Processing Recommender Top-*N* Queries in Relational Databases

Liang Zhu[1*], Quanlong Lei[1], Guang Liu[2], Feifei Liu[1]
[1] Key Lab of Machine Learning and Computational Intelligence, School of Mathematics and Computer Science, Hebei University, Baoding, Hebei, 071000, China.
[2] School of Art, Hebei University, Baoding, Hebei, 071002, China.

* Corresponding author. Email: zhu@hbu.edu.cn
Manuscript submitted April 20, 2014; accepted September 30, 2014.

Abstract: According to the feedback information from a user in the result sets of initial or previous queries, we present in this paper a framework for processing recommender top-*N* queries in relational databases. Based on the techniques and ranking strategies of keyword search, this framework returns top-*N* results for an initial query given by the user. As soon as he or she selects some of the top-*N* results, the framework will find out related keywords from the result(s) selected by the user, calculate and modify corresponding weights of the related keywords. By using the weights, our framework determines new query words associated with the previous query to construct a recommender query. A knowledge base is created to store the related information of the tuples in the underlying database for evaluating the recommender query. The experimental results based on real datasets show the efficiency and effectiveness of our framework.

Key words: Relational database, recommendation, feedback information, top-*N* query, knowledge base.

1. Introduction

Based on the similarities between query words and data objects, Information Retrieval (IR) systems [1] by keyword search for unstructured data return the results that may not be the desired answers to a user, then he or she has to refine the query words to the system for several times. The methods for answering recommender queries are important in recommender systems [2]–[4], which are applied in many fields including books, news, research articles, live insurances, financial services, and so on. Traditional database systems [5] retrieve tuples through complex structured query language (SQL) and they support only pattern match of tuples with query conditions. The researches of keyword search in relational databases have been extensively studied since 2002 [6], [7]. For example, BANKS [6] and DISCOVER [7] are two systems that support keyword search in relational databases. They generate candidate tuple trees from multiple tables.

The BANKS system [6] handles keyword search by using a directed graph, which includes a scheme graph and a data graph. In the data graph, each tuple is represented by a node, and each “Foreign Key-Primary Key” between two tuples in the database is represented by a directed edge. The data graph uses the heuristic algorithm to search for all information nodes.

In [6], each candidate result tree has a relevance score, and every edge has a weight. Let $W(e)$ be the weight of edge e, and $N(v)$ be the weight of node v. The scoring function $Score(J)$ for a candidate result tree J is as below:

$$Score(J) = (1-\lambda)E_{score}(e) + \lambda N_{score}(v) \tag{1}$$

$$E_{score}(e) = \frac{1}{1+\sum_{e}\frac{W(e)}{W_{\min}}} \tag{2}$$

$$N_{score}(v) = \frac{N(v)}{N_{\max}} \tag{3}$$

In Equation (1), λ is a constant, $E_{score}(e)$ is the edge scoring function of an edge e, and $N_{score}(v)$ is the node scoring function of a node v; in Equation (2), W_{min} is the minimum weight of edges in the directed graph; in Equation (3), N_{max} is the maximum weight of nodes in the directed graph.

DISCOVER [7] supports both AND- and OR- semantics queries. Each tuple returned by an AND-semantics query contains all query words, while the result by an OR-semantics query contains at least one query word. Based on the breadth-first search algorithm of the graph in [7], the DISCOVER system uses the enumeration algorithm to obtain candidate tuples, and the scoring function is defined by Equation (4) as follows:

$$Score(P,q) = \sum_{f_i \in F} Score(f_i,q) \Big/ Size(P) \tag{4}$$

In Equation (4), q is a keyword query, P is the result set returned by q, F is the set of all attribute values of P, $Size(P)$ is the number of objects in P, and $Score(f_i, q)$ is the scoring function with respect to the attribute f_i and q.

In [8], Hristidis *et al*. used directed graph to deal with keyword search with a simple ranking method to compute similarities. In [9], Liu *et al*. utilized the model of computing similarities in IR to compute similarities between a given query and answer trees in relational databases.

