Efficient Continuous Top-k Keyword Search in Relational Databases

Yanwei Xu[1,3], Yoshiharu Ishikawa[2,3], and Jihong Guan[1]

[1] Department of Computer Science and Technology, Tongji University, Shanghai, China
[2] Information Technology Center, Nagoya University, Japan
[3] Graduate School of Information Science, Nagoya University, Japan

Abstract. Keyword search in relational databases has been widely studied in recent years. Most of the previous studies focus on how to answer an instant keyword query. In this paper, we focus on how to find the top-k answers in relational databases for continuous keyword queries efficiently. As answering a keyword query involves a large number of join operations between relations, reevaluating the keyword query when the database is updated is rather expensive. We propose a method to compute a range for the future relevance score of query answers. For each keyword query, our method computes a state of the query evaluation process, which only contains a small amount of data and can be used to maintain top-k answers when the database is continually growing. The experimental results show that our method can be used to solve the problem of responding to continuous keyword searches for a relational database that is updated frequently.

Key words: Relational databases, keyword search, continuous queries, incremental maintenance

1 Introduction

As the amount of available text data in relational databases is growing rapidly, the need for ordinary users to be able to search such information effectively is increasing dramatically. Keyword search is the most popular information retrieval method because users need to know neither a query language nor the underlying structure of the data. Keyword search in relational databases has recently emerged as an active research topic [1–7].

Example 1 In this paper, we use the same running example of database *Complaints* as in [3] (shown in Figure 1). In this example, the database schema is $R = \{Complaints, Products, Customers\}$. There are two foreign key to primary key relationships: $Complaints \rightarrow Products$ and $Complaints \rightarrow Customers$.

If a user gives a keyword query "maxtor netvista", the top-3 answers returned by the keyword search system of [5] are c_3, $c_3 \rightarrow p_2$ and $c_1 \rightarrow p_1$, which are obtained by joining relevant tuples from multiple relations to form a meaningful answer to the query. They are ranked by relevance scores that are computed by a ranking strategy.

Approaches that support keyword search in relational databases can be categorized into two groups: *tuple-based* [1, 6, 8–10] and *relation-based* [2–5, 7]. After a user inputs a keyword query, the relation-based approaches first enumerate all possible *query plans* (relational algebra expressions) according to the database schema, then these plans are evaluated by sending one or more corresponding SQL statements to the RDBMS to find inter-connected tuples.

Complaints

tupleId	*prodId*	*cusId*	*date*	*comments*
c_1	p121	c3232	6.30.02	"disk crashed after just one week of moderate use on an IBM Netvista X41"
c_2	p131	c3131	7.3.02	"lower-end IBM Netvista caught fire, starting apparently with disk"
c_3	p131	c3143	8.3.02	"IBM Netvista unstable with Maxtor HD"
...	...	...	...	...

Products

tupleId	*prodId*	*manufacturer*	*model*
p_1	p121	"Maxtor"	"D540X"
p_2	p131	"IBM"	"Netvista"
p_3	p141	"Tripplite"	"Smart 700VA"
...	...	...	...

Customers

tupleId	*cusId*	*name*	*occupation*
u_1	c3232	"John Smith"	"Software Engineer"
u_2	c3131	"Jack Lucas"	"Architect"
u_3	c3143	"John Mayer"	"Student"
...	...	...	...

Fig. 1. A Running Example taken from [3] (Query is "maxtor netvista". Matches are underlined)

In this paper, we study the problem of *continuous* top-k keyword searches in relational databases. Imagine that you are a member of the quality analysis staff at an international computer seller, and you are responsible for analyzing complaints of customers that are collected by customer service offices all over the world. Complaints of customers are arriving continuously, and are stored in the complaints database shown in Example 1. Suppose you want to find the information related to Panasonic Note laptops, then you issue a keyword query "panasonic note" and use one of the existing methods mentioned above to find related information. After observing some answers, you may suspect that some arriving claims will also be related to Panasonic Notes, so you want to search the database continuously using the keyword query. How should the system support such a query?

A naive solution is to issue the keyword query after one or several new related tuples arrive. Existing methods, however, are rather expensive as there might be huge numbers of tuples matched and they require costly join operations between relations. If the database has a high update frequency (as in the situation of the aforementioned example), recomputation will place a heavy workload on the database server.

