Ranking Friendly Result Composition for XML Keyword Search

Ziyang Liu[1], Yichuang Cai[2], Yi Shan[3](✉), and Yi Chen[4]

[1] LinkedIn, Mountain View, CA, USA
ziliu@linkedin.com
[2] Microsoft, Redmond, WA, USA
yica@microsoft.com
[3] School of Computing, Informatics, and Decision Systems Engineering,
Arizona State University, Tempe, AZ, USA
yshan1@asu.edu
[4] School of Management, New Jersey Institute of Technology, Newark, NJ, USA
yi.chen@njit.edu

Abstract. This paper addresses an open problem of keyword search in XML trees: given relevant matches to keywords, how to compose query results properly so that they can be effectively ranked and easily understood by users. The approaches adopted in the literature are oblivious to user search intention, making ranking schemes ineffective on such results. Intuitively, each query has a search target and each result should contain exactly one instance of the search target along with its evidence about its relevance to the query. In this paper, we design algorithms that compose atomic and intact query results driven by users' search targets. To infer search targets, we analyze return specifications in the query, the modifying relationship among keyword matches and the entities involved in the search. Experimental evaluationsvalidate the effectiveness and efficiency of our approach.

Keywords: Keyword search · XML Tree · Search intent

1 Introduction

Keyword search provides a simple and friendly mechanism to access the information in XML documents for users who do not know structured query languages, or for the applications in which data schemas are too complex or fast-changing.

To generate results for XML keyword search, we need to (1) Identify relevant matches to input keywords. (2) Compose query results based on the relevant matches. (3) Rank the results according to their relevance to input keywords.

Much research has been performed to address the challenges in the first and third steps. For identifying relevant matches, existing approaches use *V*ariants of *L*owest *C*ommon *A*ncestors to connect keyword matches, referred as VLCA in this paper [1–9]. For example, consider a query "*Arizona, state*" on the XML tree in Fig. 1. In Fig. 1, we assign integer IDs to some nodes to facilitate illustration.

P. Johannesson et al. (Eds.): ER 2015, LNCS 9381, pp. 441–449, 2015.
DOI: 10.1007/978-3-319-25264-3_33

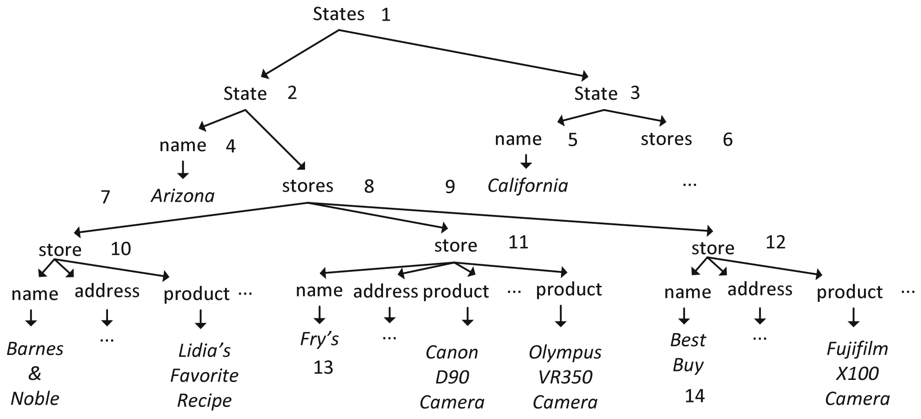


Fig. 1. Sample XML tree about stores

Although there are many matches to *state*, only the one (with ID 2) that corresponds to the *Arizona* match is considered relevant. For result ranking, a variety of ranking factors were proposed [2], including result size, keyword proximity, number of keyword matches in the result, as well as IR-style ranking functions.

However, little study has been done on the second problem: composing query results. In text search, a retrieval unit is a *document* in the repository, and thus the problem of result composition is inapplicable. In contrast, XML keyword search engines returns subtrees in the XML data tree in order to provide finer-grained query results and to better satisfy the users' needs. Given relevant keyword matches, how to dynamically extract a right subtree from the original data tree as a query result is not trivial. At the same time, the way of composing results in XML keyword search has crucial effects on result ranking and user search experience, as to be shown in the following examples.

