Data & Knowledge Engineering 124 (2019) 101758

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

XSnippets: Exploring semi-structured data via snippets

Mehdi Naseriparsa [a,*], Md. Saiful Islam [b], Chengfei Liu [a], Lu Chen [a]

[a] Department of Computer Science and Software Engineering, Swinburne University of Technology, Melbourne, Australia
[b] School of Information and Communication Technology, Griffith University, Gold Coast, Australia

ARTICLE INFO

Keywords:
Data exploration
Recommendations
XML

ABSTRACT

Users are usually not familiar with the content and structure of the data when they explore the data source. However, to improve the exploration usability, they need some primary hints about the data source. These hints should represent the overall picture of the data source and include the trending issues that can be extracted from the query log. In this paper, we propose a two-phase interactive exploratory search framework for the clueless users that exploits the snippets for conducting the search on the XML data. In the first phase, we present the primary snippets that are generated from the keywords of the query log to start the exploration. To retrieve the primary snippets, we develop an A* search-based technique on the keyword space of the query log. To improve the performance of computations, we store the primary snippet computations in an index data structure to reuse it for the next steps. In the second phase, we exploit the co-occurring content of the snippets to generate more specific snippets with the user interaction. To expedite the performance, we design two pruning techniques called *inter-snippet* and *intra-snippet* pruning to stop unnecessary computations. Finally, we discuss a termination condition that checks the cardinality of the snippets to stop the interactive phase and present the final Top-l snippets to the user. Our experiments on real datasets verify the effectiveness and efficiency of the proposed framework.

1. Introduction

Database systems are prevalent everywhere. Both experts and naive users inevitably need to interact with these complex systems to meet their information needs. However, it is very difficult for an ordinary user to explore a data source when s/he has no hint about its content and its structure. To improve the usability of the complex data systems, we argue that there should be an exploratory option to assist the clueless users in exploring the data source instead of prohibiting them from using the systems.

To set the problem area clearly, assume a user who does not have any experience or information about the movies, would like to search the *internet movie database* (IMDB), which is modeled as XML data, via *keywords*. It would be almost impossible for a clueless user to come up with appropriate keywords that could possibly retrieve some meaningful and interesting results. To provide a user-friendly environment for such users, the system could assist them by providing some hints (e.g., important keywords to summarize the underlying data source and provide an overall picture) to explore the database and initiate the search for them. These hints can be generated by consulting the expert users' experience, e.g., by analyzing the query log that stores previous queries. The query log analysis can extract the keywords that are used recently by the experienced users and can be considered as trending items for ensuring the recency of the exploration. However, to explore some meaningful and interesting results for certain user, we cannot solely use the extracted keywords of the query log, user feedback should also be reflected in the system generated results. Thus, we advocate providing an interactive environment for capturing user preferences that might be useful for personalizing the exploration in addition to query log analysis.

* Corresponding author.
E-mail addresses: mnaseriparsa@swin.edu.au (M. Naseriparsa), saiful.islam@griffith.edu.au (M.S. Islam), cliu@swin.edu.au (C. Liu), luchen@swin.edu.au (L. Chen).

https://doi.org/10.1016/j.datak.2019.101758
Received 24 April 2018; Received in revised form 14 April 2019; Accepted 18 October 2019
Available online 23 October 2019

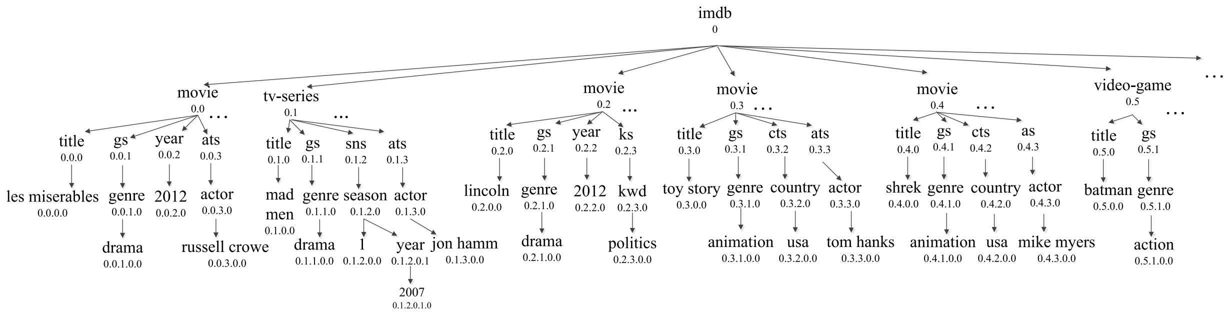


Fig. 1. A part of IMDB dataset.

Table 1
Movies retrieved from IMDB using 2 combination from $\mathcal{K}$ = {drama, animation, 2012, usa}.

k_1	k_3	#1	#2	#3	#4	#5	#6	...
genre = “drama”	year = “2012”	cloud atlas	magic mike	argo	life of pi	the master	savages	...
k_1	k_3	#7	#8	#9	#10	#11	#12	...
genre = “drama”	year = “2012”	the vow	mirror mirror	les miserables	lincoln	lawless	man with iron fist	etc.
k_2	k_4	#1	#2	#3	#4	#5	#6	...
genre = “animation”	country = “usa”	toy story	zootopia	frozen	moana	shrek	despicable me	...
k_2	k_4	#7	#8	#9	#10	#11	#12	...
genre = “animation”	country = “usa”	minions	tangled	cars	trolls	madagascar	aladdin	etc.

1.1. Motivation

Consider a part of IMDB data as shown in Fig. 1. Assume we have extracted the keyword pool $\mathcal{K}$ = {“*drama*”, “2012”, “*animation*”, “*usa*”, “*english*”} by analyzing the query log. Then, by using $\mathcal{K}$ we can produce some result hints as follows: the movies that their genre is “*drama*” and is produced in the year “2012”, the “*animation*” movies that are produced in the “*usa*”, the tv-series that their genre is “*drama*” and their language is “*english*”, and they are produced in the “*usa*” and so on. For each result hint, we can present a summary called snippet [1] to the user for exploration. Suppose the clueless user selects the snippets of the first two hints for exploration. Then according to the IMDB data source, there is a list of movies for each of the selected snippets as shown in Table 1. However, these snippets are general, and they contain an enormous number of results; thus, it is difficult for the user to find some interesting results from these snippets.

To navigate the exploration to the most interesting results, the system can generate more specific snippets by exploiting the co-occurring content of the snippets and increases the size of the snippets to narrow down the exploration space in several iterations. For instance, by adding the feature of movie *actor* “*crowe*” or the movie *plot* “*politics*” to the first snippet, or the movie feature of *actor* “*hanks*” to the second snippet, we can navigate the clueless user search to some interesting items. Finally, for the above example, the system can generate the following subtree snippets from the general primary snippets: r_1 = {[0.0], {*movie/gns/genre*, “*drama*”}, {*movie/year*, “2012”}, {*movie/ats/actor*, “*crowe*”}} by adding the feature *actor* “*crowe*” (this subtree retrieves the “*drama*” movies that is produced in “2012” and “*russell crowe*” is the *actor* of those movies such as “*the man with iron fist*” or “*les miserables*”), r_2 = {[0.2], {*movie/gns/genre*, “*drama*”}, {*movie/year*, “2012”}, {*movie/ks/kwd*, “*politics*”}, } (this subtree result retrieves the “*drama*” movies that is produced in the year “2012” and their plot keyword is about “*politics*” such as *lincoln*), and r_3 = {[0.3], {*movie/gns/genre*, “*animation*”}, {*movie/cts/country*, “*usa*”}, {*movie/ats/actor*, “*hanks*”}} (this subtree retrieves the “*animation*” movies that is produced in the “*usa*” and “*tom hanks*” is the *actor* of those movies.

1.2. Our work

Though, data exploration is studied extensively in the literature ([2–4], and [5] for survey), there is barely any work that proposes a solution for exploring the XML data. Kalinin, Cetintemel and Zdonik [3] employed semantic windows for the users to set their favorite conditions and perform exploration queries. Qarabaqi and Riedewald [2] proposed exploratory search using probability distribution over the entities attributes. Therefore, the users are helped to specify right conditions on imprecise queries. Drosou and Pitoura [5] presented a database exploration framework which recommends additional items called ‘You May Also Like’ results. However, most of these works focus on relational databases. The XML keyword search is studied extensively in the last decade ([6,7] for survey). Liu, Huang and Chen [1] exploit result snippets to improve the XML search results. However, there is no work in semi-structured data like XML that tackles data exploration by interaction with the user. In this paper, we propose to use subtree snippets to explore XML data where each subtree snippet summarizes a part of the data source. However, these subtree snippets must be diverse enough to reflect the overall picture of the data source. We propose a two-phase interactive framework to navigate the exploration on the XML data via snippets. In the first phase, we exploit the keywords of the query log to generate primary snippets. Then, in the second phase, we process the user chosen snippets, terminate and present the Top-l snippets in this phase.

1.3. Challenges

There are some challenges for computing the exploratory subtree snippets in XML data. Assume there are $|\mathcal{K}|$ unique keywords in the query log, Then, in the worst case scenario, we can generate $2^{|\mathcal{K}|} - (|\mathcal{K}| + 1)$ subtree result snippets. Thus, it is impractical to compute and present all of the subtree snippets to the clueless user in the first phase. Hence, we propose to rank the subtree snippets based on their interest and novelty w.r.t. the data source and only present the Top-ranked snippets. We design a novel A* search-based technique to perform an informed search on the keywords for generating the snippets efficiently. However, the primary snippets are not specific, and they contain many results. Thus, in the interactive phase and in each iteration the user chooses some snippets for processing and the system generates more specific snippets to navigate the user in the exploration. To generate the more specific snippets, we exploit the co-occurring content of the snippets and add more features from the co-occurring content of the snippets in each iteration. Thus, the final subtree snippets should be generated by considering all possible combination of the primary snippets content with the remaining or co-occurring part that is not reflected in the snippet. Assume d is the maximum number of features that could be added to the primary subtree snippet to generate the ultimate subtree snippet, and $|\widehat{\mathcal{K}}|$ is the average number of keywords for each of d features. Then, we can generate $2^d \times |\widehat{\mathcal{K}}|^d$ subtree snippets. Due to this combinatorial explosion and with the fact that each snippet should be analyzed and ranked in terms of its interest and novelty, we present two pruning techniques called *inter-snippet* and *intra-snippet* pruning to efficiently avoid the unnecessary computations in the interactive phase. Finally, we check a termination condition at each iteration, and if it satisfies the condition, we stop the iterations and present the Top-l highly ranked subtree snippets to the user.

1.4. Contribution

Our main contributions in this paper are as follows:

1. We are the first to formulate and design a framework for the interactive exploration of XML data via subtree snippets for the scenario that the user is clueless.
2. We propose A* search-based technique to generate top snippets for the user as primary hints to start the exploration. Moreover, we propose an indexing structure to store the primary snippet computations and to reuse them for the next steps.
3. We propose two pruning techniques called *inter-snippet* and *intra-snippet* pruning for the interaction phase to expedite the interactive snippet processing.
4. We conduct extensive experiments on two real datasets to verify the effectiveness and efficiency of our proposed approach.

