eXtract: A Snippet Generation System for XML Search

Yu Huang Ziyang Liu Yi Chen
Arizona State University
{yu.huang.1, ziyang.liu, yi}@asu.edu

ABSTRACT

Snippets are used by almost every text search engine to complement ranking schemes in order to effectively handle user keyword search. Despite the fact that XML is a standard representation format of web data, research on generating result snippets for XML search remains untouched. In this work, we present eXtract, a system that efficiently generates self-contained result snippets within a given size bound which effectively summarize the query results and differentiate them from one another, according to which users can quickly assess the relevance of the query results.

1. INTRODUCTION

Keyword search is widely used for information retrieval on both unstructured and structured data. To improve user search experience, various ranking schemes have been proposed so that users can focus on the ones that are deemed highly relevant. However, due to the intrinsic ambiguity of keyword search, no ranking schemes can always perfectly assess the relevance of query results, and typically a user needs to navigate through several results to find the desirable ones. To compensate the inaccuracy of ranking functions and reduce users' navigation burden, result snippets are used by almost every text search engine. A snippet provides a brief quotable passage of the query result in order to help users quickly judge the relevance of the query result and choose the relevant ones among many results.

Despite the fact that XML is a standard representation format of web data, the problem of generating result snippets for XML keyword search remains untouched. Compared with text documents, XML data are semi-structured with mark-ups providing meaningful annotations to data content, therefore present better opportunities for generating helpful result snippets. In this paper we address this important yet open problem.

As an example, consider a query "*Texas, apparel, retailer*" whose result fragments are shown in Figure 1.[1] Some statistics of the whole query result are shown at the right portion in the figure, where for each distinct value, we show the number of its occurrences in the query result. For instance, "*city: Houston: 6*" indicates that there are 6 stores in the query result that are in the city of Houston. Values with low occurrences are omitted.

[1]Snippet generation takes query results as input, thus we omit the description of the procedure of query result generation.

A good snippet for this query result would look like the one shown in Figure 2. It captures the heart of the query result in a small tree: the query result is about *retailer* named as: *Brook Brothers*. This retailer features clothes of *outwear* in *casual* situation, for both *man* and *woman*. Furthermore, this retailer has many stores in *Houston*.

We identify that a good XML result snippet should be a *self-contained* information unit within *a bounded size* that effectively *summarizes* the query result and *differentiates* itself from others. To achieve this goal, several technical challenges need to be addressed.

To generate self-contained snippets, we need to identify the semantic information units in the query results. In our example, the query result is about an entity *retailer*.

To help users distinguish different query results from one another with little effort, in analogy to text document search where the document titles are included in the snippets, we propose to include the key of a query result in its snippet. In our example, ideally the *name* of the retailer should be included. However, a query result often contains many entities, such as *store*, *clothes*, each of which can have keys. It is not clear which entities' keys can serve as the key of the query result.

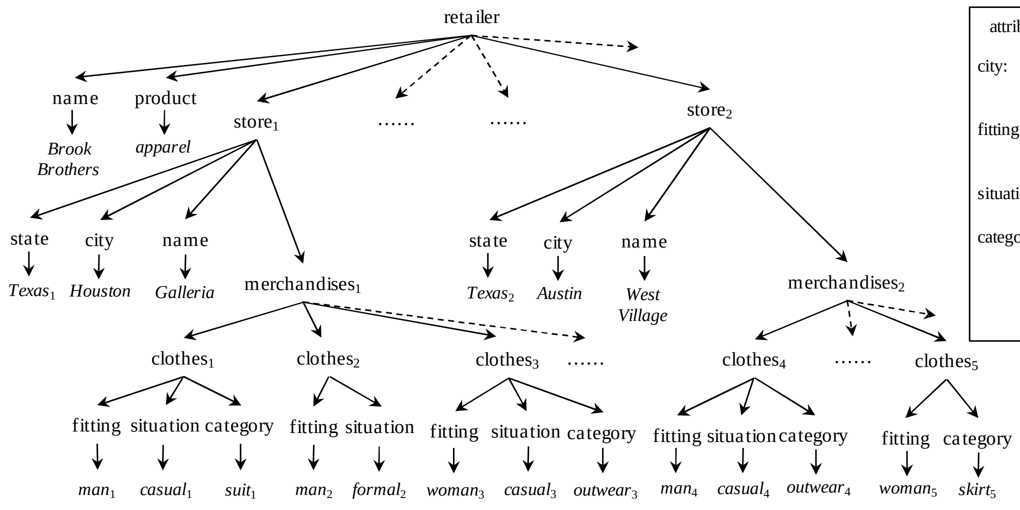
Besides, a snippet should provide a representative summary of the query result, by capturing the most prominent features of the query result. Intuitively, a prominent feature should have a dominant number of occurrences in the query result. However, this relationship is not always reliable. In our example, although the number of occurrences of *Houston*: 6, is much less than that of *children*: 40, considering that the majority of *Brook Brothers* stores are in *Houston* in the query result, it should be considered as a prominent feature.

