Keyword Search on DAG-Compressed XML Data

Stefan Böttcher, Marc Brandenburg, Rita Hartel
University of Paderborn (Germany), Computer Science,
Fürstenallee 11,
D-33102 Paderborn
stb@uni-paderborn.de, mbrande3@gmail.com, rst@uni-paderborn.de

Abstract
With the growing size of publicly available XML document collections, fast keyword search becomes increasingly important. We present an indexing and keyword search technique that is suitable for DAG-compressed data and has the advantage that common sub-trees have to be searched only once. We also present a performance evaluation that shows that our DAG-compressed index and search technique is superior to the corresponding tree-oriented keyword search technique that has been used up to now.

Keywords: Keyword Search, XML, XML compression, DAG

1 Introduction

1.1 Motivation
Nowadays, increasing amounts of XML data, e.g. product catalogues and open linked data, are made publicly available also to the non-expert users. While query languages for XML data like e.g. XPath and XQuery are powerful search tools for expert users, the non-expert users who just want to retrieve information related to some given keywords do not have the technical knowledge to write XPath or XQuery search queries. Therefore, for these users which are the great majority of users, there exists a great demand for efficient keyword search for XML data, where a user can write his query as a list of keywords expressing his search query – similar as the user is used to do this, when he uses a search engine for searching information within the internet.

1.2 Contributions
Our paper presents DAG-Index, an approach to efficient keyword search within XML data that is based on a compressed keyword index. To the best of our knowledge, DAG-Index is the first approach that combines the following features and advantages:

– Prior to building of the index, DAG-Index transforms the document into a DAG (directed acyclic graph) which removes redundant sub-trees from the document, such that DAG-Index yields a search index that is smaller, and thus can be searched faster than an index for XML trees.
– DAG-Index indexes common sub-trees only once, such that repetitive sub-trees have to be searched only once, if all the searched keywords are found in the sub-tree.
– DAG-Index uses proxy nodes for grouping equal keywords within each repetitive sub-tree into a single keyword occurrence within a proxy node, which additionally speeds-up keyword search if not all searched keywords occur in a shared sub-tree.

1.3 Paper Organization
The paper is organized as follows: Section 2 summarizes the basic idea of keyword search in XML data, and introduces the example used in this paper for visualizing our ideas. Section 3 describes the fundamental concepts used by our approach. The fourth section outlines some of the experiments that compare our prototype with keyword search on a non-compressed index. Section 5 gives an overview of related work and is followed by the Summary and Conclusions.

2 Our Goal and the Paper’s Example

2.1 Goal of XML Keyword Search versus Text Keyword Search
Plain text keyword search is known to many users from using an internet search engine. The user provides a list of keywords, and the search engine returns a list of documents containing these keywords. The search results are ranked according to the calculated ‘quality’ which is based e.g. on the importance of keywords for the document or on the distance of the keywords within the document.

Similar to the idea of plain text keyword search is the idea of keyword search for semi-structured data as it is provided in form of XML documents. The user provides a list of keywords, and the search engine returns these sub-trees of the documents that contain all keywords. Similar as for the ranking of the plain text documents, in order to get high quality information, the user not only wants any sub-tree containing all keywords, but the user might want to retrieve all the ‘smallest sub-trees’ containing all keywords, i.e. those sub-trees that contain all keywords but do not contain a smaller sub-tree that also contains all keywords.
2.2 This Paper’s Example

The example used in this paper is an excerpt of an XML document representing soccer players of the German soccer Bundesliga together with the teams they played for. Our example contains information on the player named “Manuel Neuer” who played for the teams “FC Bayern München” and “FC Schalke 04” and the player named “Timo Hildebrand” who played for the teams “FC Schalke 04” and “Sporting Lissabon”.

Figure 1 shows the binary XML tree of this document where instead of edges from parent to children there exist edges from a node to its first-child and its next-sibling. The numbers in parentheses represent the preorder number of each node.

Whereas typically relational data does not contain redundancies, this is nearly unavoidable for XML data. As there is a many-to-many relationship from players to teams, the XML document used as example contains a redundancy: the node “team” with first-child “FC Schalke 04” exists twice. This redundancy is removed by DAG compression. The second occurrence of a node is replaced by a backpointer to the first occurrence. The DAG of the example is shown in Figure 2. The second occurrence of each of both nodes “team” and “FC Schalke 04” is removed, and the next-sibling pointer from the “team” node of player “Timo Hildebrand” is directed to the first occurrence of the nodes “team” and “FC Schalke 04”.