Traditional information retrieval techniques are proposed to satisfy user's requirements extensively. However, sometimes, the techniques of traditional IR systems cannot deal with user's queries in different situations. A personalized search algorithm by using content-based filtering is presented in [10]. XML has been utilized widely in many applications, then lots of XML documents need to be managed in databases. In many applications where backend data sources are controlled by XML database management systems; therefore, keyword query is important if a user does not know the structure of the databases. BTP-Index, an XML index structure, is proposed in [11] for processing efficiently keyword queries in databases.

Keyword search will be an important component for processing recommender top-*N* queries discussed in this paper. We invoke the main idea of the techniques with a knowledge base (or an index) introduced in the previous work [12]–[14], we improve the ranking strategy in IR as our similarity in ranking the candidate results, and we obtain top-*N* answers. We create a framework to process recommender top-*N* queries in relational databases. First, we establish a knowledge base that stores the related information of tuple words in the underlying database, cache and sort the optimal results of a query. Second, according to the words in the initial/previous query and the results selected by the user, our framework finds out the correlative keywords associated with initial/previous query words to construct a new query (i.e., recommender query), and the framework returns new top-*N* results with the new query for the user. Thus, we realize the processing of recommender top-*N* queries.

2. Recommender Query Model

Assume that ***R*** is a relation with schema ***R***(TID, A_1, A_2, ..., A_m), where TID is the tuple identifier, {A_1, A_2, ..., A_m} is the set of m text attributes, and |***R***| = n (the size of ***R***, i.e., ***R*** includes n tuples). For a tuple $\boldsymbol{t} \in \boldsymbol{R}$ and an attribute $A \in \{A_1, A_2, ..., A_m\}$, $\boldsymbol{t}[A]$ is the value of the tuple ***t*** under the text attribute A and consists of one or more English word(s), which is denoted by $\boldsymbol{t}[A] = (tw_1, tw_2,..., tw_k)$, where tw_i ($1 \leq i \leq k$) is called a tuple word. For $Q_i = (qw_i, w_i)$ ($1 \leq i \leq h$) in a keyword query $Q = \{Q_1, Q_2, ..., Q_h\}$, qw_i is a query word, and w_i is the weight corresponding qw_i.

2.1. Creation of Knowledge Base

Based on the main ideas of the knowledge base of the tuple words described in [12]–[14], we create a knowledge base with Algorithm 1 to store the related information of English tuple words in the underlying database, and we build a framework by using the knowledge base to deal with recommender top-*N* queries. The knowledge base will be stored as an index table with schema ***KBTable***(*id#*, *word*, *size*, *DBValue*), where *id#* is the primary key, *word* denotes the tuple word, and *DBValue* is a string with the form as "*tid*, *cid*, *dl*, *tf*, *df*;...; *tid*, *cid*, *dl*, *tf*, *df*;". In *DBValue*, *tid* is the identifier of a tuple ***t*** containing the tuple word "*word*", i.e., *tid* = ***t***[*TID*], *cid* is the column identifier of the attribute *A*, *dl* is the number of English words (counting duplicate words) in the cell defined by *tid* and *cid*, *tf* is the number of occurrences of tuple word "*word*" in the cell with *tid* and *cid*, and *df* is the number of cells containing the tuple word "*word*" with *cid*. The attribute *size* means the number of nodes in *DBValue*, that is, the number of semicolons (";") in *DBValue*. We will not distinguish the terminologies "knowledge base" and "index" in the following discussion. Part of the index table ***KBTable*** is shown in Table 1.

Table 1. Part of the Index Table

id#	word	size	DBValue
…	…	…	…
9356	Distilled	1	0060613998,1,12,1,1;
9357	Daily	2	0060613998,1,12,1,2;0060926627,1,9,1,2;
9358	Exposing	12	0060614803,1,18,1,12;0553076426,1,2,1,12;...
…	…	…	…

For the knowledge base, its index table will be built in three steps:

1) Identify the relations, attributes and tuples to be unique in the underlying database.
2) Normalize each tuple and eliminate the meaningless symbols, characters and stop words.
3) For each tuple word tw_i, store its related information into ***KBTable***.