Example 1. A user seeking the stores selling cameras in Arizona would issue a query "*Arizona, camera, store*". Consider the XML tree in Fig. 1 as a fragment of the XML data.

One popular query result composition methods is named as *Subtree Result* in this paper, adopted in many existing work [4,10,11]. Subtree Result defines a query result as a tree rooted at a VLCA node consisting of *all* relevant matches that are descendants of this VLCA node and the paths connecting them.

For this query, Subtree Result only returns one result: the tree rooted at a *state* node (0.0) that contains the match to *Arizona* and *all* the matches to *store* and *camera*, as shown in Fig. 2(a).

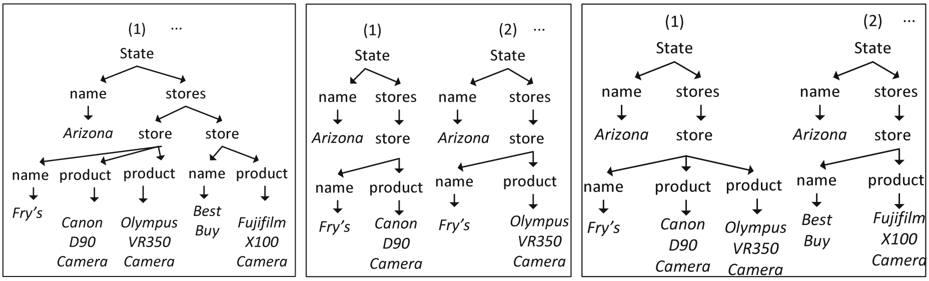
Such a result is not informative to the user. As we would imagine, there are hundreds of stores in Arizona that sells cameras. Instead of checking all of them, a user would want to find the top ranked stores to visit, where the ranking might consider store reputation (mimic to page-rank), the number of cameras sold (related to TFIDF ranking), location (related to local search), among other factors. However, with all the camera stores in Arizona in a single result, none of the existing ranking methods can rank the stores. With such a query result, a user would have to spend prohibitive amount of efforts to check the information of every store, design a ranking method herself to rank the stores in order to decide which few of them to visit.

Besides Subtree Return, the other commonly used result composition method is called *Pattern Match*, used in [2,3]. Pattern Match defines a query result as a tree rooted at a VLCA node consisting of *exactly one relevant match to each query keyword* and the paths connecting them.

For the query "Arizona, camera, store" on the XML tree in Fig. 1, some sample results generated by Pattern Match are shown in Fig. 2(b). As we can see, each result has information of exactly one camera.

Such results can be annoying. The data may have thousands of cameras, sold by hundreds of stores. The query user looks for *stores* that sells cameras, not cameras. It's much more desirable to have all distinct stores ranked automatically, rather than having the user to read tens of cameras sold by the same store and to manually rank the stores in order to find the top ranked ones.

In contrast to the two existing approaches, this paper presents techniques that compose query results based on the inferred search semantics. For the same query "Arizona, camera, store", we identify that the user is looking for camera stores in Arizona, and compose query results such that each result contains information of one distinct store, along with the related matches to *Arizona* and *camera* as evidence of its relevance, such as the two query results shown in Fig. 2(c). In this way, when results are properly ranked, a user obtains the top ranked stores.



(a) Subtree Results (b) Pattern Match Results (c) Desirable Results

Fig. 2. Results of query "Arizona, camera, store"

Intuitively, each keyword search has a goal, either a real world entity or relationship among entities, as observed in [12]. We use term *search target* to refer to the information that the user is looking for in a query, and *target instance* to denote each instance of the search target in the data. Each desirable query result should have *exactly one target instance* along with all associated evidence that shows the relevance to the user query, so that top-k ranked results corresponds to top-k ranked search target instances.

Based on this intuition, we propose a novel technique to automatically compose atomic and intact query results for XML keyword searches. Unlike the existing approaches, which are oblivious to users' search intentions, the proposed query result composition is *driven by the user search target* and hence is ranking friendly. Experimental evaluations validate the effectiveness and efficiency of our approach.

2 Target Driven Query Result Composition

Users who issue queries often desire the information of one or a set of entities that satisfy certain conditions. We name such entities as *target entities*. In this section we first introduce the data model and then discuss how to automatically infer target entities from user query and data. Then we discuss how to compose meaningful query results based on relevant matches and inferred target entities. We propose strategies to compose query result centred around the inferred search target.