1.5. Organization

The rest of the paper is organized as follows: In Section 2, we discuss the preliminaries and the problem. Section 3 presents the details of the primary snippet generation and the A* search-based technique. In Section 4, we discuss the snippet processing, pruning techniques and the cardinality stop condition for finalization of the snippets processing. Section 5 presents the experiments. Section 6 presents the related work. Finally, Section 7 concludes the paper (see Table 2).

2. Background

2.1. Preliminaries

An XML document is a tree data structure T that contains some nodes with labels and a designated root. For each node in the tree T, there is an index that is a unique identifier called Dewey code, which consists the path from the root to the corresponding node. In Fig. 1, the Dewey code 0.0 refers to a node in the tree that contains information about the $les\ miserables$ movie. A node m_i in the tree T that contains keyword k_i is a match node for k_i. E.g, from Fig. 1 we can observe the match node of the keyword $k_i = politics$ is $m_i = [0.2.3.0.0]$.

Subtree result. Given a set of keywords $\{k_1, \dots, k_n\}$, a result $r^T = (v_{slca}, \{m_1, m_2, \dots, m_n\})$ is a subtree in T which contains all keywords k_i, $i \in [1-n]$. Here we employ the established SLCA semantics [7,8]; thus, v_{slca} is the smallest lowest common ancestor (SLCA) of the match nodes $m_i, \forall i \in [1-n]$. However our solutions are independent of the search semantics. Thus, it is possible to use other kinds of SLCA-based semantics [6,9,10] for the search.

Subtree result features. Given a subtree result $r^T = (v_{slca}, \{m_1, m_2, \dots, m_n\})$, each $f_i = \{m_i.type, \text{“}k_i\text{”}\}$, $i \in [1-n]$ represents a feature for r^T where $m_i.type$ is the type of the match node m_i with respect to subtree result structure and k_i is the associated keyword for m_i. E.g., for $r^T = \{[0.0], \{[0.0.1.0.0], [0.0.2.0]\}\}$, we have $f_1 = \{movie/gs/genre, \text{“}drama\text{”}\}$.

Table 2
The list of symbols.

Symbol	Meaning
f	A feature
$\mathcal{F}$	A set of features
$\mathcal{H}$	A max heap
k	A query keyword
$\mathcal{K}$	Keyword pool
r	A snippet
$\mathcal{R}$	A set of snippets
$\mathcal{R}^1$	A set of Top-w primary snippets (first iteration)
$\mathcal{R}^i$	A set of Top-w snippets (ith iteration)
$\mathcal{R}*$	A set of Top-l snippets
T	XML data
v_{slca}	A subtree result root
$\mathcal{V}$	A set of root nodes
w	Window size
$IRF(f,T)$	Inverse result frequency of feature f
$\lambda(r,T,\mathcal{R}^1)$	The score of primary snippet r
$\lambda(r,T,\mathcal{R}^i)$	The score of snippet r
$\lambda^{int}(r,T)$	Interestingness score of snippet r
$\lambda^{nov}(r,\mathcal{R})$	Novelty score of snippet r
λ^{min}	Threshold score

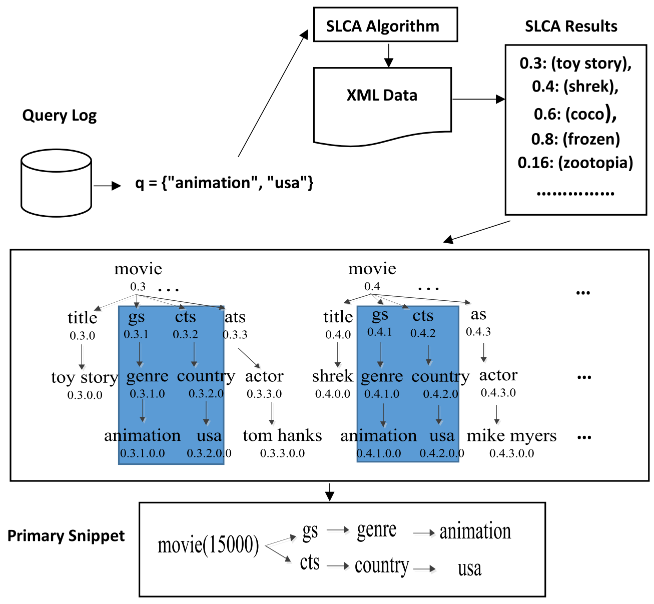


Fig. 2. Primary snippet generation.

2.2. Problem statement

One way to assist a clueless user in exploring the XML data is to analyze the data source and to recommend her some primary result snippets to set her exploration preferences. The primary snippets are generated by using trending keywords $\mathcal{K}$ from the query log to incorporate the recent trending in the exploration process. To generate the primary snippets, we combine the trending keywords of the query log issued by the expert users to generate keyword queries. Then, we execute these queries against the data source to compute the subtree results that contain the keywords. To find the high trending and interesting keywords for the clueless user, we exploit the statistics of the keywords in the query log such as the frequency of the keywords. Assume M is a set of match nodes for a keyword $k \in \mathcal{K}$ w.r.t. the data source T. Then the XML result snippet is defined as follows:

Definition 1 (*XML Subtree Snippet*). An XML subtree snippet $r = \{\mathcal{V}, \mathcal{V}.type, f_1, \dots, f_n\}$ is a set of subtree results that have the following common features: (a) $\mathcal{V}.type$ that is the type of the root nodes of the subtrees, (b) $\{f_i\}$, $i \in [1,n]$ that is the type of the ith match nodes for k_i in the subtrees. The keyword k_i, $i \in [1-n]$ is a leaf node in the subtree results and $\mathcal{V}$ is the set of root nodes.

Fig. 2 presents the details on how we produce a primary snippet from XML document. First, we generate a query from the keyword pool that is stored in the query log (as discussed in Section 3.1). Then, we run SLCA keyword search algorithm on XML data to produce the potential root nodes that contain all the query keywords within the XML data. For instance, after processing the query $q = \{$"*animation*", "*usa*"$\}$, we obtain the following root nodes with their respected movie titles: [0.3] : "*toystory*", [0.4] : "*shrek*", [0.6] : "*coco*", [0.8] : "*frozen*", [0.16] : "*zootopia*", and so on. For presenting the primary snippet, we extract the common features between these results which is highlighted in the figure, i.e., $f_1 = \{movie/gs/genre,$ "*animation*"$\}$ and $f_2 = \{movie/cts/country,$ "*usa*"$\}$. In Fig. 2, the produced primary snippet represents a hint that contains 15 000 movies that their genre is "*animation*" and those movies are produced in the "*usa*".

For example for the keyword pool $\mathcal{K} = \{drama, 2012, usa\}$ on IMDB data source, the value "*drama*" refers to a movie genre, "2012" refers to a movie production year, and "*usa*" refers to a movie production country. Therefore, we can generate 3 XML result snippets with the size of 2 such as: $r_1 = \{\mathcal{V} : 4026, movie, \{movie/gns/genre,$ "*drama*"$\}, \{movie/year,$ "2012"$\}\}$, $r_2 = \{\mathcal{V} : 7452, movie, \{movie/gns/genre,$ "*drama*"$\}, \{movie/\ cts/country,$ "*usa*"$\}\}$, $r_3 = \{\mathcal{V} : 5120, movie, \{movie/year,$ "2012"$\}, \{movie/cts/country,$ "*usa*"$\}\}$. The result cardinality of the snippet is separated by a colon from $\mathcal{V}$ that is the set of root nodes. We use the primary XML result snippets $\mathcal{R}^1 = \{r_1, r_2, \ldots, r_{|\mathcal{R}|}\}$ for the clueless user to set exploratory preferences for the exploration. However, there are an enormous list of snippets that we can retrieve from the query log after building the keyword pool and generating the queries. Moreover, many of these snippets are not suitable because they contain repetitive information. To retrieve the primary snippets that cover most of the data source as the overall picture of the data and are interesting with respect to the data source, we compute the most interesting and diversified XML result snippets based on the following equation.

$$\lambda(r, T, \mathcal{R}^1) = \lambda^{int}(r, T) \times \lambda^{nov}(r, \mathcal{R}^1) \tag{1}$$

In Eq. (1), $\lambda^{int}(r, T)$ measures the interestingness of the XML snippet r with respect to the data source T. We use TF–IDF to define and measure the intersettingness of an XML snippet. Therefore, a snippet r is interesting when it has a large number of occurrences in a document within the data source T and when a small number of documents in T contain the snippet r (as discussed in detail in Section 3.1). $\lambda^{nov}(r, \mathcal{R}^1)$ measures the novelty of r content among the result snippets $\mathcal{R}^1$. To provide the user with different primary snippets, we define and measure the novelty as the difference between the snippets content. For instance, a snippet content is associated to its results which is in the form of the its root nodes. Thus, a snippet is regarded as novel if it contains root nodes which do not exist in the current snippet list (as discussed in detail in Section 3.1). Assume there are $|\mathcal{K}|$ unique keywords in the query log. Then we can generate a snippet for each keyword query that is produced from $|\mathcal{K}|$. Thus, we can generate up to $2^{|\mathcal{K}|} - (|\mathcal{K}| + 1)$ primary snippets. However, not all these snippets are interesting and informative to the same degree. Furthermore, it is impractical to compute and present all of the primary snippets to the clueless user for exploration. That is why we compute the most interesting and novel primary snippets (Top-w) to present to the user. Assume w is the size of the window.

Definition 2 (*Top-w Primary XML Subtree Snippets*)**.** Given a set of XML subtree snippets $\mathcal{R}$, the Top-w most interesting and diversified XML result snippets $\mathcal{R}^1 = \{r_1, r_2, \ldots, r_w\}$, $w \le |\mathcal{R}|$ maximize the scoring function presented in Eq. (1).

The parameter w may be set by the system. Since the Top-w snippets should provide an overall picture for a clueless user, a big w may confuse the user. That is because a clueless user needs only a few hints about the trending and interesting items to begin her data exploration. Moreover, a big w increases the probability of adding more similar and less informative snippets to the Top-w primary snippets which is useless. Furthermore, a big w incurs heavy computational costs. In this paper, we found $w = 10$ as the best fit for this parameter. However, we proposed an A* search approach with an upper bound estimation to compute the Top-w primary snippets that avoids the unnecessary computations and improves the performance. Thus, our approach is able to accept other values without facing any issue; e.g., $w > 10$.