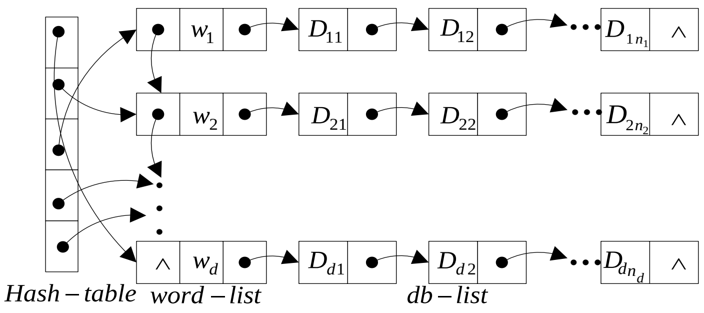


Fig. 1. Structure of knowledge base.

Being similar to the way of the previous work [13] dealing with Chinese keyword queries, the structure of the knowledge base is shown in Fig. 1, which consists of a *Hash-table* with hash function *Hash*(), a *word-list* and some *db-lists*. The process of creating our knowledge base needs three steps. (1) Normalize each tuple in

R. (2) For each tuple word $tw_i \in \boldsymbol{t}[A]$, extract related information of tw_i from tuple ***t*** and create or update a list *db-list*. (3) Create or update the *word-list* and *Hash-table*.

```
Algorithm 1 (Input: R, Output: KBTable)        /* creating the knowledge base */
1):   Initialize Hash-table and the list word-list;
2):   Read each tuple $t \in R$, and each data item $t[A]$ for each text attribute $A \in \{A_1, A_2, \ldots, A_m\}$;
3):   For each data item of $t[A]$ do
4):      Normalization the data item;
5):      Extract each tuple word $tw_i$ from $t[A]$, then obtain the related information(dl, tf, tid, cid) of $tw_i$;
6):         For each tuple word $tw_i$ of a data term do
7):            Check whether $tw_i$ is in word-list. If TRUE then update corresponding node of the tuple word
               $tw_i$ in db-list by information of cid, dl, tf, df and tid. Else create node $w_i$ and $D_{ij}$ and insert
               them into word-list and db-list respectively. Update Hash-table if necessary;
8):         End For
9):     End For
10):  Store the knowledge base into KBTable;
```

2.2. Maintenance of Knowledge Base

In order to make the query system work well, we need to maintain the knowledge base when the underlying database (or relation ***R***) is modified. The modification of ***R*** may be regarded as one of the three basic operations including insertion, deletion, and update. Since an update can be decomposed as a deletion and an insertion, we consider only insertion and deletion operations of tuples with respect to the relation ***R***. Assume that the tuple ***t*** is inserted into or deleted from ***R***.

Insertion: it is sufficient to repeat the process of creating our knowledge base with Algorithm 1 in Section 2.1 for the tuple ***t***, i.e., (1) normalize the tuple ***t***; (2) for each tuple word tw_i from $\boldsymbol{t}[A]$, update the *Hash-table*, *word-list*, and *db-lists* if necessary; (3) update the corresponding rows or columns in the table ***KBTable***.

Deletion: (1) normalize the tuple ***t***; (2) by using hash function *Hash*(), calculate the hash value $Hash(tw_i)$ for each tuple word tw_i in $\boldsymbol{t}[A]$, find out the node in *word-list* and the *db-list* corresponding tw_i, then delete the node with $tid = \boldsymbol{t}[TID]$ in the *db-list*, and decrease the value of *size* by 1 in the node; if *size* = 0, delete the node from *word-list* and update the *hash-table*; (3) update the corresponding rows or columns in the table ***KBTable***.

2.3. Definition of Similarity

Inspired by the ranking model in IR described in [1], we define the similarities between a query and tuples in order to rank the top-*N* answers. Let $\boldsymbol{T} = \{\boldsymbol{t}_1, \boldsymbol{t}_2, \ldots, \boldsymbol{t}_c\}$ be the set of candidate tuples. For $\boldsymbol{t} \in \boldsymbol{T}$, and $A \in \{A_1, A_2, \ldots, A_m\}$, we have $\boldsymbol{t}[A] = (tw_1, tw_2, \ldots, tw_k)$. Assume that $Q = \{Q_1, Q_2, \ldots, Q_h\}$, where $Q_i = (qw_i, w_i)$ $(1 \le i \le h)$, qw_i is a query word, and w_i is the weight corresponding qw_i, then we obtain the similarity between Q and ***t*** by using Equations (5)-(8) as follows.

