Q)$ according to Proposition 5.1. If so, s is removed from $s \in SLCA(\mathcal{D}_2, Q)$ as SLCA nodes do not have ancestor-descendant relationship (lines 11–12). Finally, $currLCA$ is put into the SLCA list $SLCA(\mathcal{D}_2, Q)$ if it does not have any descendant in $SLCA(\mathcal{D}_2, Q)$ (lines 13–14), checked according to Proposition 5.3. This check is necessary, as exemplified in the following example.

Example 5.7. Consider $Q = \{A, C, D\}$ on Figure 4, where $\mathcal{D}_2$ is the original data, $\mathcal{D}_1$ is the updated data with a deletion of the subtree $\mathcal{T}$ rooted at node $n = A(0.1.0.2)$ from $\mathcal{D}_2$. To show this example, suppose that node 0.1.0.1.0 is B rather than D. Notice that n is a descendant of node $s = 0.1.0$ in $SLCA(\mathcal{D}_2, Q) = \{0.1.0, 0.1.1\}$. Since $\mathcal{T}$ contains keywords A, D, we know s contains keyword $Q - \{A, D\} = \{C\}$ in its subtree. $lowest(A, s, \mathcal{D}_1) = 0.1.0$, and $lowest(D, s, \mathcal{D}_1) = 0.1$. So s is no longer an SLCA of Q on $\mathcal{D}_1$, instead, $C(0.1)$ becomes a potential new SLCA. However, since $C(0.1)$ is an ancestor of $E(0.1.1) \in SLCA(\mathcal{D}_1, Q)$, $C(0.1)$ is disqualified to be an SLCA in $SLCA(\mathcal{D}_1, Q)$. Therefore, $SLCA(\mathcal{D}_1, Q) = \{0.1.1\}$.

The complexity of Algorithm 4 is $O(|Q|d\log M_{max})$, analyzed similar as that of Algorithm 3.

6. EXPERIMENTS

For performance evaluation, we compared the proposed techniques that exploit materialized views as much as possible for query evaluation, referred to as *With Views*, with the approach that only uses indexes without views, referred to as *Without Views*. We also compared the performance of incremental view maintenance, referred to as *Incremental Maintenance*, with recomputing materialized views from indexes, referred to as *Recomputation*. The *Without Views*/*Recomputation* approach for query evaluation/view materialization is implemented based on XKSearch.

The experiments were conducted on a machine with 3.0 GHz AMD Athlon(TM) dual-core CPU, 4.0GB memory, running Microsoft Windows Server 2008 Enterprise. The algorithms were implemented in Microsoft Visual C++ 8.0. An inverted full-text index for XML data was built using Oracle Berkeley DB, and is used when a query cannot be (totally) answered by materialized views, as well as for view maintenance.

Two datasets are used in the experiments. A synthetic auction dataset generated by XMark with default schema has a size of up to 1.5GB; and a real-world DBLP dataset has a size of 436MB.[8]

6.1. Exploiting Materialized Views for Evaluating Queries

To evaluate the effectiveness of materialized views in query evaluation, we synthesized a workload of queries following the existing approaches where the distribution of keyword occurrences in the query workload satisfies Zipf-Law and the exponent z is 1 [Feng et al. 2007; Lempel and Moran 2003; Mandhani and Suciu 2005; Saraiva et al. 2007]. We extract all distinct words in tag names and text values in each XML dataset. Each word is given a random rank in the Zipfian distribution. Each test query is generated containing a random number of keywords varying from 2 to 6, where each keyword is randomly selected based on the Zipfian distribution. Since it is an open problem of selecting which views to be materialized for XML keyword queries, we take a baseline approach. According to Section 3, for a given query, only its subqueries can be used as relevant views, thus it is more reasonable to materialize small queries than larger ones. We therefore generate the views randomly in the same way as queries, except that each view contains either 2 or 3 keywords.