2.3 Standard XML Keyword Search on the Example Document

A user might ask for the keyword list (“FC Bayern München”, “Timo Hildebrand”) in order to check whether or not Hildebrand played for Bayern München. Although both keywords are contained within the document, both keywords are not very closely related. Similar, if a user asks for the keyword list (“FC Bayern München”, “player”) in order to find out, which players did play for Bayern München, the results might be the sub-tree containing the information on “Manuel Neuer” and the sub-tree containing the whole document (matches are e.g. the player-node of “Timo Hildebrand” and the “FC Bayern München”-node of the other player-sub-tree). Intuitively, only the first solution (“Manuel Neuer”) is a desired solution.

Therefore, besides the requirement that all keywords have to be contained within the sub-trees, an additional requirement is added to increase the quality of the search result. A solution must not contain another solution. This property is called the “shortest lowest common ancestor (SLCA)”. As the second solution with root node “bundesliga” contains the first one with the left player-node as root node, the second solution is not considered as a solution, as it does not fulfil the SLCA property.

3 XML Keyword Search With Uncompressed Index

3.1 Preliminaries

Our paper follows the idea of anchor-based keyword search as presented in (Sun et al., 2007). Before we explain our changes on the search index and on the search approach, we explain the ideas of anchor-based keyword search. The approach is based on inverted lists that store for each keyword occurring in the XML document, a list of references to the document nodes with the keyword as node label.
Let $V$ denote the set of nodes in an XML document tree $T$, and let $v.nl$ denote $v$'s node label for each node $v \in V$. We provide an index based on inverted lists that maps each node label $nl$ occurring in $T$ to an ordered list $L_{nl}$ of those nodes $v \in V$ in document order for which $v.nl=nl$. Given two nodes $v_1, v_2 \in V$, $v_1 \prec v_2$ denotes that $v_1$ is a preceding node of $v_2$ in document order in the document $T$; and $v_1 \preceq v_2$ denotes that $v_1 \prec v_2$ or $v_1=v_2$.

Let further $K = \{w_1, \ldots, w_n\}$ denote a set of $k$ keywords given as input to the keyword search problem, i.e., the keyword search looks for all the smallest sub-trees of $T$ containing at least one node $v_i$ with label $v_i.nl = w_i$ for each $w_i \in K$.

A set $S=\{v_1, \ldots, v_k\} \subseteq V$ of nodes is defined to be a 

**match for $K$** if for each keyword $w_i$ in $K$, $S$ contains exactly one node labeled with that keyword, i.e., if $|S|=|K|$ and for each $w_i \in K$, there is $v \in S$ such that $v.nl = w_i$.

We use $v_1 \preceq v_2$ to denote that $v_1$ is a proper ancestor of $v_2$ in $T$, and $v_1 \preceq v_2$ to denote that $v_1$ is an ancestor-or-self of $v_2$, i.e., $v_1 = v_2$ or $v_1 \preceq v_2$. A node $v_1$ is a common ancestor of $S \subseteq V$, if for all $v_i \in S$, $v_1$ is ancestor-or-self of $v_i \in S$. A node $v_1$ is a lowest common ancestor of $S$, lca$(S)$, if $v_1$ is a common ancestor of $S$ and there is no common ancestor $v_2 \in V$ of $S$ with $v_1 \prec v_2$. The function $lca(S, T) = \{v \in T \mid \forall v_i \in S \ (v_1 \preceq v_i) \text{ and not } \exists v_2 \in T \ (v_1 \prec v_2 \text{ and } v \in S : (v_2 \prec v_i))\}$ computes the LCA in $T$ of the set of nodes $S$ and returns null if $S$ is null. If $T$ is obvious, we write $lca(S)$, instead of $lca(S, T)$.

Furthermore, a node $v_1 \in V$ is a lowest common ancestor for $K$ (LCA for $K$) if $v_1$ is the lowest common ancestor node of at least one match $S$ for $K$. Moreover, $v_1$ is also a smallest lowest common ancestor (or SLCA) for $K$ if no descendant of $v_1$ in $T$ is an LCA for any match for $K$.