We generate the primary snippets in the offline mode to present an overall picture of the data source for the clueless user. After that, we interact with the user to navigate her in the exploration process until we stop the interaction and present the final snippets to the user. When the user chooses some of the primary snippets $\mathcal{R}^1$, the system navigates her into more specific items by focusing on the selected snippets and by adding more content to these snippets iteratively. Since the primary subtree snippets are general and they contain many results, we interactively tighten the exploration to more specific items in some iterations. To make the exploration more specific, we narrow down the search on user-chosen snippets and we tighten the results based on the hidden content of the snippets called co-occurring features of the snippets. Therefore in each iteration, we add more features to each of the user chosen-snippets to generate a set of more specific subtree snippets with a bigger size. Assume $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\}$ denotes the set of features in the snippet r.

Definition 3 (*Co-occurring Feature*)**.** Given a snippet $r = \{\mathcal{V}, \mathcal{V}.type, \mathcal{F}\}$, a co-occurring feature $f \in \overline{\mathcal{F}}$ is (a) $f \notin \mathcal{F}$, and (b) $\exists v \in \mathcal{V}$ such that v is an ancestor for f denoted by $v \prec_a f$.

For example, for the subtree snippet r_1 presented in Fig. 3, we may generate a more specific subtree snippet $r_1 = \{[0.0], movie, \{movie/gs/genre,$ "*drama*"$\}, \{movie/\ year,$ "2012"$\}, \{movie/ats/actor,$ "*crowe*"$\}\}$ by adding the co-occurring feature $f = \{movie/ats/actor,$ "*crowe*"$\}$.

Definition 4 (*Exploratory Snippet*)**.** Assume $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ is the set of features of the subtree snippet r and $\overline{\mathcal{F}}$ is a subset of the co-occurring features. Then, $r = \{\mathcal{V}, \mathcal{V}.type, \mathcal{F} \cup (f \in \overline{\mathcal{F}})\}$ is an exploratory snippet if $\forall v \in \mathcal{V}, v \prec_a (\mathcal{F} \cup f)$.

However, we can generate an enormous number of exploratory subtree snippets in each iteration. That is because any combination of the co-occurring features set with the snippet feature set may generate an exploratory snippet that should be

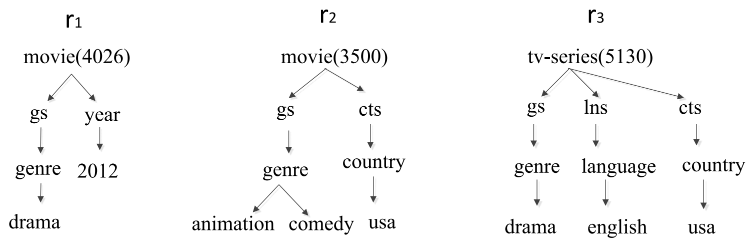


Fig. 3. XML subtree snippets.

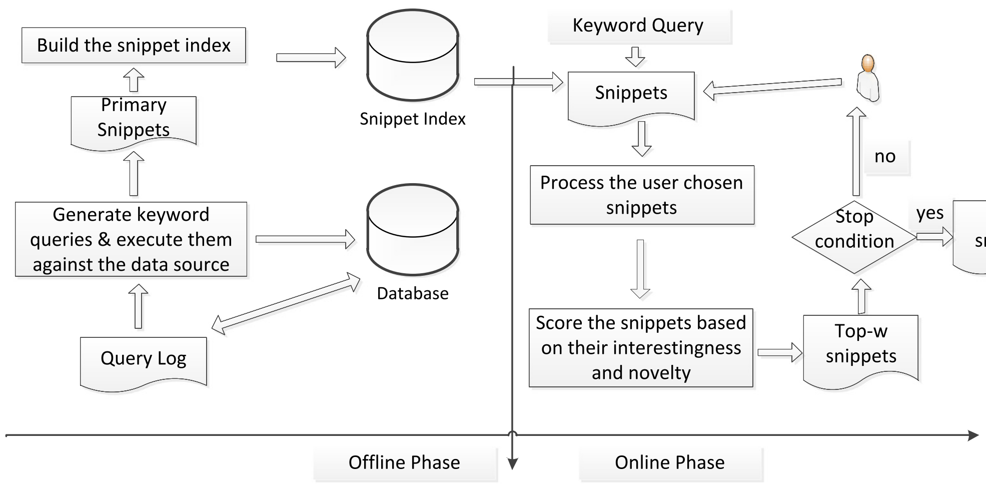


Fig. 4. XSnippets architecture overview.

considered. Thus, we need to limit the number of subtree snippets to the Top-w highly ranked snippets in each iteration. We finally propose to score and rank the subtree snippets in each iteration based on the following:

$$\lambda(r,T,\mathcal{R}^i) = \frac{\lambda^{int}(r,T) \times \lambda^{nov}(r,\mathcal{R}^i)}{\log_2(1+|r|)}, i > 1 \tag{2}$$

where $|r|$ is the result cardinality of the snippet r. We incorporate the result cardinality into the scoring equation in each iteration to navigate the exploration to the convergence point quickly. That is when a snippet contains many results, we penalize it by $\log_2(|r|)$ factor and give priority to the snippets with smaller cardinality because they contain specific items.

Definition 5 (*Top-l Exploratory Snippets*). Given a set of snippets $\mathcal{R}^i = \{r_1,\dots,r_n\}$, $n \le w$, the problem of Top-l exploratory snippets is to compute l highly ranked exploratory snippets $\mathcal{R}^*$ such that for $r \in \mathcal{R}^*$, $\lambda(r,T,\mathcal{R}^i)$ is maximized, i.e., $\mathcal{R}^* = \{|\mathcal{R}^*| = l \mid \forall r \in \mathcal{R}^*, r' \notin \mathcal{R}^*, \lambda(r,T,\mathcal{R}^i) \ge \lambda(r',T,\mathcal{R}^i)\}$.

2.3. Solution overview

We propose a two-phase solution for interactive data exploration in XML data via snippets as given as follows:

1. **Primary snippet generation:** This step finds the Top-w snippets from the dataset that represent the most trending and interesting items of the data source.
2. **Interactive snippet processing:** This step interacts with the user in several rounds to extract the user preferences. In each iteration, the user selects a set of the snippets and the system processes the selected snippets and presents the more specific snippets to the user. At the end of this step, the system stops the interactive snippet processing and presents the Top-l subtree snippets to the user.

The XSnippets architecture overview is depicted in Fig. 4. From the figure, the XSnippets includes offline and online modes. The offline mode prepares the primary snippets by executing trending keyword queries against the data source and builds the index for these snippets. In the online mode, the system iteratively adds more keywords to the user chosen snippets, rank the newly generated snippets and present the Top-w snippets to the user for evaluation. At last, it displays the Top-l snippets to the user when the stop condition satisfies.

3. Primary snippet generation

A snippet is composed of different features that contain keywords. Thus, to generate the primary snippets we need to extract some keywords that represent the interesting and trending items in the data source. Moreover, these snippets should cover different parts of the data to represent the overall picture of the data to the user. To extract these keywords, we collect the statistics of the keywords with respect to the query log. Then, we build a keyword pool that stores the most interesting and trending keywords and run some keyword queries generated from the keyword pool to generate the snippets. Afterward, we compute Top-w highly ranked snippets for the user.

3.1. Building the keyword pool

The keyword pool $\mathcal{K}$ is the key source to generate the primary subtree snippets. We extract the keywords by considering the query log that contains the queries issued by expert users. For computing the trending keywords from the query log, we use the keyword frequency as follows: $score(k, Log) = \log_2(1 + freq_{(k)})$ where Log represents the query log data and $freq_k$ is the number of occurrences of k in the log. We select the keywords that their score is higher than a threshold and store them into $\mathcal{K}$. This threshold is set using the maximum frequency of the query log keywords. For instance in this paper, we observe that we get the best keywords by setting the threshold to something near to the average frequency of the query log keywords. This would filter the keywords with less frequency. However, the threshold should be set considering the keyword query log size. For example when the query log size is small, we should loosen the threshold to include more keywords in the pool and vice versa. After building the keyword pool $\mathcal{K}$, we select the combination of keywords from $\mathcal{K}$ to generate some keyword queries. Then, we execute each keyword query q from $\mathcal{K}$ against the data source T to retrieve the primary subtree snippets. However, not all the possible queries are suitable to generate meaningful subtree snippets. That is because some queries from $\mathcal{K}$ would generate erroneous or mismatch subtree snippets based on the XML data structure. Moreover, when the query size gets larger, it generates more specific snippets that would restrict the exploration space for the user. Therefore, we check the mismatch problem [11] and put a limit for the primary snippets size to generate the subtree snippets. Since the number of generated primary snippets is enormous, we score the subtree snippets $\mathcal{R}$ based on their interestingness and their novelty with respect to the data source. This scoring guarantees that the primary snippets contain important items and cover different parts of the data. Then to start the exploration, we present the Top-w subtree snippets $\mathcal{R}^1$ to the user to set her exploration preferences.

The interestingness of a subtree snippet r is dependent on each of its constituent features $\forall f \in r$. According to the TF–IDF method [12], a feature is interesting if it has a large number of occurrences in the data source T called feature Frequency in the Results (FR) and when a large number of documents in T reflected the feature called Inverse Result Frequency (IRF). We make the similar adaptation of [12] to this formula to apply it to our problem. Thus, we assume the FR equals to 1 and compute the IRF to reflect the interestingness of f as follows:

$$IRF(f, T) = \frac{|T|}{freq_{(f.k)}^{(f.type)}} \tag{3}$$

where $freq_{(f.k)}^{(f.type)}$ returns the number of $f.type$ nodes in T that contain the keyword $f.k$ in their leaf nodes that is presented in the following formula:

$$\lambda^{int}(r, T) = \log_2(1 + \prod_{\forall f \in r} IRF(f, T)) \times \phi^d \tag{4}$$

where d returns the count of edges in the subtree snippet r, ϕ is a reduction factor between (0–1], and $freq_{(f.k)}^{(f.type)}$ is the frequency of the feature f with type of $type$ under the value node of $f.k$.

We modified the TF–IDF method to adapt it over XML data. To make this adaptation, $\lambda^{int}(r, T)$ contains two primary parts: (a) IRF which is an adapted version of IDF to compute the inverse result frequency over XML data which reflects the XML data content, and (b) the XML tree structure ϕ^d which reflects the XML data structure.

In order to present the user maximum different primary options, the Top-w subtree snippets should be informative and cover the different part of the data source. Thus, we compute the novelty of the snippets according to the following equation:

$$\lambda^{nov}(r, \mathcal{R}) = \log_2(1 + (|\{r.\mathcal{V}\} \setminus \{\mathcal{R}.\mathcal{V}\}|)) \tag{5}$$

where $r.\mathcal{V}$ and $\mathcal{R}.\mathcal{V}$ return the set of root nodes of r and $\mathcal{R}$ respectively.

For the novelty, a snippet is regarded as novel if it contains contents which do not exist in the current snippet list. Therefore to define the novelty over XML data, we assess a snippet content based on its results. The snippet results are in the form of the containing root nodes within r over XML data. Thus to assess the results, we adapted the novelty definition $\lambda^{nov}(r, \mathcal{R})$ over XML data by using the difference between the set of root nodes in the current Top-w list and the snippet r.