Whenever a query has a relevant view, we refer this as a hit to materialized views. For 2000 test queries with 1000 materialized views on Auction data, the overall hit rate is 55.4%. Among this only 3.8% of the queries can find exactly same views, and 6.4% of the queries can be completely answered by views. For DBLP data, the hit rate is 30.8%; 1.8% of the queries can find exactly the same views, and 2.5% of the queries can be completely answered by views (i.e., without using the inverted index). The average lookup time on view inverted indexes as well as the time for finding the best answering set is less than 0.01 second, which is negligible.

Note that the strategy of selecting which views to materialize may impact the efficiency of query processing. A more carefully designed view selection algorithm would decrease the number of index accesses and achieve a larger performance speedup for the whole query workload. To avoid the bias of varying hit rates, in the following we only report and analyze the average processing time for queries that can be partially or fully evaluated by views.

We also found in the experiment that the additional storage needed to store materialized views is negligible compared to the indexes of the source data used in query processing. The total size of the 1000 randomly generated materialized views is only 3.5M, compared to 778M of the size of the indexes of a 216M XML document.

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

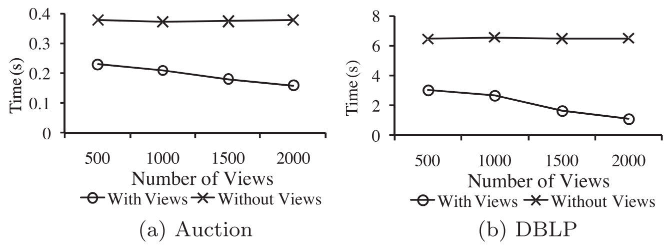


Fig. 5. Average query processing time, varying number of views.

We test the efficiency of exploiting materialized views for query evaluation from three aspects: the number of materialized views, the data size, and the number of keywords in each view. Note that for answering queries using views an alternative way of selecting views is to use an exponential algorithm to find the optimal answering set. However, it is way too slow due to the large sizes of our test data. Therefore, we only test our greedy algorithm for view selection.

(1) *Scalability over Number of Views.* In this test, we increase the number of materialized views from 500 to 2000 on both the Auction dataset of size 216MB and the DBLP dataset. 2000 queries are tested.

Figure 5(a) and (b) shows the average query processing time, varying the number of views. As we can see, if materialized views are exploited, the query processing time significantly decreases when the number of views increases. When more materialized views are available, it is more likely that a query will have relevant views, and therefore fewer accesses to data index are needed and less query evaluation computation is required. Although an increasing number of materialized views entails a larger cost for identifying relevant views and finding the optimal answering set, such performance overhead is imperceptible (< 0.01 second), and the overall performance speedup becomes larger when the number of views increases. On the other hand, the time of query evaluation without leveraging materialized views remains unchanged.

We also observe that the saving on the DBLP dataset is relatively larger than that of Auction. By analyzing its data characteristics, we find that the average number of matches to each word in DBLP is larger than that of Auction, and therefore its query evaluation is much more expensive. On the other hand, the result of a query, even for a 2-keyword query, is very small. For instance, the number of matches to "database" is huge, while the number of results of evaluating a query "database, Levine" is much smaller. Therefore, as long as a query can find a relevant view, its processing time is dramatically reduced.

(2) *Scalability over Data Size.* To test the efficiency of our approach with respect to different data sizes, we test the Auction data with sizes varying from 216MB to 1512MB, by replicating the file multiple times. 1000 materialized views and 2000 queries are randomly generated. The average query processing is shown in Figure 6. As we can see, the processing time of using views and that of from scratch both increase linearly when data size increases. The performance speedup of using views becomes larger when the data size increases.

(3) *The Effect of the Number of Keywords in Each View.* We increase the number of keywords in a materialized view from 2 to 5. We use 2000 queries and 1000 views, and record the average processing time of these queries, which is shown in Figure 7. Figure 7 indicates that the processing time of our approach increases with more numbers of keywords in the views.