In this paper we present a method to maintain answers incrementally for a top-k keyword search. Instead of full, non-incremental recomputation, our method performs incremental answer maintenance. Specifically, we retain the state for each query which is obtained through the latest evaluation of the query. A state consists of the current top-k answers, the query plans, and the related statistics. It is used to maintain top-k answers incrementally after the database is updated.

In summary, the main contributions of this paper are as follows:

– We introduce the concept of a continuous keyword query in relational databases. To the best of our knowledge, we are the first to consider the problem of incremental maintenance of top-k answers for keyword queries in relational databases.

– We propose a method for efficiently answering continuous keyword queries. By storing the state of a query evaluation process, our algorithm can handle the insertion of new tuples in most cases without reevaluating the keyword query.

The rest of this paper is organized as follows. In Section 2 the problem is defined. Section 3 briefly introduces the framework for answering continuous keyword search in relational databases. Section 4 presents the details of our method and Section 5 shows our experimental results. Section 6 discusses related work. Conclusions are given in Section 7.

2 Problem Definition

We first briefly define some terms used throughout this paper (detailed definitions can be found in [3, 5, 7]). A relational database is composed of a set of relations $R_1, R_2, \cdots, R_n$. A *Joint-Tuple-Tree* (*JTT*) T is a joining tree of different tuples. Each node is a tuple in the database, and each pair of adjacent tuples in T is connected via a foreign key to primary key relationship. A JTT is an answer to a keyword query if it contains more than one keyword of the query and each of its leaf tuples contains at least one keyword. Each JTT corresponds to the results produced by a relational algebra expression, which can be obtained by replacing each tuple with its relation name and imposing a full-text selection condition on the relations. Such an algebraic expression is called a *Candidate Network* (*CN*) [3]. For example, Candidate Networks corresponding to the two answers c_3 and $c_3 \rightarrow p_2$ of Example 1 are $Complaints^Q$ and $Complaints^Q \rightarrow Products^Q$, respectively (the notation Q indicates the full-text selection condition). A CN can be easily transformed into an equivalent SQL statement and executed by the RDBMS. Relations in a CN are called *tuple sets* (*TS*). A tuple set R^Q is defined as a set of tuples in relation R that contain at least one keyword in Q.

A *continuous keyword query* consists of (1) a set of distinct keywords, that is, $Q = w_1, w_2, \cdots, w_{|Q|}$, and (2) a parameter k indicating that a user is only interested in the top-k answers ranked by the relevance. The main difference of a continual keyword query to keyword queries in previous work [3, 10] is that the user wants to keep the top-k answers list up-to-date while the database is updated continuously. Table 1 summarizes the notation we use in the following discussion.

Table 1. Summary of Notation

Notation	Description	Notation	Description
t	a tuple in a database	T	a joining tree of different tuples
$R(t)$	the relation corresponding to t	$sizeof(T)$	the number of tuples in T
Q	a keyword query	CN	a candidate network
R^Q	the set of tuples in R that contain at least one keyword of Q	$score(T, Q)$	the relevance score of T to Q
		$tscore(t, Q)$	the relevance score of a tuple t to Q

We adopt the IR-style ranking strategy of [3]. The relevance score of a JTT T is computed using the following formulas based on the TF-IDF weighting.

$$score(T, Q) = \frac{\sum_{t \in T}(tscore(t, Q))}{sizeof(T)}$$
$$tscore(t, Q) = \sum_{w \in t \cap Q} \frac{1 + \ln(1 + \ln(tf_{t,w}))}{1 - s + s \cdot \frac{dl_t}{avdl}} \cdot \ln\left(\frac{N + 1}{df_w}\right), \tag{1}$$

where $tf_{t,w}$ is the frequency of keyword w in tuple t, df_w is the number of $R(t)$ tuples that contains w ($R(t)$ means the relation that includes t), dl_t is the size (i.e., number of characters) of t, $avdl$ is the average tuple size, N is the total number of $R(t)$ tuples, and s is a constant.

3 Query Processing Framework

Figure 2 shows our framework for continuous keyword query processing in relational databases.