Furthermore, a snippet should be small so that the user can quickly browse it and decide whether this query result is relevant for further examination or not.

To address these challenges, we present eXtract, a system that generates effective snippets for XML keyword search results to help users quickly identify the most relevant results.

The technical contributions of our work include:

- To the best of our knowledge, eXtract is the first system that generates query result snippets for XML search.
- We identify four goals that a good query result snippet should meet in order to help users quickly get the essence of a query result and assess its relevance.
- To address the goals, we identify the most significant infor-



attribute: value: number of occurrences

city:	Houston:6
	Austin: 1
	Other cities (3): 3
fitting:	Man: 600
	Woman: 360
	Children: 40
situation:	Casual: 700
	Formal: 300
category:	Outwear: 220
	Suit: 120
	Skirt: 80
	Sweaters: 70
	Other categories (7): 580

Figure 1: Part of a query result of query "Texas apparel retailer" and statistics about value occurrences.

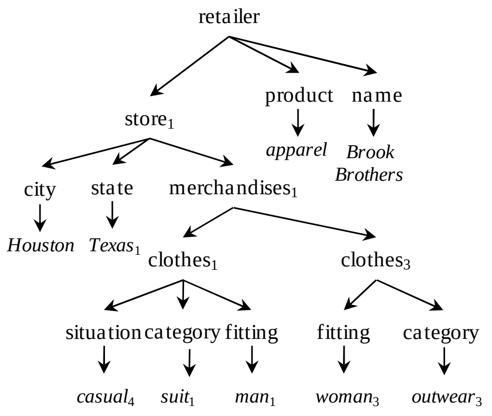


Figure 2: Snippet of the query result in Figure 1

mation in a query result to be selected into a snippet.

- We prove that the problem of constructing a snippet of a given size limit that maximally contains the significant information is NP-hard.
- We design an efficient greedy algorithm to generate snippets that are self-contained, distinguishable, representative and has a size within a given limit.

2. SNIPPET GENERATION

To generate snippets, the eXtract system takes a keyword query, an XML database, the query results, and a snippet size limit as the input. It first identifies the most important information from each query result according to the proposed goals (to be explained in Sec 2.1, 2.2 and 2.3). Such information is placed in the *Snippet Information List*, denoted as *IList*, in the order of their importances. Then eXtract selects node instances in the query result that contains the information in IList in their ranked order to build a snippet tree. Since multiple node instances in the query result can cover the same item in IList, choosing different instances generally results in different sizes of the snippet. Given a snippet size bound, eXtract carefully selects the instances of each item in IList from the query result in order to cover as many items in IList as possible till the size bound is met.

IList is initialized with the query keywords. For the query in Figure 1, *IList* is initialized as "*Texas, apparel, retailer*". Now we discuss the goals of snippet generation and how to construct *IList* to meet the goals.

2.1 Self-contained Snippets

Recall that in textual documents, each sentence is a basic information unit. A result snippet usually contains one or more "windows" of sentences in the documents containing query keywords. Analogously, XML result snippets should also be based on the basic information unit so as to be self-contained. To identify basic semantic units, we analyze that an XML database contains information about real-world entities with associated attributes as well as their relationships. An entity represents a basic semantic information unit.