Example 1. Consider the snippet r_1 presented in Fig. 3 that contains the features $f_1 = \{movie/gns/genre, \text{“}drama\text{”}\}$, and $f_2 = \{movie/year, \text{“2012”}\}$. Given $\phi = 0.8$, $IRF(f_1, T) = 350$, and $IRF(f_2, T) = 1050$, we compute the score of snippet r_1 presented in Fig. 3 as follows: (a) $\lambda^{int}(r_1, T) = \log_2(1 + 350 \times 1050) \times 0.8^5 = 6.1$, (b) $\lambda^{nov}(r_1, \mathcal{R}) = \log_2(1 + 800) = 9.65$. Finally, we compute the total score $\lambda(r_1, T, \mathcal{R}) = 6.1 \times 9.65 = 58.87$.

3.2. A-search based technique*

To generate the Top-w primary snippets, we have to execute all the queries generated from $\mathcal{K}$ against T that is practically impossible. Thus, we exploit an A*-search based algorithm to perform an informed search on the keyword pool $\mathcal{K}$ space and efficiently rank the snippets without running all the queries. The search space includes $\mathcal{K}$, and in each step, we add to the size of snippets by using the keywords $\mathcal{K}$ and generate new snippets. The A*-search continues until we find by estimation that adding to the size of a snippet r, will not contribute the snippet to be on the Top-w list $\mathcal{R}^1$. To estimate that r can beat the bottom line, we compute the upper bound score of r. Assume $\overline{IRF}$ is the maximum value for Inverse Result Frequency among the co-occurring features $f \in \overline{\mathcal{F}}$, $min(d)$ returns the minimal number of edges d of the snippet r. Then, the maximal score $\overline{\phi^d} = \phi^{min(d)}$. If c is the number of features that should be added to generate a snippet with maximum size then the upper bound score for interestingness of r is computed as follows:

$$\overline{\lambda}^{int}(r,T) = \log_2(1 + \prod_{\forall f \in r} IRF(f,T) \times \overline{IRF}^{c}) \times \overline{\phi^d} \tag{6}$$

Also, the novelty score of snippet r is maximal when all of its constituent results are new in $\mathcal{R}$. Therefore, the upper bound score for the novelty is computed by the following equation:

$$\overline{\lambda}^{nov}(r,\mathcal{R}) = \log_2(1 + |r.\mathcal{V}|) \tag{7}$$

Finally, by using the upper bound of the two scoring factors of Eq. (1), we compute the total upper bound score $\overline{\lambda}(r,T,\mathcal{R})$.

Lemma 1. *The upper bound score $\overline{\lambda}(r,T,\mathcal{R})$ is computed by using the upper bound of the two scoring factors of Eq. (1) as follows:*

$$\overline{\lambda}(r,T,\mathcal{R}) = \overline{\lambda}^{int}(r,T) \times \overline{\lambda}^{nov}(r,\mathcal{R}) \tag{8}$$

Example 2. Consider the snippet r_1 presented in Fig. 3. Given the following information: $\overline{IRF} = 10^6$, $min(d) = 8$, $\overline{\phi^d} = 0.8^8 = 0.17$, we compute the upper bound score of the snippet r_1 that is presented in Fig. 3 as follows: (a) $\overline{\lambda}^{int}(r_1,T) = \log_2(1{+}350{\times}1050{\times}(10^6)^3){\times}0.17 = 13.31$, (b) $\overline{\lambda}^{nov}(r_1,\mathcal{R}) = \log_2(1+4026) = 11.98$. Finally, $\overline{\lambda}(r_1,T,\mathcal{R}) = 13.31 \times 11.98 = 159.45$.

To ensure the correctness of the A*-search algorithm, the upper bound score $\overline{\lambda}(r,T,\mathcal{R})$ should guarantee that it does not underestimate the score of r.

Lemma 2. *Given $\lambda(r^*,T,\mathcal{R})$, $\overline{\lambda}(r',T,\mathcal{R})$ such that $\forall r'' \in \mathcal{R}$, $\overline{\lambda}(r',T,\mathcal{R}) \geq \overline{\lambda}(r'',T,\mathcal{R})$. The stop condition $\lambda(r^*,T,\mathcal{R}) \geq \overline{\lambda}(r',T,\mathcal{R})$ retrieves the optimal r^* if $\overline{\lambda}(r',T,\mathcal{R})$ is admissible.*

Proof. The estimation of $\overline{\lambda}(r',T,\mathcal{R})$ is admissible. Then, we have $\overline{\lambda}(r',T,\mathcal{R}) \geq \lambda(r',T,\mathcal{R})$. We know that $\overline{\lambda}(r',T,\mathcal{R})$ is the maximum value among other estimations. Therefore using the admissibility we have following: $\forall r'' \in \mathcal{R}, \lambda(r'',T,\mathcal{R}) \leq \overline{\lambda}(r',T,\mathcal{R}) \leq \lambda(r^*,T,\mathcal{R})$; thus, r^* is the optimal snippet.

3.2.1. The algorithm

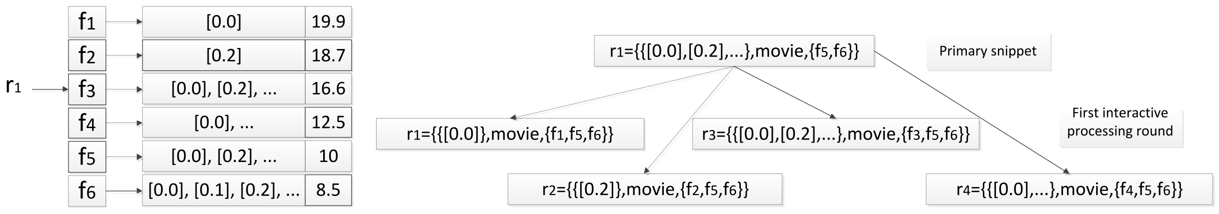
Algorithm 1 presents the A* search-based technique for generating the Top-w primary snippets. In line 1, we build the keyword pool $\mathcal{K}$ from the query log trending keywords. In lines 2–4, we initialize the heap $\mathcal{H}$ by pushing the entries that are generated by using the keywords of $\mathcal{K}$ into the heap $\mathcal{H}$. Each heap entry $e \in \mathcal{H}$ contains the following information: (a) $e.q$ that contains the keyword list of the snippet, (b) the snippet $e.r$, (c) $e.r.\lambda$ that is the score of the snippet, and (d) $e.r.\overline{\lambda}$ that is the upper bound score estimation of the snippet. In line 6 we pop the top entry into e. In lines 8–9, we fill the Top-w primary snippets list with the currently popped snippet $e.r$ if the list is not filled yet (checking if $|\mathcal{R}^1| < w$). Otherwise, we update the minimum threshold score λ^{min} with the minimum possible score within the Top-w primary snippets list $\mathcal{R}^1.min$. We stop processing the entries if the upper bound score estimation of the current entry cannot beat the bottom line score (if $e.r.\overline{\lambda} < \lambda^{min}$) in line 13 based on Lemma 2. In lines 14–15, we update the primary snippet list $\mathcal{R}^1$ if the current entry snippet $e.r$ is promising that means its score beats the bottom line score (by checking the condition $e.r.\lambda > \lambda^{min}$). In lines 18–19, we generate new entry e' by adding a keyword to the snippet keyword list $e'.q$, and generate the corresponding snippet $e'.r$. Then, we measure the newly generated snippet score $e'.r.\lambda$ (based on Eq. (1)), its upper bound score $e'.r.\overline{\lambda}$ (based on Eq. (8)), and push it into the heap $\mathcal{H}$ as pseudocoded in lines 20–22. In line 23, we return the Top-w primary snippets $\mathcal{R}^1$ to the user.

3.3. Indexing the primary snippets

The primary snippet generation is done in the offline mode to provide the clueless user with the data source overall picture and the most trending data regions. Conversely, in the interactive snippet processing, we interact with the user iteratively in the online mode that incurs heavy computational costs. Moreover, the primary snippet computations are the main source that is repeatedly used during the interactive exploration process. Therefore, we can reuse the primary snippet computations to expedite the performance of the interactive step. To exploit the primary snippet computations, we propose an index structure called Snippet BTree (*SBTree*) to store the information of the primary subtree snippets. The primary snippets contain the following information for each snippet r: (a) the root nodes of the snippet $\mathcal{V}$, (b) the features of the snippet $\mathcal{F}$, and (c) the IRF score of the features $\mathcal{F} \in r$. Similar to *BTree* that is used by [6] for XRank, we design the *SBTree* as follows: each feature $f \in \mathcal{F}$ is the key of the *BTree*, and the root nodes of the snippet is the value for the key. Moreover, the IRF of the feature is added to each entry of the tree and the entries are sorted based on $IRF(f,T)$. The indexing structure for the entry r_1 of Fig. 3 is depicted in Fig. 5.

Algorithm 1: Generating Top-w primary snippets

```
Input  : $T$, $Log$, $w$
Output: Top-$w$ Primary XML Subtree Snippets $\mathcal{R}^1$
1  $\mathcal{K} \leftarrow generateKeywordPool(T, Log)$;                // initialize keyword pool $\mathcal{K}$
2  while $k \leftarrow getNext(\mathcal{K}) \neq \emptyset$ do
3      $e.q \leftarrow k$ ;                                             // set the entry keyword using $k$
4      $\mathcal{H}.push(e)$;                                           // initialize heap $\mathcal{H}$ using generated entry $e$
5  while $\mathcal{H} \neq \emptyset$ do
6      $e \leftarrow \mathcal{H}.pop()$ ;                                // pop the top entry $e$ for next steps
7      if $|e.q| > 1$ then
8          if $|\mathcal{R}^1| < w$ then
9              $\mathcal{R}^1 \leftarrow \mathcal{R}^1 \cup e.r$;            // add $r$ to $\mathcal{R}^1$ since $\mathcal{R}^1$ is not filled yet
10         else
11             $\lambda^{min} \leftarrow \mathcal{R}^1.min$;                 // update $\lambda^{min}$ with minimum score in $\mathcal{R}^1$
12             if $e.r.\overline{\lambda} < \lambda^{min}$ then
13                 break;                                               // as per Lemma 2
14             else if $e.r.\lambda > \lambda^{min}$ then
15                 $update(\mathcal{R}^1, r)$;                           // update $\mathcal{R}^1$ since $r$ beats the bottom entry
16     if $|e.q| < 3$ then
17         while $k \leftarrow \mathcal{K} \neq \emptyset$ and $k \cap e.q = null$ do
18             $e'.q \leftarrow e.q \cup k$;                             // increase entry query $e.q$ size using $k$
19             $e'.r \leftarrow generateSnippet(e'.q, T)$;
20             $e'.r.\lambda \leftarrow score(e'.r, T, \mathcal{R}^1)$;       // compute score of $e'$ using Eq. (1)
21             $e'.r.\overline{\lambda} \leftarrow bound(e'.r, T, \mathcal{R}^1)$;  // compute bound of $e'$ using Eq. (8)
22             $\mathcal{H}.push(e')$;                                   // push extended entry $e'$ into $\mathcal{H}$ for next steps
23 return $\mathcal{R}^1$;
```


f1
f2
r1
f3
f4
f5
f6
[0.0]
19.9
[0.2]
18.7
[0.0], [0.2], ...
16.6
[0.0], ...
12.5
[0.0], [0.2], ...
10
[0.0], [0.1], [0.2], ...
8.5
r1={{[0.0],[0.2],...},movie,{f5,f6}}
Primary snippet
First interactive processing round
r1={{[0.0]},movie,{f1,f5,f6}}
r3={{[0.0],[0.2],...},movie,{f3,f5,f6}}
r2={{[0.2]},movie,{f2,f5,f6}}
r4={{[0.0],...},movie,{f4,f5,f6}}

(a) Indexing structure for snippet r_1 (b) Processing snippet r_1 by using the index

Fig. 5. Snippet processing by using indexing structure.