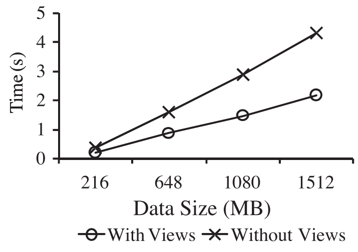


Fig. 6. Average query processing time, varying data size.

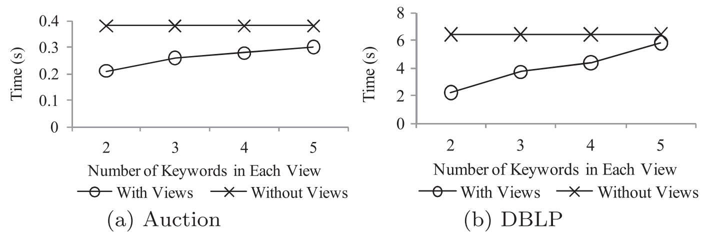


Fig. 7. Average query processing time, varying the number of keywords in each view.

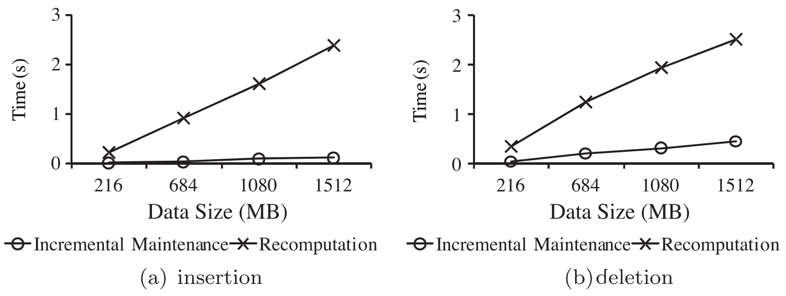


Fig. 8. Average view maintenance time, varying data size.

When the number of keywords in views increases, there can be two situations: (i) a query may have a smaller materialized view (recall that the larger number of keywords, the smaller number of SLCAs and thus the smaller size of the view), which can be utilized to evaluate the query, and the processing time decreases, and (ii) a query may no longer have a materialized relevant view that can be used to answer it or part of it, which means the query or part of the query has to be evaluated by accessing the source data, and the processing time increases. This test shows that factor (ii) is dominant, that is, when the number of keywords in views increases, there are far less chances to find an answering set for the queries.

6.2. Materialized View Maintenance

To test the efficiency of incremental view maintenance upon data update, we test its processing time with respect to different data size and delta tree size.

We randomly generate 1000 views and 10 delta trees, and report the average time required to maintain a single view upon a single updates in Figures 8 and 9. To generate a delta tree, we first find, for each tag name, the average size of its subtree. For example, the average subtree size of `item` is 63. To generate a delta tree of size about 63, we randomly pick a node of tag `item` in the original data and use a copy

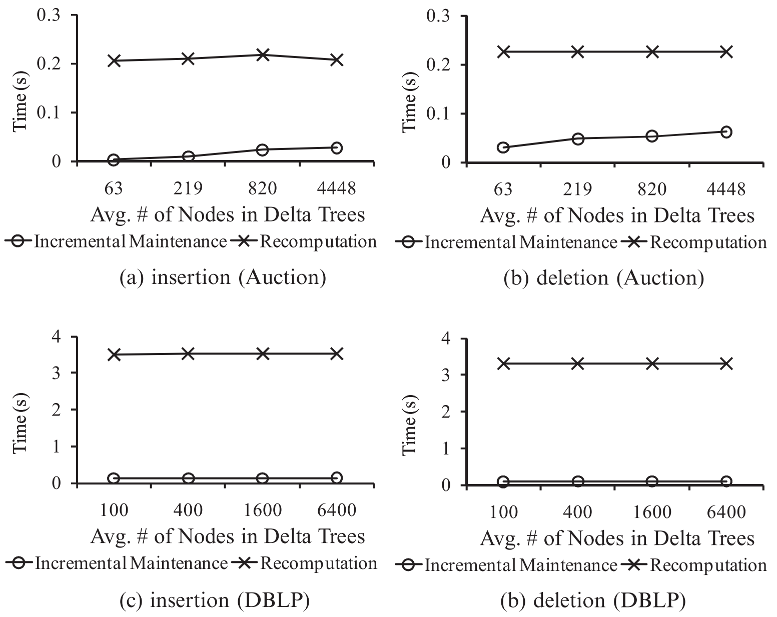
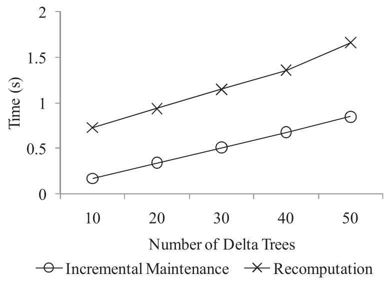