We adopt the approach in [6] that leverages DTD or XML data structure to classify XML nodes into three categories: entities, attributes, and connection nodes. Specifically, a node is considered as an *entity* if it corresponds to a *-node in the DTD. If a node is not a *-node and only has one child which is a text value, then this node, together with its value child, represents an *attribute*. A node is a *connection node* if it represents neither an entity nor an attribute.

To make the snippet self-contained, eXtract includes the names of entities involved in the query result into its snippet. In the sample query result in Figure 1, *retailer*, *store* and *clothes* are entities. Therefore we update *IList* by adding the entity names, and now have the *IList* as: "*Texas, apparel, retailer, clothes, store*".

2.2 Distinguishable Snippets

To make a snippet distinguishable from the snippets of other query results, we propose to include the key of a query result into the snippet, which resembles the title of a text document. However, it is not clear how to define the key of a query result. Recall that a query result often contains multiple entities, each of which can have a key attribute. It is critical to find the "most important" entities in the query result whose key attribute can serve as the key of the result.

Intuitively, each query has a search goal. The search goal can be used to classify the entities in a query result into two categories:

Texas, apparel, retailer, clothes, store, Brook Brothers, Houston, outwear, man, casual, suit, woman

Figure 3: *IList* of the query result in Figure 1

return entities, the entities that the user is looking for by issuing the query, and *supporting entities*, the entities that are used to describe return entities in the result. Take query "*Texas, apparel, retailer*" as example. The user is likely to search for the retailer of apparel in Texas state. Therefore *retailer* should be considered as the return entity of this query, while other entities in the query results, *store* and *clothes*, are likely to be supporting entities. The keys of return entities can be used as the key of query result.

To infer these two types of entities, we propose the following heuristics: an entity in a query result is a *return entity* if its name matches a keyword or its attribute name matches a keyword. If there is no such entity, we use the highest entity (i.e. entities that do not have ancestor entities) in the query result as the default return entity.

For the sample query "*Texas apparel retailer*", entity retailer has its name matching a keyword, and therefore is considered as a return entity, corresponding to the user search goal. After mining the keys of entities in the data, eXtract adds the value of the key attribute of retailer: *Brook Brothers*, which is considered as the key of this query result, to *IList*.

2.3 Representative Snippets

A desirable snippet should be representative, providing a good summary of the query result by including the most prominent features of the result.

We define a feature as a triplet (entity name e, attribute name a, attribute value v). The pair (e, a) is referred as the type of a feature, and attribute value v is referred as the value of a feature. (e, a, v) denotes that entity e has an attribute a with feature value v. For example (*store*, *city*, *Houston*) is a feature indicating that the store is in the city of Houston. For presentation purpose, we refer a feature by its value when there is no ambiguity.

A dominant feature of a query result is often reflected by a large number of occurrences of this feature in the result. For example, in Figure 1, there are 6 stores in *Houston*, and 4 stores in four other cities. *Houston* is thus considered as dominant.

However, the relationship between the dominance of a feature and the number of occurrences is not always reliable due to two reasons. First, different features have different domain sizes. The domain size of a feature type (e, a) is defined as the number of distinct values of this type, denoted as $D(e, a)$. The smaller size a domain has, the more chances there is for a value to have a large number of occurrences in the result. For example, the number of occurrences of *outwear* is less than that of *woman* in the query result. However, considering their corresponding feature types, (*clothes*, *category*) has a larger domain size than (*clothes*, *fitting*), thus *outwear* could be a more dominant feature than *woman*.

Second, different feature types have different total number of value occurrences, denoted as $N(e, a)$. The more occurrences of a feature type, the more chances for a value of this feature type to occur. For example, though the number of occurrences of feature *Houston* is much less than that of *children*, it should be considered as more dominant considering that the features of type (*store*, *city*) appear much less than those of type (*clothes*, *fitting*) in the result.

