Query Results Ready, Now What?

Ziyang Liu Yi Chen
Arizona State University
{ziyang.liu, yi}@asu.edu

Abstract

A major hardness of processing searches issued in the form of keywords on structured data is the ambiguity problem. A set of keywords itself is not a complete piece of information and may imply different information needs from different users. Although state-of-the-art keyword search engines are usually able to automatically identify meaningful connections among keywords in the data, users may still be frequently overwhelmed by the huge number of results and face much trouble in selecting the relevant ones. Many search engines attempt to rank the results in order of their inferred relevance, however, it is virtually impossible for a ranking scheme to be perfect and works properly for all users and all queries.

In this paper we discuss several post-processing methods for keyword searches on structured data, which ease the users' task of finding and digesting relevant results from all results returned for a keyword query. The methods to be discussed include generating result snippets, differentiating selected results as well as query expansion. Each of these methods helps users gracefully get the relevant information in its own way. At last we discuss the remaining challenges in post-processing keyword searches on structured data.

1 Introduction

Keyword searches are usually ambiguous. A set of keywords with no further information of how they should be related/connected in the results, as is the typical scenario in most keyword searches, can have many different ways to be interpreted. It may very well be the case that different users with different intensions issue the same set of keywords. As a result, it is impossible for a search engine which simply provides a set of ranked results to guarantee that the order of the results exactly matches the user's need. This may cause the users to spend lots of time navigating through the results in order to find the relevant ones. Moreover, it is unlikely that the user always comes up with a good set of keywords; often times the user may miss important keywords or use some "bad" keywords which may harm the quality of the retrieved results, and it will be desirable if the search engine can provide help to the user if so happen.

In this paper we discuss several post-processing techniques that work beyond generating a ranked list of results for keyword searches on structured data. Each of these techniques helps the user get insights of the results in its own way. Let us look at a few desirable ways of postprocessing.

Copyright 2010 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

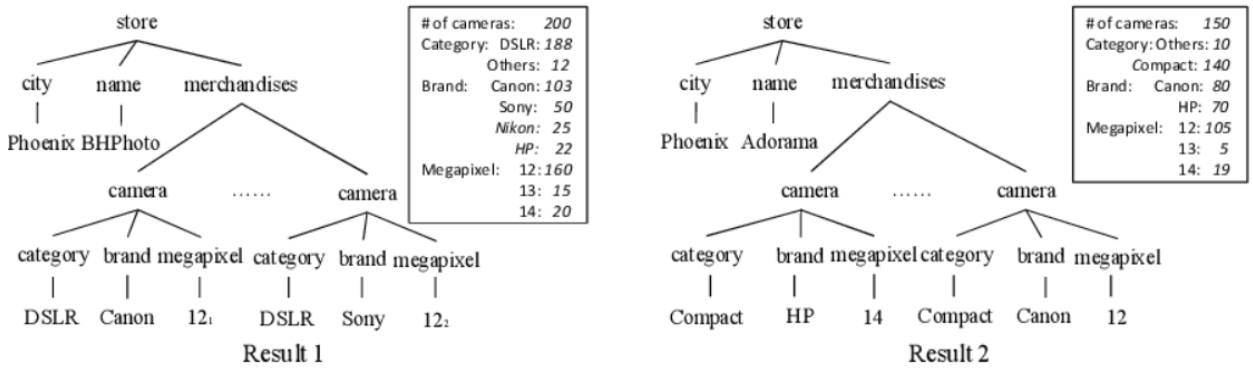

Figure 1: Two Results of Query "*Phoenix, camera, store*"

Result snippet generation is one of the most helpful postprocessing methods for keyword search and is used by almost all web search engines. Although search engines attempt to rank the results in terms of their relevance to the query, all existing ranking schemes are heuristics; the user usually has no idea based on what criteria the results are ranked and cannot possibly fully trust the ranking. Indeed, when we issue a web search, very often the top results consist of a mixture of relevant and irrelevant results. Result snippets are very helpful in these circumstances as they help the users quickly judge the relevance of each result.