Fig. 9. Average view maintenance time, varying delta tree size.

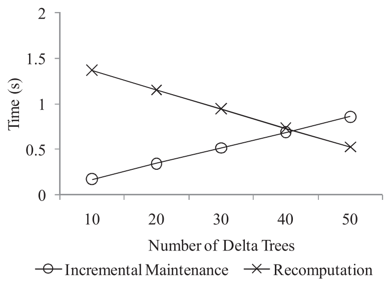
of its subtree as a delta tree. The insertion or deletion position is randomly selected. The processing times shown in the figures do not include the time for maintaining the indexes and the Dewey labels, as these are the same for both approaches, and largely depend on the implementations of indexes and the labeling schemes. In our implementation, we use Berkeley DB for our inverted index, and the index maintenance time for DBLP data with a delta tree of 5000 nodes is roughly 0.1 seconds, which is very short compared with the time for query processing. For the labeling scheme, an extension of Dewey labeling named DDE (Dynamic Dewey) [Xu et al. 2009] is able to handle insertion and deletion on the data without relabeling.

(1) *Efficiency with respect to data size.* In order to test scalability over data size, we replicate the auction data multiple times to increase its size. We choose to replicate a dataset multiple times rather than taking fragments of a large dataset, as it preserves properties of the dataset such as structure and term frequency. We vary the Auction data size from 216MB to 1512MB to test the efficiency of our approach. Delta trees are randomly selected with an average size of 63. As shown in Figure 8(a) and (b), the processing times of incremental view maintenance and view maintenance from scratch both increase linearly with the increase of data size. Incremental maintenance is far more efficient than computing views from scratch, and performance benefits become larger when the data size increases.

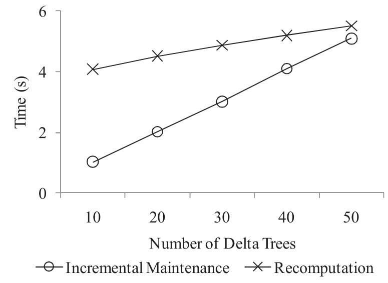
(2) *Efficiency with respect to delta tree size.* The performance of incremental view maintenance and computing views from scratch while varying the average delta tree size from 63 to 4448 nodes on the Auction data is shown in Figure 9(a) and (b). The time of computing from scratch is almost not affected, as the size of the delta tree is small compared to the original data. On the other hand, the time required for incremental view maintenance increases when the size of delta trees increases. This is because a larger delta tree contains more words, therefore more materialized views are likely to be affected, and a higher maintenance cost would be expected. Furthermore,



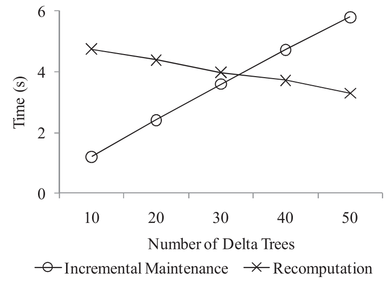
(a) insertion (Auction)



(b) deletion (Auction)



(c) insertion (DBLP)



(b) deletion (DBLP)

Fig. 10. Average view maintenance time, varying the number of delta trees.

for views that have all keywords appear in the delta tree, the number of query results within the delta tree likely increases when the delta tree size increases, and therefore also counts for a longer maintenance time.