Since comparing the number of occurrences of different features may not make sense in determining dominant features, we propose to use normalized frequency, called *dominance score*, to measure the significance of a feature in a query result. The *dominance score*

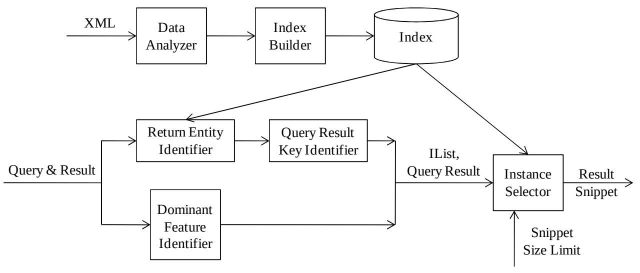


Figure 4: Architecture of eXtract

of a feature $f = (e, a, v)$ in query result R, denoted by $DS(f, R)$, is defined as follows:

$$DS(f, R) = \frac{N(e, a, v)}{\frac{N(e,a)}{D(e,a)}}$$

where $N(x)$ denotes the number of occurrences of x in R, $D(e, a)$ denotes the domain size of feature (e, a) in R. We may omit R and use $DS(f)$ when R is explicit.

A feature is dominant if its dominance score is larger than 1. Intuitively, a dominant feature should have the number of occurrences more than the average number of occurrences of the feature values of the same type. There is one exception: if the domain size is 1, $D(e, a)$ = 1, then there is only one value of this feature type, which is trivially considered to be dominant even though its dominance score is 1.

eXtract includes dominant features into *IList* in the decreasing order of their dominance scores. In Figure 1, DS(*Houston*) = 6/(10/5) = 3.0. Similarly, the dominance scores of *man, woman, casual, outwear* and *suit* are 1.8, 1.1, 1.4, 2.2 and 1.2 respectively. They are added to *IList*, which now becomes *Texas, apparel, retailer, clothes, store, Brook Brothers, Houston, outwear, man, casual, suit, woman*.

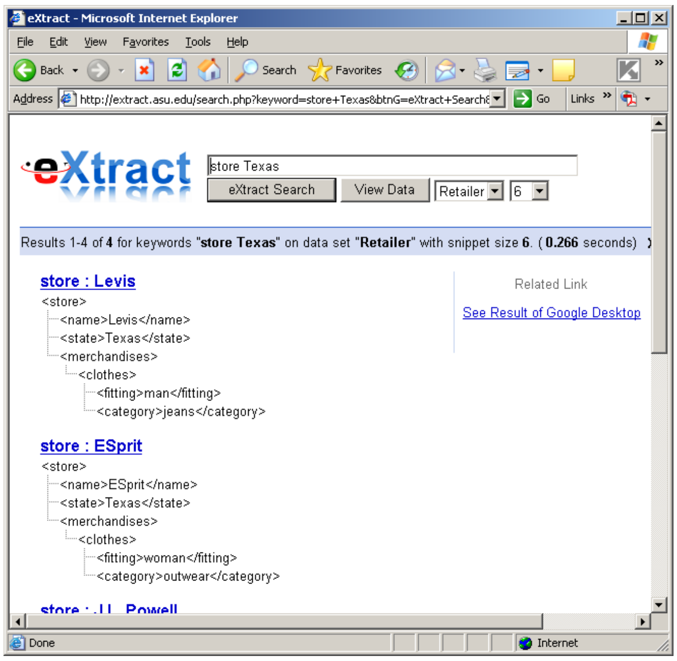
2.4 Small Snippets

Given a snippet size bound, eXtract aims at including as many items in *IList* as possible in the order of their significance, by carefully selecting the instances of each item from the query result. Intuitively, we should select instances of each item such that they are close to each other, so as to occupy a small space and leave room to include more items in *IList*. Considering feature *Houston* and *outwear* as an example. Choosing $outwear_3$ in Figure 1 results in a smaller tree with *Houston* than $outwear_4$.

The problem of maximizing the number of items in *IList* that are captured in a snippet within a bounded size is NP-hard. To provide a practical and efficient solution, we design a greedy algorithm for selecting item instances. The proof of NP-hardness and the detailed algorithm can be found in [3].