After the user roughly understands the content of each result, the user may be interested in the relationship of a set of results by comparing them. Such needs are common in applications like online shopping, employee hiring, job/institution hunting, etc. It is shown in [2] that 50% of keyword searches on the web are for information exploration purposes, and inherently have multiple relevant results. Such queries are classified as informational queries, where a user would like to investigate, evaluate, compare, and synthesize multiple relevant results for information discovery and decision making, rather than reaching a particular website. Generating a comparison table which highlights the key differences of the selected results will be very helpful.

Furthermore, although result snippets and differentiation tables are helpful for users to judge the relevance of results, yet if the keywords selected by the user is imperfect to begin with, the query may retrieve so many irrelevant results that it is still very difficult for the user to pick the relevant ones even with the help of snippets and differentiation techniques. By suggesting a set of related queries to the keyword query the user has issued (also called query expansion), the search engine can help the user get more insights of what is the important information contained in the results, and narrow down the search scope to get more precise results.

Sometimes the keywords have multiple relationships in the data, and in this case clustering the search results and showing one representative for each cluster makes it easy for the user to quickly browse all types of results. Last but not the least, the search engine can also work interactively with the user to improve the result quality through relevance feedbacks.

The rest of the paper is organized as follows. Section 2 discusses result snippet generation techniques proposed in eXtract [4]; Section 3 introduces the approach for generating comparison tables for selected results proposed in XRed [7]; in Section 4 we introduce two related approaches, Data Clouds [5] and [10], which selects important keywords from the search results for query expansion. At last we discuss the remaining challenges of processing keyword searches on structured data and potential future research directions.

2 Generating Result Snippets

Snippets help the user quickly understand the essence of a result without reading through its entirety. As an example, let us look at fragments of two tree-structured search results of query "*Phoenix, camera, store*" shown

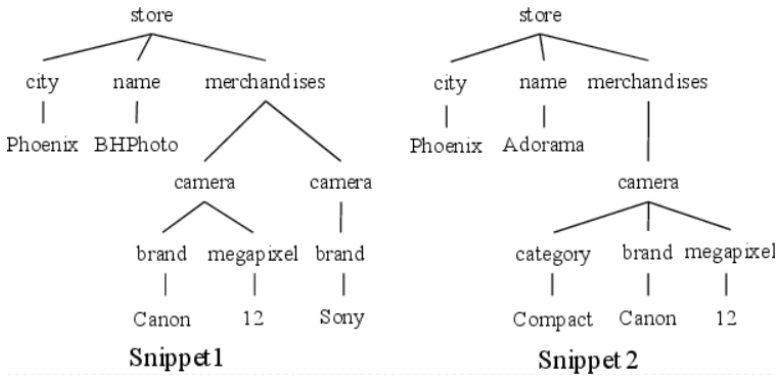

Figure 2: Snippets of the Result in Figure 1

in Figure 1. Some nodes in the results are omitted due to space constraint, and the box next to each result records the statistics information of the result, e.g., *DSLR: 188* indicates there are 188 DSLR cameras in this result.

Two sample snippets of the results are shown in Figure 2. As can be seen, each snippet captures the heart of the result in a small tree, e.g., from snippet 1, we know that the result is about a store named BHPhoto in Phoenix, which features Canon and Sony cameras and mainly sells cameras with 12 megapixel. Users can judge the relevance of the query result from this brief subtree of carefully selected nodes.

Snippet generation on structured data faces several unique challenges comparing with snippet generation on plain text documents. Most importantly, unlike results on text documents, results on structured data typically have a tree structure, which is not composed of sentences or paragraphs. Therefore, selecting representative sentences as snippets is not a valid solution. Instead, the selection should be made at the node level. Except nodes that match keywords, it is not directly obvious which nodes are significant and helpful for the user to judge the result relevance. In this subsection we introduce a system eXtract [4], which adopts techniques for snippet generation in XML keyword search.

eXtract attempts to achieve four goals in generating a snippet: (1) the snippet should be self-contained (i.e., represent a semantic unit); (2) the snippet should distinguish the result from other results; (3) the snippet should be a reasonable summary of the result; (4) the snippet should be small to be easily comprehensible. Note that the first three goals interfere with the last one: the more nodes a snippet has, the better it achieves the first three goals but the harder it is for the user to comprehend it. eXtract takes all four goals into account by creating a list of important *features* in the result, called IList. eXtract then adds the features in the IList into the snippet one by one, until the snippet size has reached a limit. Each feature is a triplet consisting of an entity, an attribute and a value, e.g., (*camera, category, DSLR*). Entities and attributes are inferred using the heuristics proposed in [6].