2.1 Identifying Target Entities

We first introduce the data model of an XML tree. We consider a node in the XML data tree has one of three categories: *entities* that represent real world objects, *attributes* that describe the corresponding entities, and *connection nodes* which connect other nodes but do not have much meaning. Our system adopts the heuristics developed in [10] to infer node categories. For example, in Fig. 1, *state* is entity and *name* is its attribute. *stores* is a connection node. A *Keyword Query* is a set of words. A *Query Result* is a tree that consists of a set of relevant matches as well as the edges connecting them. Relevant keyword matches and VLCA nodes can be identified by applying one of the existing works introduced in Sect. 1. An entity instance is a *relevant entity instance* if it is on the path from a VLCA node to a relevant match in XML tree. The types of such data nodes are *relevant entities*. Consider query "Arizona, camera, store" on the data in Fig. 1. The relevant entities are *state*, *store* and *product*.

We infer target entities by analyzing the matches to input keywords and the XML data structure.Often a query has two parts: the information that a user is looking for, referred as *return node*, and the constraints that should be satisfied, referred as *search predicates*. These are analogous to the *select* clause versus *where* clause in SQL queries. For example, a user may look for stores in Arizona and issue a query "Arizona, store". In this query, store is a return node, and Arizona is a search predicate.

To automatically detect return nodes and search predicates from a user query, we observe that if an entity or attribute node name is specified in a query without information about its associated attribute values, then likely this node name represents a return node, and its attribute values are what the user is looking for. On the other hand, an attribute value node (e.g., *Arizona*), or a pair of attribute name and value (e.g., *state, Arizona*) is likely to be a search predicate.

In a query Q, we consider a keyword k to be a *return node* if one of the following conditions holds, otherwise is a search predicate. (1) k matches an entity e, and there is no keyword k' that matches its attribute or attribute value. (2) k matches a connection or attribute node u, and there is no keyword k' matching a node v, such that u is an ancestor of v.

Next we can infer target entities from return nodes. If a return node is an entity node, then it is a target entity. If a return node is an attribute (e.g., *address*), then the associated entity (e.g., *store*) is considered as a target entity. Otherwise, a return node is a connection node (e.g., *stores* in sample XML document), then its nearest descendant entities are considered as target entities (e.g., *store*).

However, not all user queries provide hints for return nodes. In case the query keywords do not contain return nodes, we exam all the relevant entities and the relationships between search predicates and theses entities to identify target entities. We have two observations. First, a user may use the attribute values to modify an entity (e.g. find stores that are named Fry's), or attribute values of a related entity to modify an entity (e.g. find stores that are in the state of Arizona). Second, each search predicate keyword shall modify the instances of the target entity. In other words, removing any keyword from the query will return different target instances. For example, users will not use query "Arizona, Store" to search for states that are named as "Arizona" and that have a store, because every state has stores, and keyword "store" does not modify instances of state. We propose *modifier* to represent a keyword match of search predicate and *modifying relationship* to represent the relationship between the match and an entity. Based on observation 1, we define modifier to be any attribute value or name-value pair A connected to entity E. Furthermore, based on observation 2 A is a modifier if there is at least one instance of E in the XML tree which does not have A connected. Otherwise A cannot be used to further describe E. So E is a target entity when all search predicates can modify it. Due to limited space, we refer readers to [13] for details.

When there is only a single target entity, it is called the *center entity* of the query result which is the search target of the keyword query. When there are no target entities found, we consider all relevant entities as target entities. When there are multiple target entities found, besides these entities we further consider the relationships between them as search targets.

2.2 Composing a Query Result

As illustrated by examples in Sect. 1, results of XML keyword search should be *atomic*, i.e., consist of a single target instance; and *intact*, i.e., contain the whole target instance together with all its supporting information. For example, in Example 1, each query result should correspond to a distinct store. Atomicity enables the ranking method to rank the target instances and show the top-k most relevant ones to the user. Also, each query result should have all supporting information, all cameras sold by the store. With intactness, a ranking method can have the whole view of each target instance to give a fair ranking. A keyword match of k is supporting information of a target instance e if there are no other keyword matches of k with closer relationship to e [13].