3. SYSTEM ARCHITECTURE

The architecture of eXtract is presented in Figure 3. The *Data Analyzer* parses the input XML data and identifies the entities, attributes and connection nodes. The *Index Builder* builds indexes for efficiently retrieving matches to user input keywords, as well



http://extract.asu.edu/search.php?keyword=store+Texas&btnG=eXtract+Search&

store Texas | Retailer | 6

Results 1-4 of **4** for keywords "**store Texas**" on data set "**Retailer**" with snippet size **6**. (**0.266** seconds)

store : Levis

```
<store>
   <name>Levis</name>
   <state>Texas</state>
   <merchandises>
      <clothes>
         <fitting>man</fitting>
         <category>jeans</category>
```

See Result of Google Desktop

store : ESprit

```
<store>
   <name>ESprit</name>
   <state>Texas</state>
   <merchandises>
      <clothes>
         <fitting>woman</fitting>
         <category>outwear</category>
```

store : JJ Powell

Figure 5: Search results of eXtract

as the information about node category, and parent-children relationship. The core components of the system are *Return Entity Identifier, Query Result Key Identifier, Dominant Feature Identifier* and *Instance Selector*. The eXtract system takes a user keyword query, its query results and the snippet size bound as input. Note that the query results can be generated by any XML keyword search engine. The *Return Entity Identifier* identifies the search target of the user among all the entities in the query result. The *Query Result Key Identifier* finds the key value of the return entity, which serves as the key of the query result to distinguish different query results. *Dominant Feature Identifier* traverses the query result and calculates the dominance score for each feature. Then dominant features are identified according to their dominance scores. The query keywords, entities in the query results, key of query results, and dominant features in the order of their dominance scores compose the *IList*. Finally, the *Instance Selector* selects an instance of each item in the *IList* in their ranked order to build a snippet tree, using a greedy strategy. It aims at including as much information as possible in the snippet without exceeding a given size limit.

User study and performance evaluation showed that eXtract can effectively generate high-quality snippets for XML keyword search. Details of the experimental evaluation are presented in [3].

4. DEMONSTRATION

In the demonstration, we present eXtract, the first system that generates snippets for keyword search results on XML documents. The development of eXtract fills a gap in developing a full-fledged XML keyword search engine with functionalities from query result construction, ranking, to providing result snippets.

Compared with snippet generation in text documents, generating meaningful snippets within a size bound for XML search results is much more challenging: a NP-hard problem. This demonstration will present the challenges of searching tree-structured data, as well as some practical and effective solutions.

eXtract has a web-based user interface (http://eXtract.asu.edu/) which allows users to specify XML data sets and keywords for retrieval. It takes a query result and snippet size bound as input and efficiently generates meaningful result snippets within the size bound. Then the user can examine the snippets, and click the relevant ones for complete query results. eXtract is implemented in C++; and query results are presented on the web site with Apache and PHP, both on the Windows platform. A screen shot of eXtract is shown in Figure 5. Currently XSeek [6] is used as the XML keyword search engine to generate query results. Since snippet generation is orthogonal to query result generation and ranking, eXtract can also be used on top of any XML keyword search engines such as [1, 2, 4, 5, 7].

In our demonstration, we will show various example scenarios, such as movies and stores. Users can select an XML document, view it by clicking the "view data" button. They can issue keyword queries on the selected XML file as they normally do at any web search engine. We also provide users with the option of customizing the upper bound of snippet size, which is defined as the number of edges in the tree.

For example, as shown in Figure 5, a user issues a query "store texas", searching for the information about the stores in Texas, with a snippet size upper bound of 6. After the query results are produced, eXtract generates result snippets conforming to the size upper bound with a link to each query result. The user can easily judge whether a query result is of his/her interest by looking at the concise yet informative snippets. As we can see in the figure, the user can clearly see that the store named as *Levis* features *jeans*, especially for *man*; while the store named as *ESprit* focuses on the *outwear* clothes, mostly for *woman*.

For comparison purpose, we also present the snippets produced by Google Desktop for the generated XML keyword search results, on our web site. Since Google is a text document search engine and ignores XML tags and all structural information, the advantages of developing an XML-specific snippet generation system can be clearly demonstrated.

5. ACKNOWLEDGEMENT

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

6. REFERENCES

[1] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A semantic Search Engine for XML. In *Proceedings of VLDB*, 2003.

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

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

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

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

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

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