$$Sim(Q,t) = \sum_{Q_i \in Q} Sim(Q_i, t) \tag{5}$$

$$Sim(Q_i,t) = \sum_{A \in \{A_1, A_2, \cdots, A_m\}} Sim(Q_i, t[A]) \tag{6}$$

$$Sim(Q_i, t[A]) = \sum_{qw_i \in t[A]} w_i * Sim(qw_i, t[A]) \tag{7}$$

$$Sim(qw_i, t[A]) = \frac{1+\ln(1+\ln(tf))}{(1-s)+s*\frac{dl}{avdl}} * \ln\frac{n+1}{df} \tag{8}$$

Equation (5) shows the similarity between Q and a candidate tuple $\boldsymbol{t}$, which is the sum of similarities between $\boldsymbol{t}$ and $Q_i \in Q$ (i=1, 2, ..., h). For $Q_i \in Q$, Equation (6) is the similarity between Q_i and $\boldsymbol{t}$, i.e., the sum of similarities between $\boldsymbol{t}[A]$ and Q_i for all $A \in \{A_1, A_2, ..., A_m\}$. For an attribute $A \in \{A_1, A_2, ..., A_m\}$, Equation (7) calculates the similarity between Q_i and $\boldsymbol{t}[A]$ by the inner product function, where w_i is the weight of query word qw_i in query Q, i.e., the w_i in $Q_i = (qw_i, w_i)$. Component $Sim(qw_i, t[A])$ computes the similarity for each query word qw_i in the text attribute $\boldsymbol{t}[A]$. Equation (8) is similar to the one of the most widely used weighting methods in IR [1], where $n = |\boldsymbol{R}|$ is the total number of tuples in $\boldsymbol{R}$, s is a constant (usually set to 0.2), and *avdl* is the average number of words in tuples corresponding to the attribute.

3. Query Processing

The processing of recommender top-*N* queries includes two phases: (1) deal with free-form keyword search (or keyword query) as shown in Fig. 2, and (2) generate a new/recommender query if the user chooses some result(s) of the previous keyword search, and then go to phase (1). Fig. 3 illustrates the phase (2).

We discuss the phase (1) firstly that deals with the top-*N* keyword query based on our knowledge base created in Section 2.1 and the similarity defined in Section 2.3. The process is illustrated in Fig. 2.

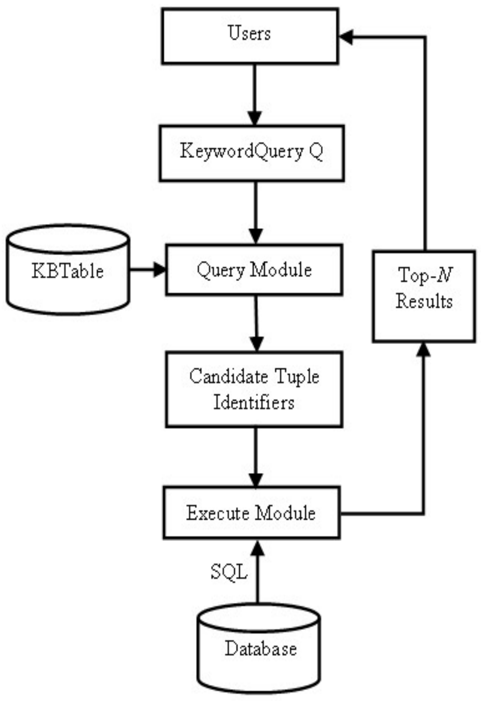


Fig. 2. Keyword Query Processing.

For a keyword query $Q = \{Q_1, Q_2, ..., Q_h\}$ where $Q_i = (qw_i, w_i)$ ($1 \le i \le h$) with the query word qw_i and weight w_i. (1) Matching query word qw_i and tuple words in our knowledge base by loading ***KBTable*** in the main memory, we obtain the set of identifiers and the related information of candidate tuples, denoted by $\boldsymbol{T}$, and we compute the similarities between Q and its candidate tuples in $\boldsymbol{T}$ by using Equations (5)-(8) (in Section 2.3) based on the related information (e.g., *dl*, *tf*, *df*). (2) Using the candidate tuple identifiers with N highest similarities and other selection conditions, we generate the SQL selection statement(s) by the Execute Module in Fig. 2, and then retrieve top-N results from the underlying database via the relational database management system. (3) We display the ranked top-N answers.