The IList is initialized to include the keywords in the query. To make the snippet self-contained, eXtract adds all entities in the result into IList. Take result 1 in Figure 1 as an example. The entities in the result are store and camera.

To make the snippet distinguishable, eXtract identifies the key of a result and adds it to the IList. The key of a result is considered to be the key attribute of an entity (called return entity) selected by eXtract. In result 1 in Figure 1, the return entity is *store*. The key attribute can be retrieved from the DTD or schema (if available), otherwise the most selective attribute can be used. eXtract adds the key attribute value of the return entity to the IList.

To make the snippet a reasonable summary of the result, eXtract identifies features whose occurrences are dominant. A feature is dominant if its number of occurrences in the result is large compared with features with the same type, where feature type refers to the (entity, attribute) pair. For example, according to the statistics of result 1 in Figure 1, *DSLR*, *Canon*, *Sony* and *12* are dominant features. A quantitative measure is adopted to assess the dominance of a feature, i.e., the dominance score of feature f wrt result r, defined as following:

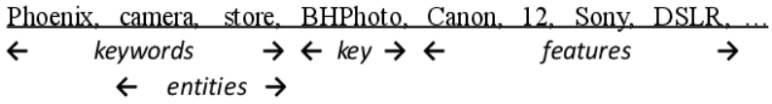

Figure 3: IList for Result 1 in Figure 1

Feature Type	DFS of Result 1 (D_1)	DFS of Result 2 (D_2)
store:name	BHPhoto	Adorama
camera:brand	Canon Canon Sony	Canon HP
camera:category	DSLR	Compact

Figure 4: Comparison Table of the Result in Figure 1

$$DS(f,r) = \frac{N(f)}{\frac{N(f.t)}{D(f.t)}}$$

where $N(\cdot)$ is the number of occurrences of a feature or feature type in the result, $f.t$ is the feature type of f, $D(\cdot)$ is the domain size (i.e., number of distinct features) of a feature type. For example, suppose there are 3 camera categories (DSLR, compact, single-use), then the dominance score of feature *DSLR* is 188/(200/3) = 2.82. eXtract adds the features in the result into the IList in the order of their dominance score. The final IList of result 1 in Figure 1 looks like the one in Figure 3.

eXtract then includes the items in the IList one by one into the snippet until it reaches a preset size limit in terms of number of edges. The challenge hereby is to choose the instance of each item so that the snippet can accommodate as many items as possible. For example, to include items *canon* and *12*, using 12_1 in Figure 1 is better than using 12_2 as the former choice results in a smaller tree size, thus leaving more space for other items. The problem of including the most number of items in the snippet is proved to be NP-hard, and eXtract uses a greedy algorithm for snippet generation. The details of the algorithm are omitted in this paper and can be learned from [4].

3 Query Result Differentiation

From reading the snippets, the users will understand the most important features in the results. However, many snippets may look similar to each other, and in order for the user to further judge the relevance of results, he/she needs to learn the key differences of these results by comparing and analyzing them. From Figure 1, the store in result 1 mainly sells Canon and Sony DSLR cameras, with roughly twice as many Canon cameras as Sony cameras; while the store in result 2 mainly sells Canon and HP Compact cameras. From their snippets, we know result 2 focuses on Compact cameras, but have no idea whether or not result 1 focuses on Compact or DSLR, since the category information about the store is missing in its snippet. Similarly, result 1 has many Sony cameras, but we do not have information about whether result 2 has many Sony cameras or not. As we can see, snippets are designed to summarize a single result, rather than compare and differentiate multiple results. Figure 4 shows the comparison table of the two results, which highlights the key difference of the two results, e.g., the first store focuses on DSLR and the second one focuses on compact camera, which is helpful for the user to make his/her decision.

In this section we introduce XRed [7], a structured search result differentiation system that generates a *Differentiation Feature Set* (DFS) for each result to help users compare and contrast results. Similar as eXtract, XRed considers each (entity, attribute, value) triplet as a feature in the result. XRed allows users to select a set of results for comparison, then it automatically selects a set of features from each result to generate a DFS that shows the differences of the result from others.