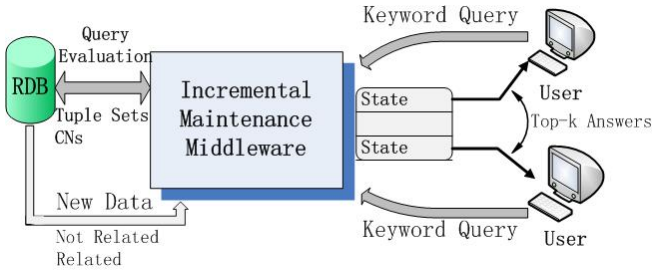


Fig. 2. Continuous Query Processing Framework

Given a keyword query, we first identify the top-k query results. Specifically, we first generate all the non-empty query tuple sets R^Q for each relation R. Then these non-empty query tuple sets and the schema graph are used to generate a set of valid CNs. Finally, the generated CNs are evaluated to identify the top-k answers. For the step of CN evaluation, several query evaluation strategies have been proposed [3, 5]. Our method of CN evaluation is based on the method of [3], but can also find the JTTs that have the potential to become top-k answers after some new tuples are inserted.

At the end of the CN evaluation process, the *state* of the process is computed and stored. After being notified of new data, the Incremental Maintenance Middleware (*IMM*) starts the answer maintenance procedure for each continuous keyword query.

The IMM uses some filter conditions to categorize the new data into two types for each keyword query based on their relevance: *not related* and *related*. Then the related new data and the stored state are used by the IMM to start the incremental query evaluation process and compute the new top-k answers. If the variations of the new top-k answers fulfill the update conditions, the new top-k answers are sent to the corresponding users.

4 Continuous Keyword Query Evaluation

In this section, we first present a two-phase CN evaluation method for creating the state for a keyword query. Then we will show how to calculate the effects of new tuples.

4.1 State of a Continuous Keyword Query

Generally speaking, two tasks need to be done after tuples are inserted. New tuples can change the values of df, N and $avdl$ in Eq. (1) and hence change the tuple scores of existing tuples. Therefore, the first task is to check whether some of the current top-k answers can be replaced by other JTTs whose relevance scores have been increased. The new tuples may also lead to new JTTs and new CNs. Therefore, the second task is to compute the new JTTs and check whether any of them can be top-k answers.

For the first task, a naive solution is to compute and store all the JTTs that can be produced by evaluating the CNs generated when the query is evaluated for the first time. After new tuples are inserted, we recompute the relevance score of the stored JTTs and update the top-k answers. This solution is not efficient if the number of existing tuples is large, since it needs to join all the existing tuples in each CN and store a large number of JTTs.

Fortunately, our method only needs to compute and store a small subset of the JTTs. For this purpose, we use the two-phase CN evaluation method shown in Algorithm 1 to evaluate a set of candidate networks *CNSet* for keyword query Q efficiently, and create the state of Q. The first phase (lines 1-11) is for computing the top-k answers, based on the method of [3]. The second phase (lines 15-22) is for finding the JTTs that have the potential to become top-k answers.

The key idea of lines 1-11 is as follows. All CNs of the keyword query are evaluated concurrently following an adaptation of a *priority preemptive*, *round robin* protocol [12], where the execution of each CN corresponds to a process. Tuples in each tuple set are sorted in descending order of their tuple scores (line 2). There is a *cursor* for each tuple set of all the CNs that indicates the index of the tuple to be checked next (line 3). All the combinations of tuples before the cursor in each tuple set have been joined to find the JTTs. At each loop iteration, the algorithm checks the next tuple of the "most promising" tuple set from the "most promising" CN (lines 8-10). The first phase stops immediately after finding the top-k answers, which can be identified when the score of the current k-th answer is larger than all the priorities of the CNs (line 6). We call the tuples before the cursor of each tuple set the "*checked* tuples", and the tuples with indexes not smaller than the cursor are called "*unchecked* tuples".