For example, for the node set $S=\{4, 7\}$ in the document $T$ shown in Figure 1, $lca(S)$ is the node 2.

Consider a node $v$ and a set of nodes $S$. The function $first(S)$ returns that node $v'$ in $S$ with $v'$ \preceq $v$ for each $v \in S$. Similarly, the function $last(S)$ returns that $v'$ in $S$ with $v \preceq v'$ for each $v \in S$. Both functions return null if $S$ is null.

The function $next(v, S)$ returns the first node in $S$ that follows $v$ if it exists; otherwise, it returns null. The function $pred(v, S)$ returns the predecessor of $v$ in $S$, i.e., the last node in $S$ that precedes $v$ if it exists; otherwise, it returns null.

The function $closest(v, S)$ computes the closest node in $S$ to $v$ as follows:

$closest(v, S) = \{v, \text{ if } v \in S; \text{ otherwise } pred(v, S), \text{ if } lca\{v, \text{ next}(v, S)\} \prec lca\{v, \text{ pred}(v, S)\}; \text{ next}(v, S), \text{ otherwise}\}$. However, $closest(v, S)$ returns null if $v \in S$ and both $pred(v, S)$ and $next(v, S)$ are null; and it returns the non-null value if $v \in S$ and exactly one of $pred(v, S)$ and $next(v, S)$ is null.

A match $S = \{v_1, \ldots, v_k\}$ is said to be anchored by a node $v_a \in S$ if for each $v_i \in S \setminus \{v_a\}$, $v_i = closest(v_a, L_i)$. $v_a$ is then called the anchor node of $S$.

### 3.2 XML Keyword Search

The anchor-based keyword search on an XML document is then performed as follows (for details see (Sun et al., 2007)):

**Step 1: Chose an Anchor.** Initially chose that node $n$ as an anchor that occurs last in document order from all the first nodes of the inverted element lists $L_i$ of the keyword $w_i$, i.e. $n = last(\{first(L_i) \mid w_i \text{ is a searched keyword}\})$.

If we consider a keyword search for $w_1$="FC Bayern München" and $w_2$="FC Schalke 04" in order to find out, when they did play against each other, we get the following two inverted element lists, where each list entry is the preorder position of the node contained in the list: $L_1 = (7, 17, 33)$ and $L_2 = (10, 26)$. Therefore, we chose the node $n=10$ with label "FC Schalke 04" as initial anchor.

**Step 2: Compute the SLCA Candidates.** Let $L_a$ be the inverted element list containing $n$. We compute an SLCA candidate for a list $M$ that contains all nodes $v_i = closest(n, L_a)$.

For this purpose, in each inverted element list $L_i \neq L_a$ of keyword $w_i$, we chose $v_i = pred(n, L_i)$ as current node. The node $n$ and all these nodes $v_i$ form the initial list $M$ containing the match being currently regarded.

Considering our example, $M = \{7, 10\}$.

Next, we repetitively check, whether $v_i=first(M)$, i.e. the first node of the list $M$, where $v_i = pred(n, L_i)$ could be replaced by a node $v_i' = next(n, L_i)$ being closer to $n$. As long as such a node $v_i'$ is found for the currently first node $v_i$ of $M$, $v_i$ is substituted with the closer node $v_i'$, until no replacement is possible anymore. Then, the next SLCA candidate is $A = lca(\{v_i, last(M)\})$.

In our example, we check whether $v_i=7$ could be replaced by the node $v_i'=17$.

The replacement check is to check whether a node $v_i' = next(n, L_i)$ exists, and if so, to compute $A = lca(\{v_i', last(M)\})$ and to check whether $A < v_i'$.

In our example, $lca(\{7, 10\})=2$, $v_i'=17$, and $17 < 2$, i.e. 2 is replaced by 17.

Furthermore, whenever replacing a node $v_i$ by $v_i'$, we have to chose $v_i'$ as the next anchor if the following holds: For each keyword $w_i$ (i\neq j), there exists a node that occurs after the old anchor $n$ and before $v_i'$ in document order.