4. Interactive exploratory snippet processing

In this step, we capture the user exploration preferences in various rounds. The goal of this step is to narrow down the exploration space until the results are specific enough for the user. At each iteration, after the user chooses the preferred snippets, we refine the list of snippets by generating more specific snippets that have a bigger size. We refine the snippets by selecting co-occurring features for each snippet and add it to the existing snippet to generate the more specific snippets. Finally, at each iteration, we rank the newly generated snippets to present to the user. The snippet processing operations for the r are as follows: (a) we read a feature from the co-occurring features $f \in \overline{\mathcal{F}}$ and add it to the features of r; (b) we update the list of root nodes for r by using a set intersection operation on the new feature set $f \cup \mathcal{F}$; and (c) we measure the score of the snippet r after adding f to its feature set.

Example 3. Consider the snippet r_1 presented in Fig. 3. Then, the index of the primary snippet r_1 is depicted in Fig. 5.a, and the processing steps of r_1 is presented in Fig. 5.b. From the figure, the co-occurring features include $f_1 = \{movie/title, \text{“}miserables\text{”}\}$, $f_2 = \{movie/title, \text{“}lincoln\text{”}\}$, $f_3 = \{movie/kws/keyword, \text{“}politics\text{”}\}$, $f_4 = \{movie/ats/actor, \text{“}crowe\text{”}\}$, and the primary snippet contains the features $f_5 = \{movie/year, \text{“2012”}\}$, and $f_6 = \{movie/gns\ /genre, \text{“}drama\text{”}\}$. To process snippet r_1, we add one feature from the co-occurring features $\overline{\mathcal{F}} = \{f_1, f_2, f_3, f_4\}$to the feature set $\mathcal{F} = \{f_5, f_6\}$ of r_1. Finally, for the first step, we have the following newly generated snippets: $r_1 = \{\{[0.0]\}, movie, \{f_1, f_4, f_5\}\}$, $r_2 = \{\{[0.2]\}, movie, \{f_2, f_4, f_5\}\}$, $r_3 = \{\{[0.0], [0.2], \ldots\}, movie, \{f_3, f_4, f_5\}\}$, and $r_4 = \{\{[0.0], \ldots\}, movie, \{f_4, f_4, f_5\}\}$.

4.1. Snippet processing complexity

Assume $|\mathcal{R}^i|$ is the number of snippets, $|\mathcal{F}|$ is the average number of features of the snippets, and c is the cost of the set intersection of the features. Then, the complexity of the snippet processing would be $\mathcal{O}(|\mathcal{R}^i|\times|\mathcal{F}|\times c)$. However, the interactive snippet processing is carried out in the online mode. Thus, we provide two pruning techniques called *inter-snippet pruning* and *intra-snippet pruning* to expedite the performance and efficiently reduce the sizes of $|\mathcal{R}^i|$ and $|\mathcal{F}|$ respectively.

4.1.1. Inter-snippet pruning

We provide a mechanism to stop unnecessary processing of the non-promising snippets. The *inter-snippet pruning* checks whether a snippet can beat the bottom line score when we process it. If not, we stop processing the snippets and return the Top-w snippets to the user. To decide whether a snippet is promising, we compute the upper bound score that can be achieved by the snippets and sort them according to their maximum score. If $min(d)$ represents the minimum number of edges w.r.t. the r structure $\overline{\phi^d} = \phi^{min(d)}$, and $min(|r|, T)$ returns the minimum result cardinality of the snippet r w.r.t. the data source.

Lemma 3. *The upper bound score $\overline{\lambda}(r, T, \mathcal{R}^i)$ of the snippet r is computed as follows:*

$$\overline{\lambda}(r,T,\mathcal{R}^i) = \frac{\overline{\lambda}^{int}(r,T)\times\overline{\lambda}^{nov}(r,\mathcal{R}^i)}{\log_2(1+min(|r|,T))} \tag{9}$$

Example 4. Assume 5 as the maximum size of the snippets. We have the following information: $\overline{IRF} = 10^6$, $\phi = 0.8$, $min(|r|, T) = 1$, and $\log_2(1 + min(|r|, T)) = 1$. Then, we compute the upper bound score of the newly generated snippets $\mathcal{R}^2 = \{r_1, r_2, r_3, r_4\}$ from Fig. 5.b.

(a) $min(d) = 9$, $\overline{\lambda}^{int}(r_1, T) = 78.28 \times 0.13 = 10.18$,
$\overline{\lambda}^{nov}(r_1, \mathcal{R}^2) = 1$, $\overline{\lambda}(r_1,\ T, \mathcal{R}^2) = 10.18 \times 1 = 10.18$,
(b) $min(d) = 9$, $\overline{\lambda}^{int}(r_2, T) = 77.13 \times 0.13 = 10.03$,
$\overline{\lambda}^{nov}(r_2, \mathcal{R}^2) = 1$, $\overline{\lambda}(r_2,\ T, \mathcal{R}^2) = 10.03 \times 1 = 10.03$,
(c) $min(d) = 10$, $\overline{\lambda}^{int}(r_3, T) = 74.96 \times 0.11 = 8.25$,
$\overline{\lambda}^{nov}(r_3, \mathcal{R}^2) = 6.64$, $\overline{\lambda}(r_3,\ T, \mathcal{R}^2) = 8.25 \times 6.64 = 54.78$,
(d) $min(d) = 10$, $\overline{\lambda}^{int}(r_4, T) = 70.85 \times 0.11 = 7.8$,
$\overline{\lambda}^{nov}(r_4, \mathcal{R}^2) = 2.32$, $\overline{\lambda}(r_4,\ T, \mathcal{R}^2) = 7.8 \times 2.32 = 18.1$.

Assume snippets $\mathcal{R}^i = \{r_1, r_2, \ldots, r_l, r_{l+1}, \ldots, r_{|\mathcal{R}^i|}\}$ are sorted according to their upper bound $\overline{\lambda}(r_j, T, \mathcal{R}^i)$, and λ^{min} is the min-score of the l snippets $\mathcal{R}^i_1 = \{r_1, r_2, \ldots, r_l\}$. Then, we stop processing the rest of the snippets $\mathcal{R}^i_2 = \{r_{l+1}, \ldots, r_{|\mathcal{R}^i|}\}$ if $\overline{\lambda}(r_{l+1}, T, \mathcal{R}^i) < \lambda^{min}$ because they cannot beat λ^{min}.

4.1.2. Intra-snippet pruning

The *inter-snippet pruning* expedite the performance by pruning on the snippet level. However, the snippet processing method uses the co-occurring features of a snippet to generate more specific snippets for user interaction. Thus, if we execute pruning on the feature level of the snippets during the snippet processing, we can improve the performance further. To prune the snippet at the feature level, we check the upper bound of the current snippet $\overline{\lambda}(r, T, \mathcal{R}^i)$. Then, if $\overline{\lambda}(r, T, \mathcal{R}^i) < \lambda^{min}$, we can stop processing r because the features are sorted based on IRF score and by adding any snippet, the total score cannot beat λ^{min}. We call this pruning the *intra-snippet pruning*.

4.1.3. The algorithm

Algorithm 2 presents the interactive snippet processing details. We initialize the λ^{min} and set the iteration number i in line 4. In line 5, we compute the upper bound of all current snippets $\overline{\lambda}(r, T), \forall r \in \mathcal{R}$. Then, we sort the snippets $\mathcal{R}$ according to their upper bound in line 6. In line 7, we read snippet r from the snippet list $\mathcal{R}$. If $r.\overline{\lambda} < \lambda^{min}$, we stop processing the snippets as pseudocoded in lines 8–9 (inter-snippet pruning). Next, we retrieve the feature list of snippet r from the index and store it into $\mathcal{F}$ in line 10. In lines 11–12, we read the co-occurring features of r and invoke the procedure $processSnippet$. If the current list is not filled, i.e., $|\mathcal{R}^i| < w$, we add the snippet r to the current Top-w snippet list $\mathcal{R}^i$ as pseudocoded in lines 13–14. In lines 15–16, we apply intra-snippet pruning if $r.\overline{\lambda} < \lambda^{min}$. In lines 19–20, we check the stop condition to terminate the interaction loop. If we do not terminate, we return the Top-w snippets to the user for the next iteration; otherwise, we return the final Top-l snippets $\mathcal{R}^*$ to the user in line 23.

The procedure $processSnippet$ in line 12 of Algorithm 2 is pseudocoded in Algorithm 3. In line 2, we add new feature f to the feature list of r to increase the snippet r size and make it more complete. Then, in lines 3–4, we retrieve the list of root nodes for the features $f_i \in r$, and we use the set intersection operator $\bigcap_{i=1}^{n} V_i$ to update the root node list of the snippet r since adding more features may result in changing the root node. In line 5, we compute the number of edges in the snippet r to compute the interestingness of the newly generated snippet r with respect to the data source because adding feature f results in changing the snippet structure. Next, we compute the interestingness score $r.\lambda^{int}$ and novelty score $r.\lambda^{nov}$ as pseudocoded in lines 6–7. In lines 8–9, we compute the total score of the snippet $r.\lambda$, and upper bound score $r.\overline{\lambda}$. Finally, we return r in line 10.