XRed generates DFS according to the following 3 desiderata:

(1) *Being Small.* Similar as a result snippet, to enable users quickly differentiate query results, a DFS should be small, so that users can quickly browse and understand them. XRed allows the user to specify a number which is used as the upper bound of the number of features in each DFS.

(2) *Summarizing Query Results.* For the comparisons based on DFSs to be valid, a DFS should be a reasonable summary of the corresponding result by capturing the main characteristics in the result. Otherwise, the differences shown in two DFSs do not reflect the actual differences between the corresponding query results.

Specifically, a feature that has more occurrences in the result should have a higher priority to be selected in the DFS, so that the DFS reflects the most important feature in the result, and the differences among DFSs correctly reflect the main differences of their corresponding results. Consider again the two results of query "*Phoenix, camera, store*" in Figure 1(a). Both results mainly sell *Canon* cameras. The store in result 1 also sells a couple of *HP* cameras. Suppose we have the DFS for result 1, D_1={store:brand:HP}, and the DFS for result 2, D_2={store:brand:Canon}. Obviously these two DFSs are different. However, the difference is meaningless as they give users the impression that store 1 differs from store 2 by mainly selling HP cameras instead of Canon cameras, which is untrue.

Furthermore, the distributions of features of the same type in a DFS should roughly reflect their distribution in the data (up to the size limit). For example, although both stores in these results sells *Canon* and *HP*, it is undesirable to have a single occurrence of *Canon* and *HP* in the DFS of each result. Such DFSs give users the impression that the two stores are similar in terms of their speciality on *Canon* and *HP*. In fact, the store in result 1 mainly focuses on *Canon* with just a couple of *HP*; whereas the store in result 2 focuses on both *Canon* and *HP*, with roughly the same number of cameras.

(3) *Differentiating Query Results.* Obviously, a DFS should be able to differentiate the result it represents from others. Since different results are compared on *feature types*, a differentiation unit is a feature type, e.g., (*camera, category*). A feature type t can differentiate two DFSs if the features or the order of features of type t are different in the two DFSs.

As an example, consider feature type (*camera, category*) in the DFSs in Figure 4. This feature type can differentiate the two DFSs, as feature *DSLR* is the most common feature of this type in DFS 1, while in DFS 2 the most common one is *compact*. For feature type (*camera, brand*), it can also differentiate the two DFSs as the second most common feature in the two DFSs are *Sony* and *HP*, respectively. Assuming that we use feature *HP* to replace *Sony* in DFS 1, then they are still differentiable, as the ratios between the number of occurrences of *Canon* and that of *HP* are about 2:1 in DFS 1 and 1:1 in DFS 2, respectively.

Using the concept of differentiable feature type, XRed judges the quality of two DFSs by the number of feature types that can differentiate, which is called their *degree of differentiation* (DoD). The DoD of multiple DFSs is the total DoD of all pairs of DFSs.

The DFS generation problem is formulated as the problem of generating a set of DFSs, one for each result, such that each DFS should satisfy the size limit, be valid and maximize their DoD. The problem is proved to be NP-hard. XRed adopts greedy algorithms that achieve two local optimality criteria efficiently. The detailed algorithms and proofs can be found in [7].

4 Query Expansion

Sometimes since the keywords issued by the user are imperfect, a large number of irrelevant results may be retrieved. Even with the help of snippets and differentiation, it is still a tough task for the user to quickly locate the relevant ones. In this case, the search engine can help user narrow down the search scope and improve the result quality by suggesting a set of keywords related to the original query, which the user can use to revise the query. For example, the sample query "*Phoenix, camera, store*" may retrieve a large number of results as the predicates are not very specific. The user issues this query possibly because he/she is just interested in buying a digital camera, but is unfamiliar with detailed specifications of cameras. If the search engine is able to automatically recommend to the user a set of keywords such as "DSLR", "compact", "12 megapixel", etc., the user will be able to pick the desired keywords and reduce the number of retrieved results.

Major search engines provide query expansion functionality in the form of query auto-completion. Most of them are implemented based on user query logs [1]. For instance, [3] extracts the personal information from users' desktops and query logs, and provides adaptive factors to evaluate the relevance of words. Although these techniques are designed for web search engines, they can be applied to keyword search on structured data as well. We do not further elaborate these approaches as they are not based on post-processing the query results.