In our example, we have to replace the old anchor by the new anchor $n'=v_i'=17$.

Whenever an SLCA candidate $A$ has been computed, add $A$ to the set $C$ of result candidates. Furthermore, let $N=\{n_i \mid n_j=next(A, L_i)\}$ be the set of all the nodes that occur after $A$ in document order in one of the lists $L_i$ of nodes that have the keyword $w_i$ as their label, and repeat Step 2 taking $n=last(N)$ as new anchor until the end of the XML tree has been reached for at least one i, i.e., $next(A, L_i)$ does not exist.

In our example, we compute the new SLCA candidate for nodes 10 and 17, which is the node 8.

As node 33 is not a descendant of node 8, we add node 8 to the set of result candidates and continue the computation with node 33 as the new anchor resulting in node 24 as result candidate.
Step 3: Compute the Result Set from the Set C of Candidates. Remove all nodes $c_a \in C$ from the set of candidates $C$ for which a node $c_d \in C$ exists with $c_a < c_d$. All remaining nodes form the result set $R$.

As nodes 8 and 24 of our example are not in ancestor-descendant-relationship, we do not have to remove any node and return the set of candidates as final result.

4 XML Keyword Search Based on a Compressed Index

Instead of computing the inverted elements lists $L_k$ for each keyword $k_i$ based on the XML document (i.e., one list entry into the lists $L_k$ for each XML document node that has a label $k_i$), we compute the keyword list based on the compressed DAG of the XML document. Besides keeping the index small, the goal of using DAG compression is to search shared sub-trees only once, and thereby achieve a faster search speed.

4.1 Compressed Index

Prior to computing the index, we transform the XML document into its minimal DAG by replacing each repeated occurrence of a sub-tree with a pointer to the sub-tree’s first occurrence.

Similarly as for the uncompressed index, our compressed index consists of inverted element lists $L_k$ for each potential keyword $k_i$ that occurs as an element label or as a text node within the DAG. We do a bottom-up search for DAG nodes with multiple incoming edges and split the DAG into multiple sub-DAGs as follows: Whenever a node $v$ of the DAG D has more than one incoming edge, i.e., $v$ has the incoming edges $e_{v_1}, \ldots, e_{v_n}$, we remove $v$ from D and start a new sub-DAG Dv, where Dv is a copy of D with all nodes not being a descendant-of-self of $v$ in D being removed from Dv and with all dangling edges being removed, such that $v$ is the root node of Dv. Let $L_v$ be the set of labels occurring in Dv. Each edge $e_{v_j}$ gets a new (virtual) target node $v_j$, called proxy node of $v$, and for each $k_i \in L_v$, $v_j$ is added to the inverted element list $L_k$ representing all the occurrences of keyword $k_i$ in D.

Additionally, the information that $v_j$ is a proxy node for $v$ is stored in a table of proxy references where each node $v_j$ has a reference to the root node $v$ of Dv.

Figure 3 shows the document of our example where the DAG is split into two DAGs connected by the proxy nodes p1 and p2 (represented by white rectangles) and their references to the common team node (1') which is the second DAG’s root node.

4.2 Keyword Search on the Compressed Index

Keyword search on the DAG-compressed index works similar to keyword search on the uncompressed index, with the following differences: Due to the introduction of proxy nodes that represent multiple keywords occurring in a sub-DAG, the same proxy node-ID may occur in multiple inverted element lists, and the same proxy node-ID may occur multiple times within the currently considered list $M$ of actual nodes. Whenever during the computation of $M$, all elements of $M$ contain the same proxy node $v_j$, where $v_j$ refers to the root node $v$ of a sub-DAG Dv, the complete match is contained in Dv or in a sub-DAG of Dv. In this case, first, we remove a possible SLCA candidate C in D, second, if $C \leq v_j$, we perform the keyword search in Dv, and third, we start a new keyword search within D with a new anchor among the nodes after $v_j$, i.e., we continue after we have increased the pointer positions in all inverted keyword lists of D to next($v_j$). In this case, we have the advantage of computing the SLCA s within Dv for all shared sub-trees represented by Dv only once. Whenever this optimization is possible, we yield a faster search compared to computing all these solutions individually.

