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 Keyword Search over Key-value stores (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 tuple 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 Q1; "Artificial Intelligence". Tuple 1 is more relevant than tuple 3 in Table 1 to Q1, 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 Q1. 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 Q1 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.

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Title</th>
<th>Authors</th>
<th>Booktitle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Artificial Intelligence</td>
<td>Name: Law</td>
<td>Born: 1972</td>
</tr>
<tr>
<td>2</td>
<td>1984</td>
<td>Name: Allen</td>
<td>Born: 1921</td>
</tr>
<tr>
<td>3</td>
<td>I,Robot</td>
<td>Name: Smith</td>
<td>Born: 1970</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Title</th>
<th>Authors</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>R1</td>
<td>Point</td>
<td>Artificial Intelligence 1984</td>
</tr>
<tr>
<td>6</td>
<td>Intelligence</td>
<td>Chace</td>
<td>AAAI 1990</td>
</tr>
</tbody>
</table>
one column. Each column stores a simple value (character string) or a multi-set of values. For instance, each tuple 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 \( Q = t_1 \cdots t_n \) 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 \( H(C) = \sum \limits_{1 \leq i \leq m} P(v_i) \log(1/P(v_i)) \). 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 \( \hat{H}(C) = \frac{H(C)}{\max(C)} \), where \( n \) is the number of tuples. Hence, the structural ranking formula for a candidate tuple is:

\[
s(T) = \sum \limits_{1 \leq i \leq q} \hat{H}(C_i), C_i \in T
\]

where \( C_i \)s are the elements that match the query terms. As the number of tuples in a KV 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 \( \alpha \) in the final ranking formula as follows:

\[
r(T) = \alpha s(T) + (1 - \alpha) ir(T), 0 < \alpha < 1
\]

We set the value of \( \alpha \) 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:

\[
r(S) = \max_{T_i \in S} r(T_i)
\]

According to equation 1 and the definition of \( ir(\cdot) \), we have:

\[
r(T) = \sum \limits_{1 \leq i \leq q} r(t_i), t_i \in Q
\]

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 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.

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 KV 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 \( \alpha \). 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