(3) *Efficiency with respect to number of delta trees.* The processing time with respect to different number of delta trees are presented in Figure 10. Each delta tree has a size of 5MB; we increase the number of delta trees from 10 to 50, which are inserted to/deleted from random positions in the data.

From Figure 10(a) and (c), we observe that the processing time of both approaches increases almost linearly with the number of delta trees, which is consistent with their time complexities. As expected, when the total delta size approaches the data size, the processing time ratio of recomputation and incremental maintenance becomes smaller. On DBLP data, our approach's processing time increases faster than recomputation because the delta tree size is relatively small compared to the data size. Since we process one delta tree at a time, our processing time mainly depends on the number of delta trees, while the processing time of recomputation mainly depends on the total size of the data and delta trees. From Figure 10(b), we observe that when there are many deleted delta trees, recomputation can be more efficient than incremental maintenance. This is because our approach has to process the delta trees one by one, thus when there are 50 delta trees, the processing time is roughly 50 times longer than processing a single delta tree. On the other hand, since the total delta size is quite large compared to the data size, the remaining data size is relatively small, which makes recomputation from scratch more efficient.

Therefore, there are two cases where recomputation can be faster than incremental maintenance: many small delta trees are inserted, or many delta trees are removed and the total size of the removed delta trees is large. In these cases it is advisable to recompute the views from scratch.

In summary, experiment evaluation shows that our approach is much more efficient in answering XML keyword queries using views and incrementally maintaining views, compared with answering queries and maintaining views from scratch.

7. CONCLUSIONS AND FUTURE WORK

This article addresses an open problem of exploiting and maintaining materialized views for XML keyword queries. We analyze the problem of identifying the best answering set of materialized views to evaluate a given query, which is NP-hard. We present the first XML keyword search engine that can answer queries using materialized views for SLCA semantics. We propose a polynomial-time approximation algorithm for finding a good answering set of a given query from a set of materialized views, and develop the algorithm of answering query using its answering set. For materialized views to be useful for dynamic XML data, we design incremental view maintenance algorithms upon data updates. Our techniques can be incorporated into the XML keyword search systems that adopt SLCA semantics [Liu and Chen 2007, 2008b; Sun et al. 2007; Xu and Papakonstantinou 2005]. Experimental evaluation shows significant performance improvements of our approach over computing query results or views from scratch.

One of our future works is to investigate the strategies to determine which queries should have their results materialized as views according to workload information. A good strategy may significantly increase the cache hit rate and thus improve the efficiency of query processing. For example, one strategy is to select the most frequently asked queries based on the query log. However, recall that views of a subquery of Q are helpful for answering Q. Therefore, for two similar queries which are both frequent, it may be worthwhile to materialize their intersection, rather than materialize queries.

Another future work is to use materialized views to efficiently produce ranked results. It is easy to see that, with monotonic ranking functions (i.e., the score of a result R of query Q monotonically increases with the score increase of the results of Q's subqueries that are used to compute R) our approach can be directly applied for top-k query processing. Specifically, the nodes in each materialized view are sorted by their score, hence the order that query results are generated is indeed the ranked order, that is, top-k results can be efficiently obtained without generating all results. Monotonic ranking factors include TF-IDF, number of keyword matches, etc., which are adopted in several existing search engines [Bao et al. 2009; Cohen et al. 2003; Li et al. 2008]. However, it is an open question how to utilize materialized views to incrementally produce ranked results for XML keyword queries when nonmonotonic ranking factors are used. One approach to generate top-k keyword query results in relational databases with a nonmonotonic scoring function is proposed in Luo et al. [2007], which finds a monotonic function that is an upper bound of the nonmonotonic scoring function, uses the former for incrementally generating results and then ranks the results based on the actual scoring function. We plan to investigate whether and how this idea can be applied for XML keyword queries.