Algorithm 2: Interactive snippet processing

```
Input  : $Index$, $\mathcal{R}^1$, $w$
Output: Top-$l$ XML Subtree Snippets $\mathcal{R}^*$
1  $i \leftarrow 1$;
2  while true do
3      $\mathcal{R} \leftarrow getSnippet(\mathcal{R}^i)$;          // set the snippets $\mathcal{R}$ using current snippets
4      $\lambda^{min} \leftarrow 0$; $i \leftarrow i+1$;          // initialize $\lambda^{min}$ and set the iteration $i$
5      $r.\overline{\lambda} \leftarrow bound(r,T), \forall r \in \mathcal{R}$;          // compute bound of all current snippets
6      $sortSnippet(\mathcal{R})$;          // sort $\mathcal{R}$ list using upper bound $\overline{\lambda}$
7      while $r \leftarrow getNext(\mathcal{R}) \neq \emptyset$ do
8          if $r.\overline{\lambda} < \lambda^{min}$ then
9              break;          // inter-snippet pruning
10         $\mathcal{F} \leftarrow retrieve(r, Index)$;$\mathcal{R}^i \leftarrow null$;
11         while $f \leftarrow getNext(\mathcal{F}) \neq \emptyset$ do
12             $r \leftarrow processSnippet(r, f, Index, \mathcal{R}^i)$;
13             if $|\mathcal{R}^i| < w$ then
14                 $|\mathcal{R}^i| \leftarrow |\mathcal{R}^i| \cup r$;          // add snippet $r$ to the current list $\mathcal{R}^i$
15             else if $r.\overline{\lambda} < \lambda^{min}$ then
16                 break;          // intra-snippet pruning
17             else if $r.\lambda > \lambda^{min}$ then
18                 $update(\mathcal{R}^i, r)$; $\lambda^{min} \leftarrow \mathcal{R}^i.min$;          // inter-snippet pruning
19     if $Terminate(\mathcal{R}^{i-1}, \mathcal{R}^i)$ then
20         break;          // stop the interaction
21     else
22         continue;
23 return $\mathcal{R}^*$;
```

Algorithm 3: processSnippet

```
Input  : $r$, $f$, $Index$, $\mathcal{R}^i$
Output: XML Subtree Snippet $r$
1  procedure $processSnippet(r, f, Index, \mathcal{R}^i)$
2  $r.\mathcal{F} \leftarrow r.\mathcal{F} \cup f$;          // add new feature $f$ to the feature list $r.\mathcal{F}$
3  $V_i \leftarrow retrieveList(r.f_i, Index), \forall i \in [1-n]$;
4  $r.\mathcal{V} \leftarrow \bigcap_{i=1}^{n} V_i$;          // set intersection operator to update $r$ root node
5  $d \leftarrow countEdge(r)$;          // compute the edges existed in snippet $r$
6  $r.\lambda^{int} \leftarrow \prod_{\forall f \in r.\mathcal{F}} IRF(f,T) \times \phi^d$;          // compute interestingness score of $r$
7  $r.\lambda^{nov} \leftarrow \log_2(1 + \{r.\mathcal{V}\} \setminus \{\mathcal{R}^i.\mathcal{V}\})$;          // set minus operator for novelty
8  $r.\lambda \leftarrow \frac{r.\lambda^{int}(Index) \times r.\lambda^{nov}(\mathcal{R}^i)}{\log(1+|r.\mathcal{V}|)}$;          // compute total score of $r$
9  $r.\overline{\lambda} \leftarrow bound(r, T, \mathcal{R}^i)$;          // compute the upper bound of $r$
10 return $r$;
```

4.2. Finalization of the snippets processing

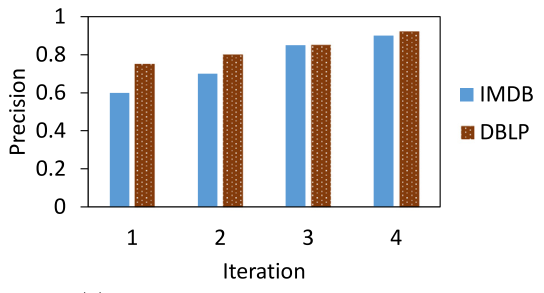
In this step, the user or the system (more preferable) stops the interactive snippet processing and presents the Top-l computed subtree snippets $\mathcal{R}^*$ to the user. The procedure $Terminate$ on line 19 of Algorithm 2 checks whether to stop the interaction automatically. Here, we use the cardinality of the results generated by the Top-w snippets as the condition to continue the interaction. Assume $|\mathcal{R}^i.\mathcal{V}|$ is the cardinality of the results for the current Top-w snippets and $|\mathcal{R}^{i-1}.\mathcal{V}|$ is the cardinality of the results for the previous Top-w snippets. Then, if the condition $|\mathcal{R}^{i-1}.\mathcal{V}| - |\mathcal{R}^i.\mathcal{V}| > \eta$ satisfies, we continue the interaction; otherwise, we stop the interaction and present the current Top-l snippets.

5. Experiments

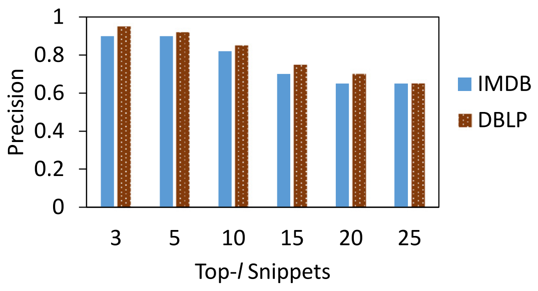
We evaluate the effectiveness and efficiency of our proposed approach. All the experiments are conducted on two real datasets: (a) Internet Movie Database (IMDB) 300 MB, and (b) Digital Bibliography and Library Project (DBLP) 600 MB. All the algorithms are implemented using C# language. The experiments are run on a PC with 3.2 GHz CPU, 8 GB memory and with 64-bit windows 7. Since our approach is interactive, the user would decide in each step to navigate the exploration process based on her favorite snippets.

Table 3
Top-1 primary snippet keywords.

Dataset	Snippet keywords	*Dataset*	Snippet keywords
IMDB	usa, drama	DBLP	web, search
IMDB	action, 2019	DBLP	DKE, 2018
IMDB	gladiator, action, english	DBLP	semantic, search, 2019
IMDB	animation, horror,	DBLP	stonebraker, pvldb
IMDB	usa, horror, killer	DBLP	DKE, exploration, snippet



(a) Iteration effect on the effectiveness



(b) Top-l size on the effectiveness

Fig. 6. Quality of Top-l snippets.

Thus, the effectiveness of our approach is tested by the user in each step of interaction (experiments presented in Section 5.1), and then we focus on the evaluation of the efficiency of the proposed solutions.

5.1. *Effectiveness*

We conducted a comprehensive user study to evaluate the quality of our snippets. To do a reliable user study, we select our users among experts and ordinary users and ask them to score each of our Top-l XML snippets between [0–1] based on their relevance to the original user data exploration criteria (0 means the least relevance and 1 means the most relevance). The users score the snippets in each iteration separately. A part of the sample Top-1 primary snippet keywords (after running A* search on keyword pool) are presented in Table 3.

5.1.1. *Quality of snippets*

Fig. 6 presents the user study evaluation results for IMDB and DBLP datasets. To carry out these experiments we generate a large number of snippets by using a keyword pool (100 keywords) for each data set that are often used by the expert users recently (we run A* search on the keyword pool to produce the Top-ranked primary snippets). Then, we present the Top-5 primary snippets to the users for evaluation. Fig. 6.a presents the quality of the Top-5 snippets. From the figure, we observe the iteration number effect on the system effectiveness. Clearly, as we move forward from iteration 1 to 4, the precision of the Top-5 snippets increases in both datasets. That is because when the exploratory search advances to higher iterations, our method effectively navigates the user's exploration to her favorite results by adding more specific features.

5.1.2. *Effect of Top-l size*

Fig. 6.b presents the effect of Top-l snippet list size on the effectiveness. To carry out the experiments, we set the Top-l size to $\{3, 5, 10, 15, 20, 25\}$ and ask the users to score the Top-l list for each dataset separately. From the figure, we observe that when the list size is set to lower values, e.g. $\{3, 5\}$, the precision of the snippets achieves the highest score. However, when we set the list size to bigger values, e.g. $\{20, 25\}$, the precision of snippets deteriorates. That is because when the list size gets larger, more snippets are added to list that they may not be interesting to the users. However, when the list size is small, most of the snippets are relevant to the user exploration criteria. This fact verifies that our snippet scoring function effectively ranks the more interesting snippets first and then puts the less relevant snippets after the interesting snippets in the Top-l list.

5.2. *Efficiency of computing primary snippets*

Although computing the primary snippets is done in the offline mode, the efficiency of computing the primary snippets is important. That is because we may need to update the trending primary snippets after a specific period of time when the query log changes. To evaluate the efficiency of computing primary snippets, we run experiments by using two methods: (a) Baseline that generates all the combination of keywords up to size of 3 and then executes these queries against the data source, generates the snippets and ranks the Top-w snippets and, (b) AStar that is our proposed method in Section 3.2.

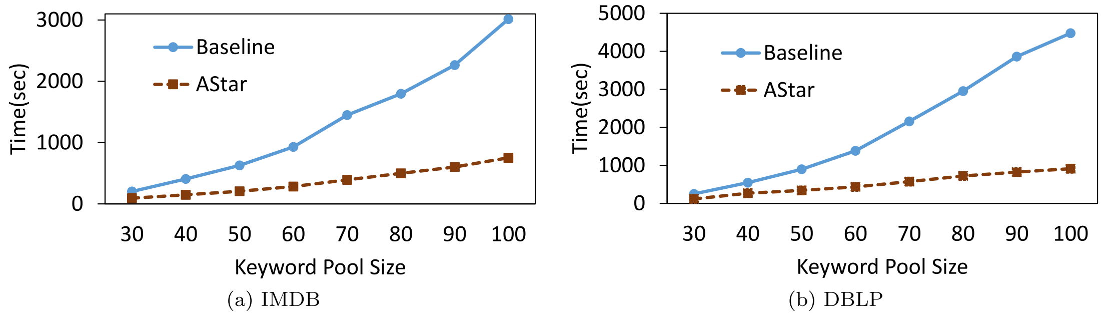


Fig. 7. Keyword pool size $|\mathcal{K}|$ effect on the efficiency of computing primary snippets.

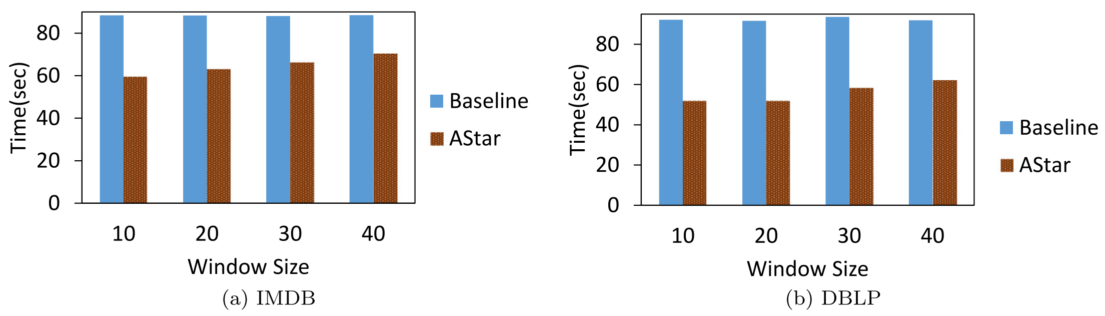


Fig. 8. Window size w effect on the efficiency of computing primary snippets.