Figure 3 shows the main data structure of our CN evaluation method. In order to facilitate discussion, only the CN $Complaints^Q \rightarrow Products^Q$ is considered and we suppose that we want to find the top-2 answers. In Figure 3(a), tuples in the two tuple sets are sorted in descending order of tuple scores and are represented by their primary keys. Arrows between tuples indicate the foreign key to primary key relationship. The top-2 answers discovered are $c1 \rightarrow p1$ and $c3 \rightarrow p2$. All the tuples in the deep background have been joined in order to obtain the top-2 answers. For example, tuple $p1$ has been joined with tuples $c1$, $c2$ and $c3$, and one valid JTT $c1 \rightarrow p1$ has been found. After the execution of phase 1, the two *cursor*s of the two tuple sets are pointing at $c4$ and $p6$, respectively.

Algorithm 1 *CNEvaluation*($CNSet, k, Q$)

Input: *CNSet*: a set of candidate networks; k: an integer; Q: a keyword query;

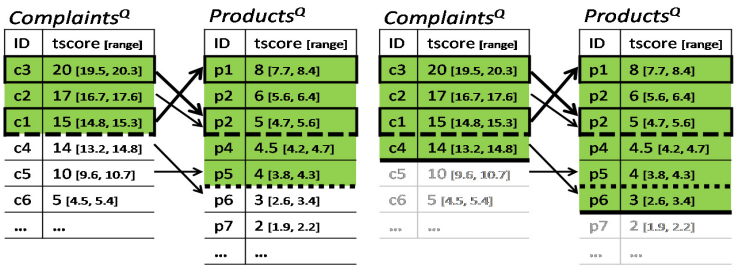
1: **declare** *RTemp*: a queue for not-yet-output results ordered by descending $score(T, Q)$; *Results*: a queue for output results ordered by descending $score(T, Q)$
2: Sort tuples of each tuple set in descending order of *tscore*
3: Set *cursor* of each tuple set of each CN in *CNSet* to 0
4: **loop**
5: Compute the priorities of each CN in *CNSet*
6: **if**(the score of the k-th answer in *Results* is larger than all the priorities) **then break**
7: Output to *Results* the JTTs in *RTemp* with scores larger than all the priorities
8: Select the next tuple from the tuple set that has the maximum upper bound score from the CN with the maximum priority for checking
9: Add 1 to the *cursor* of the tuple set corresponding to the checked tuple
10: Add all the resulting JTTs to *RTemp*
11: **end loop**
12: *FindPotentialAnswers*(*CNSet*, *Results*)
13: Set *cursor* = *cursor2* for each tuple set of each CN in *CNSet*
14: Create state for Q and return the top-k JTTs in *Results*
15: **Procedure** *FindPotentialAnswers*(*CNSet*, *Results*)
16: Compute the range of *tscore* for all the tuples in each tuple set of *CNSet* and sort the tuples in each tuple set below the *cursor* in descending order of $tscore^{\max}$
17: *lowerBound* ← the minimum lower bound of *score*s of the top-k answers in *Results*
18: **for all** tuple set ts_j of each CN C_i in *CNSet* **do**
19: Increase the value of *cursor*2 from *cursor* until $max(ts_j[cursor2]) < lowerBound$
20: **for all** tuple set ts_j of each CN C_i in *CNSet* **do**
21: Join the tuples between *cursor* and *cursor2* of ts_j with the tuples before the *cursor* in the other tuple sets of CN_i
22: Add the resulting JTTs to *Results* whose upper bound of *score* is larger than *lowerBound*

The procedure $FindPotentialAnswers$ is used to find the potential top-k answers. The basic idea of our method is to compute a range of future tuple scores using the scoring function for computing tuple scores given in Eq. (1):

$$tscore(t, Q) = \sum_{w \in t \cap Q} \frac{1 + \ln(1 + \ln(tf_{t,w}))}{1 - s + s \cdot \frac{dl_t}{avdl}} \cdot \ln\left(\frac{N+1}{df_w}\right). \tag{2}$$

We consider the situation where (a) at most ΔN new tuples are inserted; and (b) document frequency changes slightly due to the insertion. Δdf denotes the maximum increased count of the document frequency for every term. Note that the change for a keyword w Δdf_w may be 0. We assume that the average document length (*avdl*) is a *constant* to simplify the problem. Let us use the shorthand notation $A(t, w) = \frac{1+\ln(1+\ln(tf_{t,w}))}{1-s+s\cdot\frac{dl_t}{avdl}}$ and $B(t, w) = \frac{1+\ln(1+\ln(tf_{t,w}))}{1-s+s\cdot\frac{dl_t}{avdl}} \cdot \ln\left(\frac{N+1}{df_w}\right)$. $B(t, w)$ represents the contribution of keyword w to $tscore(t, Q)$.