Next, we consider the evaluation of recommender query, and focus on the generation of recommender query, i.e., the new query Q_{new} in Fig. 3.

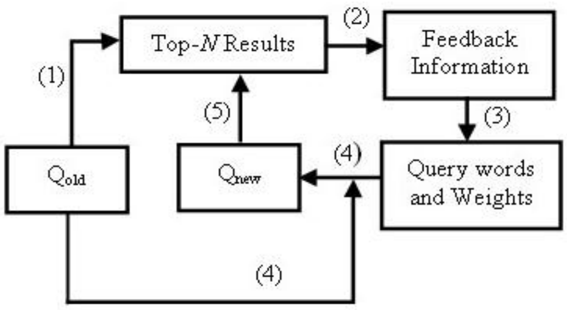


Fig. 3. Generation of recommender query.

After the user obtains the top-N results of a query Q_{old} and chooses some one(s), based on the feedback information of the result(s) chosen by the user, our framework will find out the candidate query words from the chosen result(s), and recalculate/calculate weights of the query words of Q_{old} and candidate query words. Using the weights, the query words of the recommender query Q_{new} are determined by the query words of Q_{old} and candidate query words. Then, we evaluate the query Q_{new} by the process in Fig. 2, and return recommender top-N answers to the user.

For the aim of efficiency and effectiveness, in the phase of generating and processing recommender query, we cache and sort query words with their weights as well as the top-N results of the queries, where the weights are crucial and are discussed in the following section.

4. Weights of Query Words

Due to the differences of semantic meaning in different attributes, usually, we need to define a weight of an attribute to indicate the importance of the attribute. When a user searches a book, for instance, the title of a book is more important than its publisher; therefore, the weight of attribute "*Title*" should be larger than that of attribute "*Publisher*". Defining and sorting the weights (or *I*mportant factors) of different attributes, we get the weight set $W = \{I_1, I_2, \ldots, I_m\}$ with $I_i \geq I_j$ $(i > j)$ for the attribute set $\{A_1, A_2, \ldots, A_m\}$.

For the initial query $Q = \{Q_1, Q_2, \ldots, Q_h\} = \{(qw_1, w_{o1}), (qw_2, w_{o2}), \ldots, (qw_3, w_{oh})\}$, let $w_{oi} = 1/h$ $(1 \leq i \leq h)$, where h is bounded (in our experiments, h is between 1 and 10) due to the fact that the average number of query words is not larger than 6.7 in practice [9]. According to the user's choice in the results of the initial/previous query, we find out each tuple word tw_i and calculate its weight *weight*(tw_i) by equation (9), which is related to two factors: (1) the nature of tw_i, i.e., its previous weight w_{oi}, or the attribute weight I_i, and (2) the appearance frequency of tw_i in the answers.

$$\text{weight}(tw_i) = w_i + \frac{\text{count}(tw_i)}{L} \tag{9}$$

$$\text{weight}(tw_i) := w_i + \frac{\text{weight}(tw_i)}{\sum_{i=1}^{h} \text{weight}(tw_i)} \tag{10}$$

In equation (9), $w_i = \max(w_{oi}, I_i)$ for the initial/previous weight w_{oi}, and attribute weight I_i in set $W = \{I_1, I_2, \ldots, I_m\}$; L is the number of returned results, and *count*(tw_i) is the number of tuple words tw_i appeared in the returned results if $L > 0$. If a query word qw_i is not in the results, its weight w_i does not change in equation (9). If tw_i appears in two or more attributes, we use the maximum weight in the set W. Then, we normalize and update all weights by the assignment statement (10).

5. Experimental Analysis

We use Windows XP, VC++ 6.0 and Microsoft SQL Server 2000 to carry out our experiments on a PC with Intel Core 2 Duo 2.8GHz and 2GB RAM. The database comes from a set of various kinds of English books, and stores the information of the books into the relation ***Book*** (*id#*, *Title*, *Author*, *Publisher*, *Year*), which contains 270,946 tuples (i.e., books).