When a query has multiple target entities, we observe that atomicity and intactness may not be simultaneously achievable. Consider Fig. 1 for example. If both *state* and *store* are known as target entities, then all *store* nodes in the subtree of a *state* node constitute the supporting information of the *state* node. According to intactness, they should all be included in the result that contains this *state* node. However, according to atomicity, only one *store* node can be included in one result. In this case, since atomicity and intactness are not both achievable, we choose to achieve intactness using subtree result. The reason is that subtree result can be achieved much more efficiently and scalably than pattern match.

3 Algorithms

In this section, we present the indexes and algorithms to efficiently identify target entities and generate meaningful query results.

3.1 Indexes

A core operation in the query composition is to identify target entities. There are three indexes we use: *Label Index*, *Node Index*, and *Modifier Index*. Label Index is an inverted index, which retrieves a list of data nodes given a keyword.

As discussed in Sect. 2, we have different strategies for determining target entities in two different situations. If there are return nodes specified in input keywords, target entities are the entities associated with the return nodes. Otherwise, if return nodes are not specified, we check the modifying relationship between each attribute value that matches a keyword and each relevant entity involved in the query.

For the first case, we need to quickly determine a return node's associated entity. To support this, we build a node index for all entity and attribute nodes. For an entity node, the entry includes all its attribute values. For an attribute node, the entry includes its value and its parent entity.

For the second case, we need to quickly determine whether a predicate is a modifier of an entity type. To support it, we build a Modifier index that records the modifying relationship between attribute values (more accurately, attribute name value pairs) and entities. In this index, each attribute value has an associated list that records the entities that are *not* modified by this attribute value. Since the entities that are modified by an attribute value are far more than those that are not modified by it, we record the negative cases.

3.2 Generating Query Results

Now we present our algorithms, which takes relevant matches to a keyword query (which are obtained by adopting one of the existing approaches [2–4,14]) and indexes as input, and composes meaningful query results.

Based on Sect. 2, first we need to identify target entities based on return nodes and search predicates. Second, we use target entities to construct query results which are atomic and intact.

For a keyword search, first we use the label index to retrieve all nodes matched by a query keyword. Second, we compute VLCA nodes, each of which is the root of one or more query result trees, for which purpose we use the algorithm proposed in XKSearch [4]. Third, we find relevant entity instances by checking whether they are on the paths from a VLCA to a relevant match (Sect. 2.1). Fourth, using node index, we can determine the node category of each keyword match and then infer the return nodes and search predicates of the query. To achieve that, we first retrieve the entry for each keyword. If it is matched by an entity or attribute name, we further exam whether other keywords match a value of an attribute of the entity or a value of the attribute. If a keyword is matched by an entity or attribute name but no other keywords match its value, then this keyword is a return node, otherwise search predicate. If return nodes are present, we can find the corresponding target entity instances by accessing the node index as well. Otherwise, we find the modifying relationship and consequently the target entity instances using the modifier index by checking whether each search predicate modifies the target entity instance or not. Next, if a center entity exists, then each relevant entity instance that is an instance of the center entity leads to a query result. For each such entity instance e, a query result is generated consisting of e and its supporting information (Sect. 2.2), together with their connections. Such a result is atomic and intact. If there is no center entity, it takes the default mode, which requires the result to be *intact*. To achieve that, it generates query results by returning the relevant matches in the subtrees rooted at VLCA nodes.

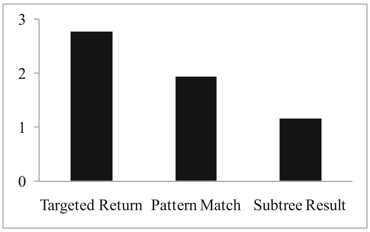


Fig. 3. User scores of query results

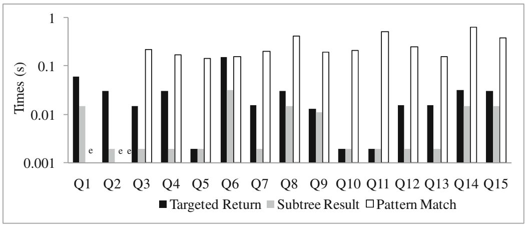


Fig. 4. Processing time on Baseball data

4 Experiments

In this section we present experimental study of our approach for composing query results, *Targeted Return*.