We derive an upper bound and a lower bound of Eq.(2) which are valid while the two constraints ΔN and Δdf are satisfied. First, we compute the maximum score for the existing tuples t. This situation occurs when all the terms in $t \cap Q$ do not appear in



(a) Compute the top-2 answers (b) Find potential top-2 answers

Fig. 3. Two-phase CN evaluation

the new documents; hence, we have $tscore(t,Q)^{\max} = \sum_{w \in t \cap Q} A(t,w) \cdot \ln\left(\frac{N+1+\Delta N}{df_w}\right)$. For each $B(t,w)$, the minimum value is achieved when the first dt_w new tuples all contain w: $B(t,w)^{\min} = A(t,w) \cdot \ln\left(\frac{N+1+\Delta df_w}{df_w+\Delta df_w}\right)$. Therefore, the lower bound of $tscore(t,Q)$ is $tscore(t,Q)^{\min} = \sum_{w \in t \cap Q} A(t,w) \cdot \ln\left(\frac{N+1+\Delta dt_w}{df_w+\Delta df_w}\right)$. Note that this lower bound only can be achieved when all the Δdt_w are equal. Using such ranges, the range of relevance scores of a JTT T can be computed as $[\sum_{t \in T} t.tscore^{\min}, \sum_{t \in T} t.tscore^{\max}] \cdot \frac{1}{sizeof(T)}$.

We continually monitor the change of statistics to determine whether the thresholds ΔN and Δdf are violated. This is not a difficult task: monitoring ΔN is straightforward; for Δdf, we accumulate Δdf_w for all the terms w in the process of handling new tuples. In the following discussion, we consider only the case that the two thresholds ΔN and Δdf are not violated.

For each tuple t in C, we use $max(t) = \left(t.tscore^{\max} + \sum_{t \notin ts_i} \max(ts_i)\right) \cdot \frac{1}{sizeof(C)}$ to indicate the maximum upper bound of *score*s of the possible JTTs that contain t, where $max(ts_i)$ indicates the maximum upper bounds of *tscore*s of tuples in ts_i. If $max(t)$ is larger than the minimum lower bound of *score* of answers in *Results* (*lowerBound* in line 17), t can form some JTTs with the potential to become top-k answers in the future. We find such tuples in lines 18-19, and join them with the tuples before *cursor2* in the other tuple set (line 21). Hence, all the JTTs that are formed by the tuples that have the potential to form top-k answers are computed. However, not all the JTTs computed in line 21 can become top-k answers in the future. In line 22, only the JTTs whose upper bound of score is larger than *lowerBound* are added to *Results*. After the execution of line 12, *Results* contains the top-k answers and the potential top-k answers.

In line 14, the state for Q is created based on the snapshot of *CNEvaluation*. The state contains three kinds of data:

- The keyword statistics: the number of tuples, and document frequencies (i.e., the number of tuples that contain at least one keyword).
- The set of candidate networks: all the checked tuples (checked tuples of multiple instances of one tuple set are merged to reduce the storage space).
- The JTT queue *Results*: each entry contains the tuple ID and the *tscore*.

Note that the tuples before *cursor2* in each tuple set can be considered as highly related to the keyword query and have a high possibility to form JTTs with newly inserted

tuples. Hence, they need to be stored in the state for the second task. We need to store the statistic $\sum_{w \in t \cap Q} A(t, w)$ for each tuple t in the state in order to recompute the tuple scores after new tuples are inserted. Fortunately, the value is static and does not change once we compute it.

Figure 3(b) shows the data structure of the CN $Complaints^Q \rightarrow Products^Q$ after the second phase of evaluation. The two tuple sets are further evaluated by checking tuples $c4$ and $p6$, respectively.