For our experiments, the initial query contains two words selected randomly from the database. The workload has 70 recommender queries that are constructed by previous query words and the tuples selected from the results of the previous query according to the weights and the size of the new query. The 70 queries contain the values of "*Title*", "*Author*" and/or "*Publisher*", respectively. The workload contains seven categories of queries, i.e., (*Title*), (*Author*), (*Publisher*), (*Title*; *Author*), (*Title*; *Publisher*), (*Author*; *Publisher*), and (*Title*; *Author*; *Publisher*). Each category contains 10 queries, and the number of query words in each query is between 1 and 10. We group the queries according to their sizes, and report the arithmetic means of the elapsed times, recalls and precisions based on the queries in each group. For the elapsed time of a query, we report *Index-Time* and *Result-Time* respectively, where *Index-Time* is the elapse time of matching a query with the knowledge base/index, and *Result-Time* is time of retrieving tuples from the underlying database and other times.

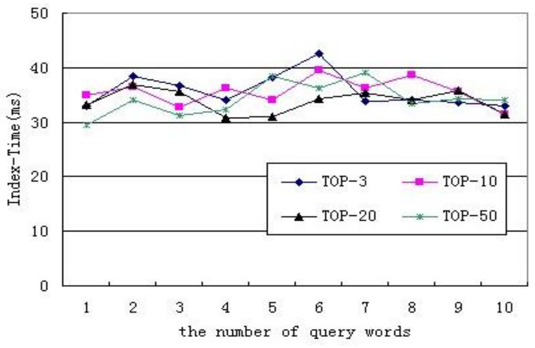


Fig. 4. Elapsed times of index.

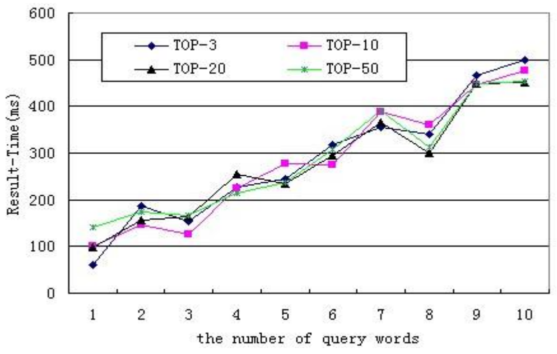


Fig. 5. Result-times for accessing database.

As shown in Fig. 4, *Index-Time* varies from 30 milliseconds to 45 milliseconds; thus, our index is efficient. From Fig. 4, we can see that the four curves are (almost) independent of the queries and the number N (N = 3, 10, 20, and 50); therefore, *Index-Time* is stable.

As illustrated in Fig. 5, *Result-Time* is between 50 and 500 milliseconds. In most cases *Result-Time* is greater than *Index-Time*, because it requires more I/O cost than *Index-Time* by using SQL selection statements. The four curves are related to queries, but not to N (N = 3, 10, 20, and 50). The general tendency of curves is that more query words will spend more *Result-Time*, and increases almost linearly as the number of query words increases.

We employ the measures *recall* and *precision* used in IR (Information Retrieval) to evaluate the effectiveness of our method in this paper.

Fig. 6 illustrates the recalls of top-N results of recommender queries. From Fig. 6, we can see that recall shows increasing trend with the increase of N for each group of the queries, and the general tendency of recalls decreases continuously with the increase of number of query words for the same *N*. The total average recalls are 0.19, 0.37, 0.50, and 0.63 for *N* = 3, 10, 20 and 50, respectively.

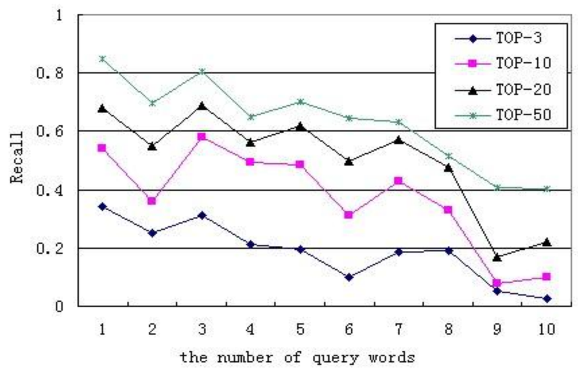


Fig. 6. Recalls of recommender top-*N* queries.