In the example of Figure 3, the first anchor node is the proxy node p1 and $v_j$ is the node 7. Then, during the computation, we set $v_j=p1$ (similar as it was the case for the non-compressed index). As now all nodes in M represent the same proxy node p1, we recursively start a new search at the node (1’), i.e. at the root node of the second DAG. Within this DAG, we find that the nodes with preorder positions 3’ and 10’ within the sub-DAG are SLCA s. Later, the second anchor node found in the first DAG is the proxy node p2 and a corresponding node
vi is the same proxy node p2. As p2 also refers to node (1’) which now has already been investigated, no new search starting in (1’) is required. Thereby, we have computed the SLCAs for both shared sub-trees only once – whereas, when using the non-compressed XML tree index, we had to compute the SLCAs represented by 3’ and 10’ twice, i.e. for both sub-trees.

5 Evaluation of Prototype Implementation

5.1 Evaluation Environment

We have implemented a prototype of our approach using Java 1.6.0. We compared our prototype using the compressed index with a similar implementation of the anchor-based keyword search on XML trees as described in Chapter 3.

As test documents, we used DISCOGS (http://www.discogs.com/data/), a discography database containing information on the releases, including information on the artist, the style, the genre, the originating country, the release date and comments.

We split the database into several chunks yielding documents starting from 50,000 releases (D50), having a file size of 42MB up to 350,000 releases (D350), having a file size of 271MB.

5.2 Evaluation Results

Figure 4: Searching the 4 most frequent keywords using scaling document sizes

In a first series of measurements, we assumed a sort of worst-case scenario for both approaches. We computed the 10 most frequent keywords contained within each document, and we searched for the n=1,2,...,10 most frequent keywords. This means that during the evaluation, the longest possible keyword lists have to be combined, yielding a sort of worst-case scenario for both approaches.

Figure 4 shows the results for the 4 most frequent keywords for documents with increasing file size. As we can see, for some documents the anchor-based approach (XML) was faster, whereas for other documents the DAG-based approach was faster. But neither approach completely outperforms the other one.

Figure 5 shows the results for document D250, containing 250,000 releases, and for an increasing number of keywords. We can see that for smaller numbers of keywords (3-6) the anchor-based approach is predominant, whereas for larger number of keywords (>7), i.e., for more complex queries, the DAG-based approach outperforms the anchor-based approach. i.e., the increase of the search times for increasing file size is much smaller for the DAG-based approach than for the anchor-based approach.

These queries form a worst-case scenario, but we consider them to occur only less frequently than the queries of the following experiments, as these queries search for high-level tag names only and do not search for XML text nodes. In a second series of measurements, we used a query, asking for the most frequent text node and its parent label (i.e., for all releases of genre “Electronic”).

Figure 5: Keyword query accessing frequent text nodes using scaling numbers of keywords

In a first series of measurements, we assumed a sort of worst-case scenario for both approaches. We computed the 10 most frequent keywords contained within each document, and we searched for the n=1,2,...,10 most frequent keywords. This means that during the evaluation, the longest possible keyword lists have to be combined, yielding a sort of worst-case scenario for both approaches.

Figure 4 shows the results for the 4 most frequent keywords for documents with increasing file size. As we can see, for some documents the anchor-based approach (XML) was faster, whereas for other documents the DAG-based approach was faster. But neither approach completely outperforms the other one.

Figure 5 shows the results for document D250, containing 250,000 releases, and for an increasing number of keywords. We can see that for smaller numbers of keywords (3-6) the anchor-based approach is predominant, whereas for larger number of keywords (>7), i.e., for more complex queries, the DAG-based approach outperforms the anchor-based approach. i.e., the increase of the search times for increasing file size is much smaller for the DAG-based approach than for the anchor-based approach.

These queries form a worst-case scenario, but we consider them to occur only less frequently than the queries of the following experiments, as these queries search for high-level tag names only and do not search for XML text nodes. In a second series of measurements, we used a query, asking for the most frequent text node and its parent label (i.e., for all releases of genre “Electronic”).

Figure 6 shows the results for increasing file sizes. It can be seen that for this query, the DAG-based approach outperforms the anchor-based approach. Only for one document (D250), the DAG-based approach needs as long as the anchor-based approach, but on average the DAG-based approach takes only 88% of the computation time of the anchor-based approach (at least 75% within our measurements).