REFERENCES

Arion, A., Benzaken, V., Manolescu, I., and Papakonstantinou, Y. 2007. Structured materialized views for xml queries. In *Proceedings of the International Conference on Very Large Databases (VLDB'07)*.

Balmin, A., Ozcan, F., Beyer, K. S., and Cochrane, R. J. 2004. A framework for using materialized xpath views in xml query processing. In *Proceedings of the International Conference on Very Large Databases (VLDB'04)*.

BAO, Z., LING, T. W., CHEN, B., AND LU, J. 2009. Effective xml keyword search with relevance oriented ranking. In *Proceedings of the International Conference on Data Engineering (ICDE'09)*.

CHEN, L. J. AND PAPAKONSTANTINOU, Y. 2010. Supporting top-k keyword search in xml databases. In *Proceedings of the International Conference on Data Engineering (ICDE'10)*.

CHEN, Y., WANG, W., LIU, Z., AND LIN, X. 2009. Keyword search on structured and semi-structured data. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. 1005–1010.

CHEN, Y., WANG, W., AND LIU, Z. 2011. Keyword-Based search and exploration on databases. In *Proceedings of the International Conference on Very Large Databases (ICDE'11)*. 1380–1383.

COHEN, E., KAPLAN, H., AND MILO, T. 2002. Labeling dynamic xml trees. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'02)*.

COHEN, S., MAMOU, J., KANZA, Y., AND SAGIV, Y. 2003. XSEarch: A semantic search engine for xml. http://www.vldb.org/conf/2003/papers/S03P02.pdf.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms* 2nd Ed. The MIT Press.

FAN, W., GEERTS, F., JIA, X., AND KEMENTSIETSIDIS, A. 2007. Rewriting regular xpath queries on xml views. In *Proceedings of the International Conference on Data Engineering (ICDE'07)*.

FENG, J., TA, N., ZHANG, Y., AND LI, G. 2007. Exploit sequencing views in semantic cache to accelerate xpath query evaluation. In *Proceedings of the International Conference on World Wide Web (WWW'07)*.

GUO, L., SHAO, F., BOTEV, C., AND SHANMUGASUNDARAM, J. 2003. XRANK: Ranked keyword search over xml documents. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

HRISTIDIS, V., KOUDAS, N., PAPAKONSTANTINOU, Y., AND SRIVASTAVA, D. 2006. Keyword proximity search in xml trees. *IEEE Trans. Knowl. Data Engin. 18*, 4.

HUANG, Y., LIU, Z., AND CHEN, Y. 2008. Query biased snippet generation in xml search. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

LEMPEL, R. AND MORAN, S. 2003. Predictive caching and prefetching of query results in search engines. In *Proceedings of the International Conference on World Wide Web (WWW'03)*.

LI, C., LING, T. W., AND HU, M. 2006. Efficient processing of updates in dynamic xml data. In *Proceedings of the International Conference on Data Engineering (ICDE'06)*.

LI, G., FENG, J., WANG, J., AND ZHOU, L. 2007a. Effective keyword search for valuable lcas over xml documents. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM'07)*.

LI, G., OOI, B. C., FENG, J., WANG, J., AND ZHOU, L. 2008. EASE: Efficient and adaptive keyword search on unstructured, semi-structured and structured data. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

LI, Y., YANG, H., AND JAGADISH, H. V. 2007b. NaLIX: A generic natural language research environment for xml data. *ACM Trans. Datab. Syst. 32*, 4.

LI, Y., YU, C., AND JAGADISH, H. V. 2004. Schema-Free xquery. In *Proceedings of the International Conference on Very Large Databases (VLDB'04)*.

LIU, Z. AND CHEN, Y. 2007. Identifying meaningful return information for xml keyword search. In *Proceedings of the ACM Conference on Management of Data*.

LIU, Z. AND CHEN, Y. 2008a. Answering keyword queries on xml using materialized views. In *Proceedings of the International Conference on Data Engineering (ICDE'08)*.