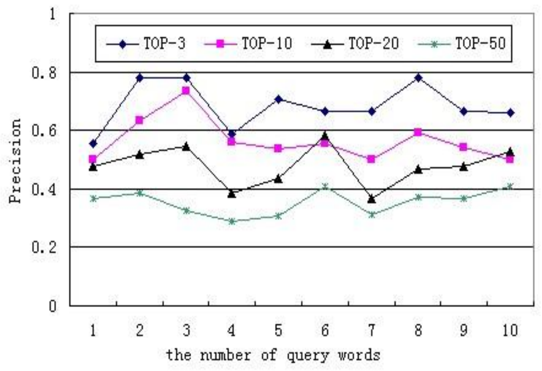


Fig. 7. Precisions of recommender top-N queries.

Fig. 7 shows the precisions of top-*N* results of recommender queries. The precisions show the trend of decrease with the increase of *N* as showed in Fig. 7, i.e., a smaller *N* indicates more matching tuples appear in its top-*N* results. The total average precisions are 0.68, 0.56, 0.48, and 0.35 for *N* = 3, 10, 20, and 50 respectively.

6. Conclusions

In this paper, we proposed a new method to realize keyword search for recommender top-*N* queries in a relation database. The method builds a knowledge base to store the related information of the tuples in the database, and improves the classic ranking strategy in Information Retrieval to calculate the similarity between a query and a candidate tuple as a ranking strategy. Given a keyword query by a user, the top-*N* answers are obtained based on the knowledge base and ranking strategy presented in this paper. Using the selected items by the user, our method finds new query words, and recalculates/calculates the weights. Then, a recommender top-*N* query is generated according to the weights by using the candidate tuple words and the initial query words. Extensive experiments are carried out to measure the performance of our method. For top-*N* (*N* = 3, 10, 20, and 50) queries with 1 to 10 query words, experimental results show that the *Index-Times* are between 30 and 45 milliseconds, and the *Result-Times* are from 50 to 500 milliseconds. Recalls are between 0.19 and 0.63, while precisions are from 0.35 to 0.68.

The knowledge base (or index) is the key component in our framework. In the future, we plan to apply the mechanism of database access control in [15] and the idea of clustering algorithm in [16] to ameliorate our knowledge base, and we will optimize our algorithm for the weights of query words to improve the recalls and precisions of recommender queries. It would be interesting to consider that multiple candidate recommender queries are listed by using the auto complete techniques of the Textbox as described in [17], which will provide the conveniences for the users and increase the efficiency in refining the recommender queries.

Acknowledgment

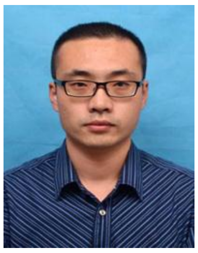
This work is supported in part by NSFC (61170039) and the NSF of Hebei Province (F2012201006).

References

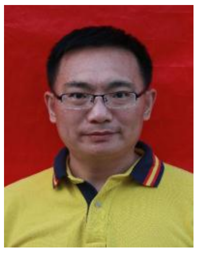
[1] Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. Addison-Wesley Professional.
[2] Melville, P., & Sindhwani, V. (2010). Recommender systems. In Sammut C., & Webb G. (Eds.),