In this section we mainly focus on a paper that studies query expansion for keyword search on structured data without the aid of query logs: Data Clouds [5]. Data Clouds allows user to issue keyword queries on relational databases, and ranks the terms in all results or selected top results, then returns the top ranked terms, which can be used to form expanded queries.

To generate query results, Data Clouds first identifies a set of entities (e.g., product) in the relational database which is done by domain experts or database administrators. Then it asks the user to specify which entity to search. Each result is an entity that contains all keywords in the tuples related to it.

To find interesting terms as suggest keywords, Data Clouds discusses three alternative ranking schemes for ranking terms in the results: popularity based ranking scheme, relevance based ranking scheme, and query-dependent ranking scheme. Data Clouds returns terms with high scores for query expansion. The ranking schemes are illustrated below.

(1) *Popularity Based Ranking Scheme.* This ranking schemes is based on the observation that the keywords that appears many times in a result are likely important.. Given query q, a result r and a term $t \in r$, the popularity score of term t is:

$$score(t, q, r) = \sum_{e \in r} \sum_{a \text{ of } e} n_a$$

where e is an entity, a is an attribute of e and n_a is the number of times t occurs in a.

(2) *Relevance Based Ranking Scheme.* Simply considering the numbers of occurrences of terms may lead to selecting terms that occur universally frequently, which is undesirable as those terms are not representative for the results. The relevance based ranking scheme additionally incorporates the idea of inverse document frequency into the ranking score. Given query q, a result r and a term $t \in r$, the relevance score of term t is:

$$score(t, q, r) = \sum_{e \in r} tf_{t,e} \times idf_t$$

where $tf_{t,e}$ is the frequency of term t within entity e, and idf_t is the log of the ratio of the number of entities in the data over the number of entities containing t. Note that relevance based ranking scheme uses term frequency $tf_{t,e}$ instead of number of occurrences in the popularity based ranking scheme, as term frequency is normalized over the size of the entities and is more fair.

(3) *Query Dependent Ranking Scheme.* Compared with the relevance based ranking scheme, this ranking scheme additionally favors keywords occurring in entities that are highly related to the query. By combining TFIDF with the scores of entities, the query dependent score of a term t is:

$$score(t, q, r) = \sum_{e \in r} (tf_{t,e} \times idf_t) \times score(e, q)$$

where $score(e, q)$ is the relevance of entity e to query q, computed using traditional TFIDF metrics.

[8] also proposes a set of ranking factors for finding important keywords related to a user query. It is designed specifically for queries related to online shopping: the keywords in the expanded queries are features of the products. [8] uses three factors to judge the importance of a candidate query. The first factor is based on the co-occurrence of the words in the candidate query with the keywords in the original query; high co-occurrence is preferred. The other two factors rely on the users' reviews of a product. They favor the attributes of a product with extreme ratings (i.e., highly positive/negative) and consistent ratings (i.e., receiving the same rating from many users), respectively. Top-k queries with the highest scores are computed and returned to the user. Another difference from Data Clouds is that [8] considers a ranked list of queries, rather than keywords. For example, for a user query "Canon", Data Clouds returns suggested keywords such as "lens", "DSLR", "zoom", etc., while [8] returns suggested queries such as "camera, lens, zoom".

Different from the two approaches introduced above, [10] ranks the terms in the order of their number of occurrences in the results. It requires the input data to be a relational database with a schema. [10] proposes an efficient algorithm that is able to compute precisely the top-k frequently occurred terms without even processing the query. This enables the query processing and query expansion to be parallelized and shorten the response time of a search engine. The detailed algorithm is omitted due to space constraint and we direct readers to the paper for more insights.

5 Conclusions and Future Work

In this paper we discuss the benefits of keyword query post-processing on structured data, and introduce several post-processing techniques which, from different angles, help users retrieve the relevant results: result snippets, result differentiation and query expansion. Result snippets help user quickly get the essence of the results without the need to read the entire results; result differentiation highlights the key differences of the selected results which help the user compare and prioritize them; query expansion suggests a set of keywords based on the results which gives the user insights of the results and helps the user narrow down the search scope.

There are other desirable post-processing methods which are open problems, for instance, query result clustering and utilizing user feedbacks.

