Keyword Search Over Key-Value Stores

Arash Termehchy Marianne Winslett
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
{termehch,winslett}@uiuc.edu

ABSTRACT

Key-value stores (KVSs) are the most prevalent storage systems for large scale web services. As they do not have the structural complexities of RDBMSs, they are more efficient. In this paper, we introduce the problem of keyword search over KVSs and analyze its differences with keyword search over documents and relational databases. We propose a novel method called **K**eyword **S**earch over Key-value **store**s (*KSTORE*) to solve the problem. Our user study using two real life data sets shows that *KSTORE* provides better ranking than the extended versions of document and relational database keyword search proposals and poses reasonable overheads.

Categories and Subject Descriptors: H.2.0 [Database Management]: General

General Terms: Algorithms, Designs, Performance

1. INTRODUCTION

Key-value stores are the only efficient systems to store and retrieve data for large scale Web services on cloud [1]. They store instances of an entity in form of key-value pairs where keys are unique IDs and values consist of data fields. They keep each entity in a separate table. They also partition some tables across different nodes to keep the response time fast. Keyword search is among the most popular types of queries submitted to large scale Web services. A candidate answer to a keyword query is a tuple containing all terms of the query. Keyword query interfaces for KVSs must spot the most relevant table to the query and find the most related tuple(s) in the table. Tables 1 and 2 show fragments of tables for the movies database (IMDB, *www.imdb.com*) and bibliographical database (DBLP, *dblp.uni-trier.de*). Tuples 1 and 3 in Table 1 and tuple 5 in Table 2 match query Q_1:*Artificial Intelligence*. Tuple 1 is more relevant than tuple 3 in Table 1 to Q_1, because users are more likely to search a movie by its name rather its keywords. Tuple 1 is also more relevant than tuple 5 in Table 2, as the title of a movie is more important than the venue of a paper to users. Thus, tuple 1 is the most relevant answer to Q_1. Since each table stores huge number of tuples and different tables may be in different nodes, it is not possible to keep a centralized inverted index for all tables. If a KVS forwards input queries to all its nodes, each node will receive many queries that do not match its content, which degrades the performance of the KVS. As KVSs store structured data, pure IR based methods are not appropriate. Current relational database keyword query interfaces require centralized inverted indexes. G-KS [4] use foreign key to primary key relationships inside each database to find the most relevant database to the input keyword query over multiple relational databases. We cannot use this approach as KVSs place the information of each entity in one isolated table. For instance, G-KS ranks tuples 1 and 3 the same for query Q_1 as the query terms appear at the same field in these tuples. In this paper, we introduce the problem of finding the most relevant tuple to a keyword query for KVSs and propose a novel solution called *KSTORE*. *KSTORE* addresses the problem in two steps. In the preprocessing step, it extracts the required structural information from each table and stores it in a summarized data structure. In query time, it picks the most relevant table using the summarized data structure and finds the most relevant tuples in the table. We have performed a user study over two real world data sets. Our experiments show that the preprocessing step takes reasonable time and poses modest space overhead for a one-time task. They also depict that *KSTORE* has better ranking quality than the extended versions of G-KS and pure IR-style ranking for KVSs.

Key	*Title*	*Actors*		*Keywords*
1	Artificial Intelligence	*Name*	*Born*	Future
		Law	1972	
2	1984	*Name*	*Born*	Future
		Allen	1921	
		Johns	1930	
3	I,Robot	*Name*	*Born*	Artificial Intelligence
		Smith	1970	Future
		Hogan	1965	

Table 1: IMDB Fragment

Key	*Title*	*Authors*	*Booktitle*
5	R1	Polit	Artificial Intelligence 1984
6	Intelligence	Chace	AAAI 1990

Table 2: DBLP Fragment

2. RANKING APPROACH

We model a table in a key-value store as a set of tuples $S = (k_i, v_i), 0 \leq i \leq n$ where k_i represents a unique ID and v_i represents a *value*. Each value consists of at least

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

one *column*. Each column stores a *simple value* (character string) or a multi-set of values. For instance, each tuple in in Table 1 contains columns such as *Title* and *Actors*. *Title* stores strings and *Actors* consists of multi-sets of values, where each value contains columns *Name* and *Born*. A *keyword query* is a set $Q = t_1 \cdots t_q$ of terms. The comparison between the relevant and the candidate answers to query Q_1 reveals that the more distinctive a column is the more important it is to users. We use *entropy* to measure the amount of information an element contains. More distinctive elements have higher entropies. Since each column is eventually a multi-set of simple values, we can define the entropy of a column based on the probabilities of its simple values. Given a column C that contains simple values $v_i, 0 \leq i \leq m$ with probabilities $P(v_i)$ respectively, the entropy of C is $H(C) = \sum_{1 \leq i \leq m} P(v_i) \lg (1/P(v_i))$. For instance $H(Title)$=1.58 and $H(Keywords)$=0.77 in Table 1. $H(Title)$ is also greater than $H(Keywords)$ considering all tuples in original IMDB data set. As the number of tuples in different tables may be different, we normalize the entropy of an element as $\hat{H}(C) = \frac{H(C)}{\lg(n)}$, where n is the number of tuples. Hence, the structural ranking formula for a candidate tuple is:

$$s(T) = \sum_{1 \leq i \leq q} \hat{H}(C_i), C_i \in T \quad (1)$$

where C_is are the elements that match the query terms. As the number of tuples in a KVS table is very large, the values of normalized entropies remain relatively the same unless a drastic change happens in the table. Thus, *KSTORE* has to compute the entropies of the columns only once in a relatively long time. We store the hashes of each column's simple values in a hash table in the main memory, to compute the element entropy. In our experiments, the hash table occupies at most 10MB for a table of size 400MB, which is a reasonable space overhead. We also consider each tuple as a small document and use Okapi-BM25 ranking formula ($ir(.)$) [2] in the final ranking formula as follows:

$$r(T) = \alpha s(T) + (1 - \alpha) ir(T), 0 < \alpha < 1, \quad (2)$$

We set the value of α based on empirical evaluations. The most related table to the query contains the tuple with the highest score. Hence, the score of table S is:

$$r(S) = \max_i r(T_i), T_i \in S. \quad (3)$$

According to equation 1 and the definition of $ir(.)$, we have:

$$r(T) = \sum_{1 \leq i \leq q} r(t_i), t_i \in Q. \quad (4)$$

In the preprocessing step, *KSTORE* computes the maximum score for each stemmed non-stop term in every table and stores it in the summary data structure for the table. In query time, it sums up the scores of query terms for every table and picks the table with highest score. This table is most likely to contain the most related tuple according to the equations 4 and 3. Using the inverted index for the selected table, it returns the tuple with highest score. However, this method may generate false positives as the terms might not occur in the same tuple. Moreover, each term may have different scores in different tuples. For instance, *Artificial* has different scores in tuple 1 and 3 in Table 1. Hence, even if the terms occur in the same tuple, they may not have their maximum scores in that tuple. Thus, *KSTORE* computes and stores the maximum score for every two terms occurring in the same tuple. Since the maximum number of keywords in a web query is relatively low, we expect that this summarized information to provide high quality ranking. We assign a unique ID to each term, and store the IDs in the summary data structure. In our experiments, the summary data structure occupies a modest space overhead of 3.8MB for a table of size 400MB.

G-KS	$\alpha = 0$	$\alpha = 0.2$	$\alpha = 0.26$	$\alpha = 0.4$	$\alpha = 0.6$	$\alpha = 0.8$	$\alpha = 1$
0.51	0.55	0.63	0.65	0.61	0.55	0.53	0.44

Table 3: Average MRR for different methods

3. EVALUATION

We have randomly partitioned IMDB data set (1.4 million tuples, 1.2GB) to 5 tables of different sizes between 200MB and 400MB. We have also created a separate tables for each entity type such as *book* and *article* in DBLP data set (.6 million tuples, .38GB). It has resulted in 5 tables of sizes between 50MB and 100MB. We have stored each table in Berkeley-DB4.8 in a separate node with 8GB memory and 2.5Ghz quad-core CPU. A separate machine stores the summarized information of all tables and answers the input queries. We have used the query workload from [3] containing 40 queries for IMDB and 25 queries for DBLP. The maximum time spent on entropy computation for a table is 12.1 minutes and the longest summary information construction time is 11.3 hours. We have extended G-KS for KVS keyword search. In order to implement G-KS, we have considered each column containing multi-sets of values as a table that refers to the original table through a foreign key. Table 3 shows the average mean reciprocal ranks for G-KS, pure IR style method ($\alpha = 0$), and *KSTORE* with different values for α. As we see $\alpha = 0.26$ delivers the best ranking. Customized okapi-BM25 formula works better than G-KS, but as it does not use the structural information; its average MRR is less than the best version of *KSTORE*. The average query processing time for the query load is 2.4 seconds.

4. CONCLUSION AND FUTURE WORKS

This paper introduces the problem of keyword search over KVSs and presents a novel and effective approach called *KSTORE* to address this problem. We are working on ranking top-K candidate answers and efficiently updating summary data structures.

Acknowledgements. This work is supported by NSF grant #0938071.

5. REFERENCES

[1] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[2] C. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.

[3] A. Termehchy and M. Winslett. Effective Ranking of XML Keyword Search Results (Extended Version), University of Illinois, UIUCDCS-R-2009-3043, 2009.

[4] Q. H. Vu et al. A Graph Method for Keyword based Selection of the topK Databases. In *SIGMOD*, 2008.