5.2.1. *Effect of keyword pool size $|\mathcal{K}|$ on the efficiency*

In this experiment, we vary the size of the keyword pool to $\{30, 40, 50, 60, 70, 80, 90, 100\}$ and compute the execution time of the two methods. Clearly from both datasets in Fig. 7, the AStar method outperforms the Baseline since it does not generate all combinations of $\mathcal{K}$. Moreover, AStar performs an informed search on $\mathcal{K}$ and efficiently stops the computations when the upper bound estimation of the remaining keywords cannot beat the bottom line score of the current snippets. From the figure and in Baseline method, there is a big jump in execution time when $|\mathcal{K}| \geq 60$ which shows that Baseline is not scalable. However, the AStar method shows a small jump in execution time when $|\mathcal{K}|$ increases that means the AStar is scalable.

5.2.2. *Effect of window size w on the efficiency*

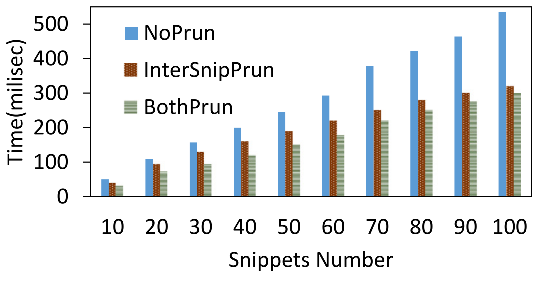
The effect of window size on computing the primary snippets is presented in Fig. 8. To conduct the experiments, we set $|\mathcal{K}| = 20$ and vary the window size w to {10,20,30,40} to compute w snippets for the user by using the Baseline and AStar methods. From the figure, we observe that the baseline method is not sensitive to $|w|$ because it generates all combinations and score them to retrieve Top-w snippets. In the AStar method; however, by increasing the w, there is a small increase in the execution time. That is because in AStar when we increase w, the bottom line score gets smaller; thus, more entries are processed which makes the stop condition to be less effective. However, the AStar method execution time is still better for all window sizes than the Baseline method since it does not generate all the combinations from $\mathcal{K}$.

5.3. *Efficiency of interactive snippet processing*

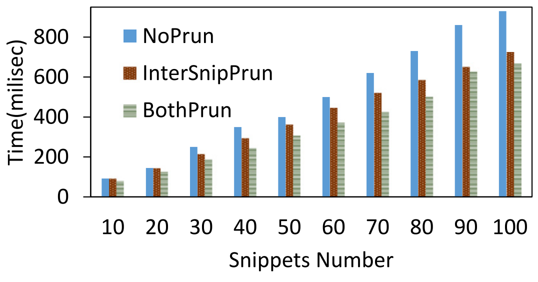
In this section, we evaluate the efficiency of our proposed snippet processing method.

5.3.1. *Pruning improvement*

To show the effectiveness of our pruning techniques, we employ and compare the following three methods: (a) The method that uses index to process the snippets but it does not apply any pruning, (b) the method that uses index to process the snippets and only implements the inter-snippet pruning technique, and (c) the method that uses index to process the snippets and implements both inter and intra-snippet pruning techniques (Algorithm 2). We evaluate the performance of the three methods by assuming the user would choose $|w|$ snippets in one interaction. The results are presented in Fig. 9 for both datasets when we vary the number of snippets to $\{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$. From the figure, we observe that the processing time for the methods that implement the pruning techniques is improved. Inter-snippet pruning is the most effective approach for improving the performance when the number of snippets increases. The performance improves the most when many of the snippets are not processed by using inter-snippet pruning, e.g., in the IMDB when $|w| = \{90, 100\}$. That is because when $|w|$ increases, the possibility that the bottom line score λ^{min} becomes bigger than the upper bound score of the rest of snippets; thus, it makes the inter-snippet pruning technique

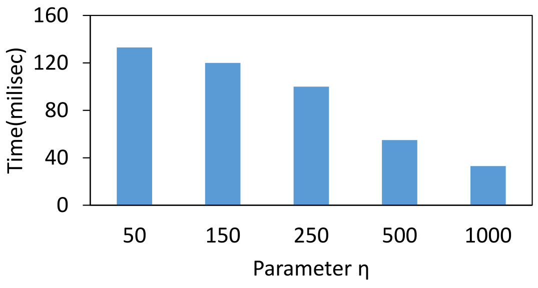


(a) IMDB

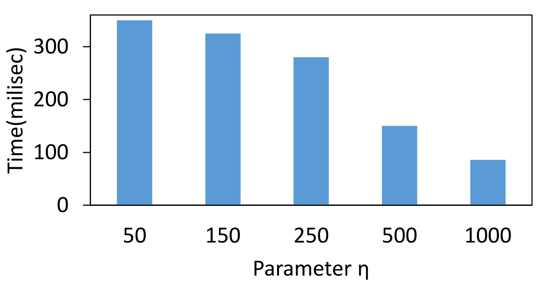


(b) DBLP

Fig. 9. Snippet processing efficiency in one step.



(a) IMDB



(b) DBLP

Fig. 10. The effect of parameter η on the efficiency.

more effective. For intra-snippet pruning technique; however, the performance is better when there are a big number of co-occurring features that should be processed, and some of these features are skipped by the intra-snippet pruning because adding them to the snippet cannot beat the bottom line λ^{min}. In conclusion, both pruning techniques are effective; however, the inter-snippet pruning is the most effective technique when the number of snippets is big, and then the intra-snippet pruning is effective when the number of skipped co-occurring features is huge.

5.3.2. *Effect of parameter η on the efficiency*

We set the parameter η to the values $\{50, 150, 250, 500, 1000\}$ and compute the execution time for the interactive snippet processing when the window size $|w| = 10$. During the interaction, the system chooses 5 snippets randomly to continue the snippet processing. The results of this experiment are presented in Fig. 10. From the figure, we observe that when η is set to a smaller value, the execution time increases, i.e., $\eta = \{50, 150\}$. However, when we set η to a bigger value, the execution time decreases. That is because when we set η to a bigger value, there is smaller possibility that the previous iteration's cardinality is bigger than the big η; therefore, the system stops the interaction early, i.e., in the first or second iterations. Conversely, when we set η to a smaller value, the stop conditions would satisfy after more iterations; therefore, it increases the execution time. To sum up, for the users who are interested in interacting with the system in various iterations, η could be set to a smaller value. However, for the users who would like to get the exploration results with the minimum interaction, η could be set to a bigger value.

5.3.3. *Naive method comparison with our approach*

A naive way to process the snippets is that we retrieve the co-occurring features at each step. Then for each co-occurring feature, we add a feature to the snippet and compute the root nodes of the newly generated snippets in the data source without using the index. The result of snippet processing using this method is presented in Fig. 11 for both datasets. Clearly, in both datasets, the execution time shows a big jump when the number of snippets increases which indicates that the naive approach is not scalable. Moreover, the execution time is not comparable to the method that uses indexing structure to store the primary snippet computations. This indicates that the naive method is impractical due to its heavy computational costs and we have to use the index to reuse the computations of the primary snippets in the interactive phase.

6. Related work

In this section, We present the various related works to data exploration and review the literature.

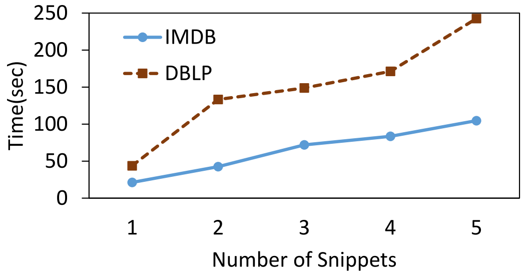


Fig. 11. Naive method without using index.

6.1. *Exploratory search*

Users often tend to perform an exploratory search on the data stored in databases [13–20]. However, traditional systems are not equipped with interactive tools to support this kind of exploration tasks [4]. OLAP cubes enable the users to explore the data by providing them a relational representation of aggregate data [14,17]. Giacometti et al. [18] proposed OLAP cubes query recommendation by exploring the items that are visited by other users. Kalinin, Cetintemel and Zdonik [3] employed semantic windows for the users to set their favorite conditions and perform exploratory queries. They also provide the users with the online and partial result option to improve the interactivity of the system. Qarabaqi and Riedewald [2] proposed exploratory search using probability distribution over the entities attributes. Therefore, the users are helped to specify right conditions on imprecise queries. However, these works focus on structured data like relational databases. Moreover, they propose their methods with special kind of queries like range queries. Conversely, we target the exploration of semi-structured data like XML. We consider the scenario when the user is clueless, and we assist her in the exploration process by generating primary snippets from the trending keywords of the query log. Ouksili et al. [21] exploits patterns to facilitate the exploration over RDF graphs. Unlike this work, XSnippets focuses on interactive exploratory search for clueless users over general XML data.

6.2. *Recommendation systems*

Recommendation systems explore the data source for the users who have visited the data before to retrieve similar items for them or finding similar content that has been looked up by the similar users [22–25]. Akbarnejad et al. [26] proposed query recommendation based on a prediction of the items that user is interested in those items. Yao et al. [27] proposed to exploit structural semantics for query reformulation. Meng, Cao, and Shao [28] used the semantic relationships between keywords and keyword queries to suggest a set of keyword queries from the query log. The semantic relationship between keywords and keyword queries is measured as the combination of the internal and external co-occurrence of the keywords. However, there is no interaction with the user for query recommendation. Drosou and Pitoura [5] presented a database exploration framework which recommends additional items called ‘You May Also Like’ results. These results are not a part of the original user query results; however, they are highly correlated to the original results. These similar items are retrieved based on the most interesting sets of attribute-values, called faSets, that is extracted from the content of the user query result. However, the exploration process is based on the results of the user given query. In contrast, our work incorporates the users feedback in the data exploration in several iterations to navigate the exploration. Furthermore, most of these works focus on the exploration of the relational databases while our work presents a framework for interactive data exploration for the clueless user on the XML data. Naseriparsa et al. [29] proposed a comprehensive framework for XML keyword search to retrieve semantically related results when the user original query is unable to produce sufficient results, but semantically related content exists. Conversely, this work focuses on assisting the clueless users through exploration process by presenting some exciting hints. Some works such as [30,31] proposed recommendation of keyword query search results for the user over XML data. They focus on result diversification of keyword search results and exploiting the underlying XML data statistics to retrieve meaningful results for the user. In [32], a recommendation system called XPloreRank is proposed that uses two correlation scores to generate “You May Also Like” keyword queries for the users over XML data. However, XSnippets is the first interactive exploratory search framework that navigates a clueless user through her exploration over XML data by using XML snippets.