To summarize, in our measurements for worst-case scenarios, the DAG-based approach performs approximately as good as the anchor-based approach, whereas for keyword queries, which we expect to occur more frequently, the DAG-based approach is superior and outperforms the anchor-based approach.

6 Related Works

There exist several approaches that address the problem of keyword search in XML.
These approaches can be roughly divided into two categories: approaches that examine the semantics of the queries in order to achieve query results of higher relevance on the one hand and approaches that concentrate on a higher performance for the computation of the set of query results on the other hand.

Within the first category, (Guo et al., 2003) not only focus on an efficient algorithm for keyword search based on inverted element lists, but they aim to rank the search results in such a way, that the user gets the (probably) most interesting results prior to the other results. SUITS (Zhou et al., 2008) is a heuristic-based approach, and the approach presented in (Petkova et al., 2009) uses probabilistic scoring to rank the query results. In order to enhance the usability, (Li et al., 2010) propose an approach on how to group the query results by category.

Within the second category (efficient result computation) most approaches are based on finding a set of (SLCA) nodes for all matches of a given keyword list.

Early approaches were computing the LCA for a set of given keywords on the fly. (Schmidt et al., 2001) propose the meet-operator that computes the LCA for a pair of nodes that match two query strings without requiring additional knowledge on the document structure from the user.

In contrast, recent approaches try to enhance the query performance by using a pre-computed index.

(Florescu et al., 2000) propose an extension of the XML query language XML-QL by keyword search. In order to speed-up the keyword search, they compute the so-called “inverted file” for the XML document – a set of inverted element lists – and store the contents within a relational database.

(Li et al., 2004) present two approaches to compute the Meaningful Lowest Common Ancestor (MLCA), a concept similar to the SLCA considered in our approach. Their first approach allows computing the MLCA with the help of standard XQuery operations, whereas their second approach is a more efficient approach that is based on a stack-based algorithm for structural joins.

XKSearch (Xu & Papakonstantinou, 2005) is another stack-based approach to compute the LCA. For each keyword k, they store two lists: one inverted element list L

containing all nodes with label k, and an ancestor list A

containing all nodes that have a descendant with label k. They process the nodes bottom-up and store the nodes that have not yet been completely examined on the stack. Whenever a node is found, the descendants of which have all the search keywords as labels, it is returned as a result. In this case, all its ancestors are removed from the stack, as they cannot form a result anymore.

JDeweyJoin (Chen & Papakonstantinou, 2010) returns the top-k most relevant results. They compute the results bottom-up by computing a kind of join on the list of DeweyIDs of the nodes in the inverted element list. Whenever they find a prefix that is contained in all relevant element lists, the node with this prefix as ID is a result candidate. In addition they use a weight function to sort the list entries in such a way that they can stop the computation after k results, returning the top-k most relevant results.

(Zhou et al., 2012) present a more efficient, but more space-consuming approach. The elements of their inverted element lists do not only contain the nodes that have the keyword as label, but also contain all ancestors of these nodes. Therefore, they can compute the SLCA by intersecting the inverted element lists with the list of keywords and by finally removing each result candidate, the descendant of which is another result candidate.

Like the contributions of the second category, our paper focuses on efficient result computation. It follows the anchor-based approach as it was presented in (Sun et al., 2007). However, different from all other contributions, instead of computing an XML-index, we compute a DAG-Index. This helps to compute several keyword search results in parallel, and thereby speeds-up the SLCA computation. To the best of our knowledge, DAG-Index is the first approach that improves keyword search by using XML compression before computing the search index.

7 Summary and Conclusions
Keyword search is of increasing interest for searching relevant data within large XML document collections, especially for the huge majority of non-expert users. Due to the increasing amount of publicly available data in the XML format, there is an increasing interest in fast keyword search techniques. We have presented DAG-Index, an indexing and keyword search strategy for large XML documents that allows compressing an XML tree and the search index in such a way that common sub-trees have to be indexed only once. As a consequence, a repeated keyword search within a repeated sub-tree can be avoided. Therefore, we consider our DAG-Index-based keyword search to be a significant contribution to improve the search performance especially for the majority of the non-expert users.

8 References