Encyclopedia of machine learning (pp. 829-838). Berlin: Springer.
[3] Jannach, D., Zanker, M., Felfernig, A., & Friedrich, G. (2010). *Recommender Systems: An Introduction*. Cambridge: Cambridge University Press.
[4] Shani, G., & Gunawardana, A. (2011). Evaluating recommendation systems. In F. Ricci, L. Rokach, B. Shapira, & P. B. Kantor (Eds.), *Recommender Systems Handbook* (pp. 257-297). Berlin: Springer.
[5] Silberschate, A., Korth, H., & Sudarshan, S. (2001). *Database System Concept* (4th ed.). Beijing: McGraw-Hill Companies, Inc.
[6] Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., & Sudarshan, S. (2002). Keyword searching and browsing in database using BANKS. In R. Agrawal, & K. R. Dittrich (Eds.), *Proceedings of the 18th International Conference on Data Engineering* (pp. 431-440). San Jose, CA, USA: IEEE Computer Society.
[7] Hristidis, V., & Papakonstantinou, Y. (2002). Discover: Keyword search in relational database. *Proceedings of 28th International Conference on Very Large Data Bases* (pp. 670-681). Hong Kong, China: Morgan Kaufmann.
[8] Hristidis, V., Gravano, L., & Papakonstantionou, Y. (2003). Efficient IR-style keyword search over relational database. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, & A. Heuer (Eds.), *Proceedings of 29th International Conference on Very Large Data Bases* (pp. 850-861). Berlin, Germany: Morgan Kaufmann.
[9] Liu, F., Yu, C., Meng, W., & Chowdhury, A. (2006). Effective keyword search in relational database. In S. Chaudhuri, V. Hristidis, & N. Polyzotis (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 563-574). Chicago, Illinois, USA: ACM.
[10] Zeng, C., Xing, C., & Zhou, L. (2003). A personalized search algorithm by using content-based filtering. *Journal of Software*, *14(5)*, 999-1004.
[11] Jin, Y., & Bao, X. (2010). An efficient XML index for keyword query with semantic path in database. *Journal of Software*, *5(10)*, 1052-1059.
[12] Zhu, L., Ma, Q., Liu, C., Mao, G., & Yang, W. (2010). Semantic distance based evaluation of ranking queries over relational databases. *Journal of Intelligent Information Systems*, *35(3)*, 415-445.
[13] Zhu, L., Zhu, Y., & Ma, Q. (2010). Chinese keyword search over relational databases. *Proceedings of second WRI World Congress on Software Engineering* (pp. 217-220). Wuhan, China: IEEE Computer Society.
[14] Zhu, L., Pan, L., & Ma, Q. (2012). Chinese keyword search by indexing in relational databases. *Journal of Software Engineering and Applications*, *5*, 107-112.
[15] Ma, X. Q. (2011). Database deductive access control. *Journal of Computers*, *6(5)*, 1024-1031.
[16] Yuan, D. R., Cuan, Y. W., & Liu, Y. Q. (2014). An effective clustering algorithm for transaction databases based on k-mean. *Journal of Computers*, *9(4)*, 812-816.
[17] Yao, Z. Q., & Sen, A. (2013). Implementation of the autocomplete feature of the textbox based on AJAX and web service. *Journal of Computers*, *8(9)*, 2197-2203.

Liang Zhu received the BS degree in mathematics in 1983 from Nankai University, China, the M.S. degree in applied mathematics in 1990 from Hebei University, China, and Ph.D. degree in computer science in 2009 from Beijing University of Technology, China. He is currently a professor in the School of Mathematics and Computer Science at Hebei University, China. He was a visiting research scholar at the Department of Computer Science, State University of New York at Binghamton, USA from 2002 to 2003. He has published more than 30 papers in mathematics and computer science in various journals and international conference proceedings, including JMAA, WAIM, DKE, and JIIS. He has been a member or principal investigator for several projects of software engineering and software products. His research interests include information retrieval, machine learning, database system, keyword search, top-*N* query, and query processing and optimization.



Quanlong Lei received his B.S. degree in computer science in 2008 from Weifang University, China. He is currently working on his M.S. degree in computer science in the School of Mathematics and Computer Science at Hebei University, China. His current research interests include information retrieval, and keyword query in relational database. He has won the university-level scholarship of outstanding student and the prize of the excellent graduation dissertation for his B.S. degree.



Guang Liu is currently an associate professor in the School of Art at Hebei University, China. His research interests include information retrieval, audio retrieval, and multimedia database systems. He has published more than 10 papers, and won the first prize of the twelfth multimedia design competition of Ministry of Education.

Feifei Liu received her B.S. degree in information and computing science in 2011 from Hebei University, China. She is currently working on her M.S. degree in computer science in the School of Mathematics and Computer Science at Hebei University, China. Her current research interest includes top-*N* query and keyword search in relational database. She has won the university-level scholarships of outstanding student for three consecutive years.