(1) *Result Clustering.* Results of a keyword search can have many subcategories. For example, for query "Phoenix, camera, store", some stores sell mainly DSLR cameras, some sell compact cameras, and some focus on selling used cameras. Moreover, keyword "Phoenix" may have different semantics in different results: it may match a city in Arizona, appear in a review, or even appear in the description of a camera. Note that ranking schemes tend to give similar ranking scores to similar results, thus results in the same category will likely get similar ranks. If the user only browse the top results (which is the case most of the time), he/she may only see one category and miss many potentially relevant results. A natural way of solving this issue is to cluster the query results. By grouping similar results together and showing either one result per cluster or a natural language description for each cluster, the user will be able to conveniently view the result categories and navigate into the relevant ones.

Much research has been done for clustering structured data; however, they are not query-aware, thus their clustering criteria does not necessarily correspond to different interpretations of the query semantics. Furthermore, existing clustering criteria cannot be easily understood by the user, thus by looking at one result in a cluster, it can be very difficult for the user to tell in what way they are similar. Query result clustering is a useful post-processing method whose techniques await development.

(2) *Utilizing Relevance Feedback.* After the results are generated, the search engine can also interact with the user to improve the result quality. For example, the search engine can ask the user to provide some quality judgement, e.g., specifying which results are relevant or assigning satisfaction scores to some results. Based on the feedback from the user, the search engine can adjust its strategy in composing and ranking the results. The search engine may also utilize implicit feedbacks, e.g., the results clicked by the user and the time each result is browsed.

Utilization of user feedback is widely studied and adopted in information retrieval, whereas for keyword search on structured data, the problem is not much studied. The structure of the data poses new challenges for utilizing relevance feedback, e.g., the retrieval units are nodes rather than documents, and the ranking schemes for structured data can usually be more complicated than those for text documents, requiring novel techniques for incorporating feedbacks. [9] briefly discusses the problem of learning weights of edges in the schema graph for keyword searches on relational databases. Whenever the user specifies a desirable way to connect the keywords, the edge weights are adjusted, so that the score of any other way of connecting keywords is lower by a margin that is equal to the distance of the two trees. Barring this work, there are still many open problems on utilizing feedbacks. For example, it remains unclear how to adjust the weight of state-of-the-art ranking factors, such as TFIDF, number of keywords appearing in the result, height of the result tree, etc., in face of user feedbacks. User feedbacks can also be used by the search engine to either automatically refine the original query, or suggest a set of new queries that can retrieve user preferred results.

6 Acknowledgement

This material is based on work partially supported by NSF CAREER award IIS-0845647, IIS-0740129 and IIS-0915438.

References

[1] Ziv Bar-Yossef and Maxim Gurevich. Mining Search Engine Query Logs via Suggestion Sampling. VLDB, pages 54–65, 2008.

[2] Andrei Broder. A Taxonomy of Web Search. ACM SIGIR Forum, volume 36, number 2, pages 3-10, 2002.

[3] Paul - Alexandru Chirita, Claudiu S. Firan and Wolfgang Nejdl. Personalized Query Expansion for the Web. SIGIR, 2007.

[4] Yu Huang, Ziyang Liu and Yi Chen. Query Biased Snippet Generation in XML Search. SIGMOD, 2008.

[5] Georgia Koutrika, Zahra Mohammadi Zadeh and Hector Garcia-Molina. Data Clouds: Summarizing Keyword Search Results over Structured Data. EDBT, 2009.

[6] Ziyang Liu and Yi Chen. Identifying Meaningful Return Information for XML Keyword Search. SIGMOD, 2007.

[7] Ziyang Liu, Peng Sun and Yi Chen. Structured Search Result Differentiation. SIGMOD, 2009.

[8] Nikos Sarkas, Nilesh Bansal, Nilesh Bansal, Gautam Das and Nick Koudas. Measure-driven Keyword-Query Expansion. VLDB, 2009.

[9] Partha Pratim Talukdar, Marie Jacob, Muhammad Salman Mehmood, Koby Crammer, Zachary G. Ives, Fernando Pereira and Sudipto Guha. Learning to Create DataIntegrating Queries. VLDB, 2008.

[10] Yufei Tao and Jeffrey Xu Yu. Finding Frequent Co-occurring Terms in Relational Keyword Search. EDBT, 2009.