LIU, Z. AND CHEN, Y. 2008b. Reasoning and identifying relevant matches for xml keyword search. In *Proceedings of the International Conference on Very Large Databases (VLDB'08)*.

LIU, Z. AND CHEN, Y. 2010. Return specification interference and result clustering for keyword search on xml. *ACM Trans. Datab. Syst. 35*, 2.

LIU, Z. AND CHEN, Y. 2011. Processing keyword search on xml: A survey. *World Wide Web 14*, 5–6, 671–707.

LIU, Z. AND CHEN, Y. 2012. Differentiating search results on structured data. *ACM Trans. Datab. Syst. 37*, 1, 4.

LIU, Z., HUANG, Y., AND CHEN, Y. 2010a. Improving xml search by generating and utilizing informative result snippets. *ACM Trans. Datab. Syst. 35*, 3.

LIU, Z., SHAO, Q., AND CHEN, Y. 2010b. Searching workflows with hierarchical views. *Proc. VLDB 3*, 1, 918–927.

LIU, Z., NATARAJAN, S., AND CHEN, Y. 2011. Query expansion based on clustered results. *Proc. VLDB 4*, 6, 350–361.

LUO, Y., LIN, X., WANG, W., AND ZHOU, X. 2007. SPARK: Top-k keyword query in relational databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

MANDHANI, B. AND SUCIU, D. 2005. Query caching and view selection for xml databases. In *Proceedings of the International Conference on Very Large Databases (VLDB'05)*.

O'NEIL, P., ONEIL, E., PAL, S., CSERI, I., AND SCHALLER, G. 2004. ORDPATHs: Insert-Friendly xml node labels. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

ONOSE, N., DEUTSCH, A., PAPAKONSTANTINOU, Y., AND CURTMOLA, E. 2006. Rewriting nested xml queries using nested views. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

SARAIVA, P.-C., DE MOURA, E. S., ZIVIANI, N., MEIRA, W., FONSECA, R., AND RIBEIRONETO, B. 2007. Rank-Preserving two-level caching for scalable search engines. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*.

SAWIRES, A., TATEMURA, J., PO, O., AGRAWAL, D., AND CANDAN, K. S. 2005. Incremental maintenance of path-expression views. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

SAWIRES, A., TATEMURA, J., PO, O., AGRAWAL, D., ABBADI, A. E., AND CANDAN, K. S. 2006. Maintaining xpath views in loosely coupled systems. In *Proceedings of the International Conference on Very Large Databases (VLDB'06)*.

SHAO, F., GUO, L., AND BOTEV, C. 2007. Efficient keyword search over virtual xml views. In *Proceedings of the International Conference on Very Large Databases (VLDB'07)*.

SUN, C., CHAN, C.-Y., AND GOENKA, A. 2007. Multiway slca-based keyword search in xml data. In *Proceedings of the International Conference on World Wide Web (WWW'07)*.

TANG, N., YU, J. X., OZSU, M. T., CHOI, B., AND WONG, K.-F. 2008. Multiple materialized view selection for xpath query rewriting. In *Proceedings of the International Conference on Data Engineering (ICDE'08)*.

TATARINOV, I., VIGLAS, S., BEYER, K. S., SHANMUGASUNDARAM, J., SHEKITA, E. J., AND ZHANG, C. 2002. Storing and querying ordered xml using a relational database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

XU, L., LING, T. W., WU, H., AND BAO, Z. 2009. DDE: From dewey to a fully dynamic xml labeling scheme. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

XU, W. AND OZSOYOGLU, Z. M. 2005. Rewriting xpath queries using materialized views. In *Proceedings of the International Conference on Very Large Databases (VLDB'05)*.

XU, Y. AND PAPAKONSTANTINOU, Y. 2005. Efficient keyword search for smallest lcas in xml databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*.

XU, Y. AND PAPAKONSTANTINOU, Y. 2008. Efficient lca based keyword search in xml data. In *Proceedings of the International Conference on Extending Database Technology (EDBT'08)*.

Received June 2009; revised September 2011; accepted October 2012