We have tested one data set: Baseball [15] with 15 real user queries.[1] We compare Targeted Return with Subtree Result and Pattern Match. For each

[1] https://db.tt/omJcnxdX.

query, the users were given the results generated by the three approaches with shuffled order, and were asked to give an overall satisfaction score based on their impression of each query result of scale [1, 3]: 3 means I have no trouble finding what I am looking for; 2 means I need efforts to extract the desired information from the results; 1 means I cannot find what I am looking for without re-organizing the results. The average scores of the three approaches of all 15 test queries given by the user is shown in Fig. 3. Targeted Return got the best score of 2.76, followed by Pattern Match 1.92 and Subtree Result 1.15. This indicates that the organization of query results by Targeted Return is closest to users' expectations. Response time of all three systems over the 15 queries are shown in Fig. 4. *e* indicates that the system Pattern Match fails to return any query results because of its data size limit. As we can see, Targeted Return achieves comparable processing time with other systems.

5 Conclusions

Our approach of composing query results is driven by search targets, and produces atomic and intact results. To identify user search targets, we infer return nodes for keyword searches and the modifying relationship among attribute values and entities in the data. Then we determine the target entities and center entity for a keyword search, based on which query results are composed. Experimental evaluation has shown the effectiveness and efficiency of our approach.

Acknowledgements. This work is partially supported by NSF CAREER Award IIS-0845647, Google Cloud Service Award and the Leir Charitable Foundations.

References

1. Liu, Z., Cai, Y., Chen, Y.: TargetSearch: a ranking friendly XML keyword search engine. In: ICDE (2010)
2. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: a semantic search engine for XML. In: VLDB (2003)
3. Li, Y., Yu, C., Jagadish, H.V.: Schema-free XQuery. In: VLDB (2004)
4. Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD (2005)
5. Zhou, J., Bao, Z., Wang, W., Zhao, J., Meng, X.: Efficient query processing for XML keyword queries based on the IDList index. VLDB J. **23**(1), 25–50 (2014)
6. Zeng, Y., Bao, Z., Ling, T.W., Li, G.: Removing the mismatch headache in XML Keyword search. In: SIGIR (2013)
7. Le, T.N., Bao, Z., Ling, T.W.: Schema-independence in XML keyword search. In: Yu, E., Dobbie, G., Jarke, M., Purao, S. (eds.) ER 2014. LNCS, vol. 8824, pp. 71–85. Springer, Heidelberg (2014)
8. Zeng, Y., Bao, Z., Ling, T.W., Li, G.: Efficient XML keyword search: from graph model to tree model. In: Decker, H., Lhotská, L., Link, S., Basl, J., Tjoa, A.M. (eds.) DEXA 2013, Part I. LNCS, vol. 8055, pp. 25–39. Springer, Heidelberg (2013)

9. Shi, J., Lu, H., Lu, J., Liao, C.: A skylining approach to optimize influence and cost in location selection. In: Bhowmick, S.S., Dyreson, C.E., Jensen, C.S., Lee, M.L., Muliantara, A., Thalheim, B. (eds.) DASFAA 2014, Part II. LNCS, vol. 8422, pp. 61–76. Springer, Heidelberg (2014)
10. Liu, Z., Chen, Y.: Identifying meaningful return information for XML keyword search. In: SIGMOD (2007)
11. Lin, R.-R., Chang, Y.-H., Chao, K.-M.: Improving the performance of identifying contributors for XML keyword search. SIGMOD Rec. **40**(1), 5–10 (2011)
12. Cheng, T., Chang, K.C.-C.: Entity search engine: towards agile best-effort information integration over the Web. In: CIDR (2007)
13. Liu, Z., Cai, Y., Chen, Y.: Ranking friendly result composition for XML keyword search, ASUCIDSE-2015-001. Technical report, Arizona State University (2015)
14. Li, G., Feng, J., Wang, J., Zhou, L.: Effective keyword search for valuable LCAs over XML documents. In: CIKM (2007)
15. Baseball Dataset. http://www.ibiblio.org/xml/books/biblegold/examples/baseball