6.3. *XML keyword search*

XML keyword search is studied extensively in the literature [33]. The XML search results are considered as the fragments of XML document that satisfy the search conditions. Guo et al. [6] proposed the Lowest Common Ancestor (LCA) to extract the XML nodes in the tree that contain all query keywords in the same subtree. Xu and Papakonstantinou [7] introduced the concept of Smallest Lowest Common Ancestor (SLCA) to tighten the extracted nodes for the query result to the smallest tree that contains all the keywords in the same subtree. Zhou et al. [34] exploit the set intersection problem for improving the XML keyword search processing by using a modified inverted list, namely IDList. Other LCA-based search semantics include ELCA [9,35], MLCA [36] and VLCA [37].

Since the XML keyword search may retrieve numerous results, some works proposed ranking schemes to retrieve only highly relevant results [6,38–40]. Hristidis, Papakonstantinou, and Balmin [38] only considered the proximity of the keywords within the XML documents to rank the results. Guo et al. [6] modified the popular Google's PageRank measure to rank the XML elements according to (a) the structure of the XML document and (b) the proximity of the keywords in the XML document. Termehchy and Winslett [39,40] utilized the mutual information measures to design coherency ranking scheme for the XML search results. Barros et al. [41] proposed two algorithms for processing multiple keyword queries over XML streams. Le, Bao, and Ling [42] proposed CR semantics (Common Relative) for XML keyword search which returns answers independent from schema designs. However, in all these works, the keyword search methods do not include an exploratory option to assist the user during the search process. To improve the usability of the search [43], we provide an interactive environment to navigate a user who is unfamiliar with the content and structure of data. We exploit the XML snippet which summarizes the result of keyword queries to present the user an overall picture of the data and then interactively navigate her to the favorite part of data in several iterations.

7. Conclusion

We propose a framework for interactive exploratory search on semi-structured data. We employ the trending keywords of the query log to generate primary snippets that present an overall picture of the data to the clueless user. Since the number of snippets is enormous, we propose an A*-search based technique to avoid generating the non-promising snippets. To reuse the primary snippet computations, we design an index structure to store the snippet computations. Then we exploit the co-occurring content of the snippets to navigate the exploration in an interactive environment with user feedback about her favorite snippets. In each iteration of the interaction, the snippets are ranked based on their interestingness, novelty and their cardinality. To avoid the unnecessary computations in the interactive phase, we design two novel pruning techniques and a stop condition to check whether to stop the interactive phase automatically. The extensive experiments demonstrate the effectiveness and efficiency of our approach.

Acknowledgments

This work is supported by the Australian Research Council discovery grants DP140103499 and DP160102412.

References

[1] Z. Liu, Y. Huang, Y. Chen, Improving XML search by generating and utilizing informative result snippets, ACM Trans. Database Syst. 35 (3) (2010) 19:1–19:45.
[2] B. Qarabaqi, M. Riedewald, Merlin: Exploratory analysis with imprecise queries, IEEE Trans. Knowl. Data Eng. 28 (2) (2016) 342–355.
[3] A. Kalinin, U. Çetintemel, S.B. Zdonik, Interactive data exploration using semantic windows, in: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, 2014, pp. 505–516.
[4] A. Kashyap, V. Hristidis, M. Petropoulos, Facetor: cost-driven exploration of faceted query results, in: Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010, 2010, pp. 719–728.
[5] M. Drosou, E. Pitoura, Ymaldb: exploring relational databases via result-driven recommendations, VLDB J. 22 (6) (2013) 849–874.
[6] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram, XRANK: ranked keyword search over XML documents, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003, 2003, pp. 16–27.
[7] Y. Xu, Y. Papakonstantinou, Efficient keyword search for smallest lcas in XML databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005, 2005, pp. 537–538.
[8] C. Sun, C.Y. Chan, A.K. Goenka, Multiway slca-based keyword search in XML data, in: Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007, 2007, pp. 1043–1052.
[9] Y. Xu, Y. Papakonstantinou, Efficient LCA based keyword search in XML data, in: EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings, 2008, pp. 535–546.
[10] Z. Bao, T.W. Ling, B. Chen, J. Lu, Effective XML keyword search with relevance oriented ranking, in: Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China, 2009, pp. 517–528.
[11] Z. Bao, Y. Zeng, T.W. Ling, D. Zhang, G. Li, H.V. Jagadish, A general framework to resolve the mismatch problem in XML keyword search, VLDB J. 24 (4) (2015) 493–518.
[12] M. Hadjieleftheriou, A. Chandel, N. Koudas, D. Srivastava, Fast indexes and algorithms for set similarity selection queries, in: Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México, 2008, pp. 267–276.
[13] G. Chatzopoulou, M. Eirinaki, N. Polyzotis, Query recommendations for interactive database exploration, in: Scientific and Statistical Database Management, 21st International Conference, SSDBM 2009, New Orleans, la, USA, June 2-4, 2009, Proceedings, 2009, pp. 3–18.
[14] J. Gray, A. Bosworth, A. Layman, H. Pirahesh, Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total, in: Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, 1996, pp. 152–159.
[15] P.J. Haas, J.M. Hellerstein, Ripple joins for online aggregation, in: SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA, 1999, pp. 287–298.
[16] J.M. Hellerstein, P.J. Haas, H.J. Wang, Online aggregation, in: SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA, 1997, pp. 171–182.
[17] S. Sarawagi, R. Agrawal, N. Megiddo, Discovery-driven exploration of OLAP data cubes, in: Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings, 1998, pp. 168–182.
[18] A. Giacometti, P. Marcel, E. Negre, A. Soulet, Query recommendations for OLAP discovery-driven analysis, IJDWM 7 (2) (2011) 1–25.
[19] A. Freitas, J.G. Oliveira, S. O'Riain, J.C.P. da Silva, E. Curry, Querying linked data graphs using semantic relatedness: A vocabulary independent approach, Data Knowl. Eng. 88 (2013) 126–141.
[20] J. Sun, J. Xu, K. Zheng, C. Liu, Interactive spatial keyword querying with semantics, in: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017, 2017, pp. 1727–1736.
[21] H. Ouksili, Z. Kedad, S. Lopes, S. Nugier, Pattern oriented RDF graphs exploration, Data Knowl. Eng. 113 (2018) 171–183.
[22] R.J. Mooney, L. Roy, Content-based book recommending using learning for text categorization, in: ACM DL, 2000, pp. 195–204.
[23] M.J. Pazzani, D. Billsus, Learning and revising user profiles: The identification of interesting web sites, Mach. Learn. 27 (3) (1997) 313–331.

[24] G. Adomavicius, A. Tuzhilin, Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions, IEEE Trans. Knowl. Data Eng. 17 (6) (2005) 734–749.
[25] C. Palmisano, A. Tuzhilin, M. Gorgoglione, Using context to improve predictive modeling of customers in personalization applications, IEEE Trans. Knowl. Data Eng. 20 (11) (2008) 1535–1549.
[26] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, J.S.V. Varman, SQL querie recommendations, PVLDB 3 (2) (2010) 1597–1600.
[27] J. Yao, B. Cui, L. Hua, Y. Huang, Keyword query reformulation on structured data, in: IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012, 2012, pp. 953–964.
[28] X. Meng, L. Cao, J. Shao, Semantic approximate keyword query based on keyword and query coupling relationship analysis, in: Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014, 2014, pp. 529–538.
[29] M. Naseriparsa, M.S. Islam, C. Liu, I. Moser, No-but-semantic-match: computing semantically matched xml keyword search results, World Wide Web 21 (5) (2018) 1223–1257.
[30] J. Li, C. Liu, J.X. Yu, Context-based diversification for keyword queries over XML data, IEEE Trans. Knowl. Data Eng. 27 (3) (2015) 660–672.
[31] J. Li, C. Liu, R. Zhou, W. Wang, XML keyword search with promising result type recommendations, World Wide Web 17 (1) (2014) 127–159.
[32] M. Naseriparsa, C. Liu, M.S. Islam, R. Zhou, Xplorerank: exploring xml data via you may also like queries, World Wide Web 22 (4) (2019) 1727–1750.
[33] Z. Liu, Y. Chen, Processing keyword search on XML: a survey, World Wide Web 14 (5–6) (2011) 671–707.
[34] J. Zhou, Z. Bao, W. Wang, J. Zhao, X. Meng, Efficient query processing for XML keyword queries based on the idlist index, VLDB J. 23 (1) (2014) 25–50.
[35] R. Zhou, C. Liu, J. Li, Fast ELCA computation for keyword queries on XML data, in: EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings, 2010, pp. 549–560.
[36] Y. Li, C. Yu, H.V. Jagadish, Schema-free xquery, in: (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004, 2004, pp. 72–83.
[37] G. Li, J. Feng, J. Wang, L. Zhou, Effective keyword search for valuable lcas over xml documents, in: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007, 2007, pp. 31–40.
[38] V. Hristidis, Y. Papakonstantinou, A. Balmin, Keyword proximity search on XML graphs, in: Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India, 2003, pp. 367–378.
[39] A. Termehchy, M. Winslett, Effective, design-independent XML keyword search, in: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009, 2009, pp. 107–116.
[40] A. Termehchy, M. Winslett, Using structural information in XML keyword search effectively, ACM Trans. Database Syst. 36 (1) (2011) 4:1–4:39.
[41] E.G. Barros, A.H.F. Laender, M.M. Moro, A.S. da Silva, Lca-based algorithms for efficiently processing multiple keyword queries over XML streams, Data Knowl. Eng. 103 (2016) 1–18.
[42] T.N. Le, Z. Bao, T.W. Ling, Exploiting semantics for XML keyword search, Data Knowl. Eng. 99 (2015) 105–125.
[43] F. Li, H.V. Jagadish, Usability, databases, and HCI, IEEE Data Eng. Bull. 35 (3) (2012) 37–45.

Dr. Mehdi Naseriparsa, received his BSc and MS degree in Computer Engineering from IAU, Iran, in 2007 and 2013, respectively. He finished his PhD in 2018 at the Swinburne University of Technology, Australia. He is currently a sessional academic at Swinburne University of Technology. His current research interests are in the areas of big data, data analytics, machine learning, and medical informatics.

Dr. Md. Saiful Islam is a Lecturer in the School of Information and Communication Technology, Griffith University, Australia. He has finished his PhD in Computer Science and Software Engineering from Swinburne University of Technology, Australia in February, 2014. He has received his BSc (Hons) and MS degree in Computer Science and Engineering from University of Dhaka, Bangladesh, in 2005 and 2007, respectively. His current research interests are in the areas of database usability, spatial data management and big data analytics.

Prof. Chengfei Liu received the BS, MS and PhD degrees in Computer Science from Nanjing University, China in 1983, 1985 and 1988, respectively. Currently, he is a Professor in Swinburne University of Technology, Australia. His research interests include keywords search on structured data, query processing and refinement for advanced database applications, query processing on uncertain data and big data, and data-centric workflows. He is a member of IEEE and ACM.

Dr. Lu Chen finished his PhD in 2018 at the Swinburne University of Technology, Australia. He is currently a postdoctoral research fellow at Swinburne University of Technology. His current research interests are in the areas of data management, information retrieval, and data analytics.