4.2 Handling Insertions of Tuples

After receiving a new tuple, the IMM first checks whether the values of df and N still satisfy the assumptions, that is, the differences between the current values of df and N and their values when the state was first created are smaller than Δdf and ΔN, respectively. If the assumptions are satisfies, the algorithm *Insertion* shown in Algorithm 2 is used to incrementally maintain the top-k answers list for a keyword query; if the assumptions are not satisfied, then the query must be reevaluated.

In Algorithm 2, lines 1-3 are for the first task, and lines 5-18 are used to compute the new JTTs that contain the new tuples. In line 1, the values of N and df for the relation $R(t)$ are updated. Then, if it is necessary (line 2), the relevance scores of the JTTs in the JTT queue are updated using the new values of N and df(line 3).

Algorithm 2 *Insertion*(t, Q, S)

Input: t: new tuple; Q: keyword query; S: stored state for Q
Output: New top-k answers of Q

```
 1: update the keyword statistics of R(t)
 2: if there are some tuples of R(t) are contained in the JTT queue of S then
 3:     recompute the scores of the JTTs in the JTT queue of S
 4: if t does not contain the keywords of Q then return
 5: if R(t)^Q is a new tuple set then generate new CNs
 6: compute the value and range for the tscore of t
 7: CNSet ← CN in S that contains R(t)^Q ⋃ all the new CNs
 8: for all CN C in CNSet do
 9:     for all R(t)^Q of C do
10:         if t.tscore^max > min^C(R(t)^Q) then
11:             add t to the checked tuples set of R(t)^Q
12:             join t with the checked tuples in the other tuple sets of C
13:             if t.tscore^max > max^C(R(t)^Q) then
14:                 for all the other tuple sets ts of C do
15:                     query the unchecked tuples of ts from the database
16:                     delete the newly inserted tuples from ts that have not been processed
17:                 call FindPotentialAnswers({C}, S.Queue) while replacing R(t)^Q by {t}
18: end for
19: return S.Queue.Top(k)
```

If $R(t)^Q$ is a new tuple set, the new CNs that contain $R(t)^Q$ need to be generated (line 5). In line 6, the value of *tscore* for the new tuple is computed using the actual

values of df and N; but the values of df and N used for computing the range of *tscore* are the values when the state is created in order to be consistent with the ranges of *tscore*s for existing checked tuples of $R(t)^Q$. New tuples can be categorized into two groups by deciding whether each new tuple belongs to the new top-k answers (*related* or *not related*). Generally speaking, new tuples that do not contain any keywords of the query are not related tuples (line 4), and new tuples that contain the keywords may be related. However, a new tuple t that contains the keywords cannot be related is its upper bound of *tscore* is not larger than $min^C(R(t)^Q)$, which is the minimum $tscore^{\max}$s of checked tuples of $R(t)^Q$ (line 10). For the related new tuples, they are processed from line 11 to line 17. In line 12, t is joined with the checked tuples in the other tuple sets of C. Then the algorithm uses another filtering condition, $t.tscore^{\max} > max^C(R(t)^Q)$ in line 13, to determine whether the new tuple t should be joined with the unchecked tuples of the other tuple sets of C. If $t.tscore^{\max} > max^C(R(t)^Q)$, which is the maximum $tscore^{\max}$s of checked tuples of $R(t)^Q$, some $max(ts_i(cursor2))$ may be larger than the minimum lower bound of current top-k answers. Hence, after querying the unchecked tuples from the database in line 15, the procedure *FindPotentialAnswers* of C is called while replacing $R(t)^Q$ by $\{t\}$ (line 17, set of the new tuple). Note that the relevance scores of the new JTTs produced in lines 12 and 17 should be computed using the actual values of dfs and Ns.

The execution of lines 14-17, needed to query unchecked tuples from the database and perform the second phase of the evaluation of C, place a heavy workload on the database. However, our experimental studies show a very low execution frequency for lines 14-17 when maintaining the top-k answers for a keyword query.

5 Experimental Study

For the evaluation, we used the DBLP[4] data set. The downloaded XML file is decomposed into 8 relations, article(articleID, key, title, journalID, crossRef, $\cdots$), aCite(id, articleID, cite), author(authorID, author), aWrite(id, articleID, authorID), journal(journalID, journal), proc(procID, key, title, $\cdots$), pEditors(pEditorID, Name), procEditor(id, procEditorID, procID), where underlines and underwaves indicate the keys and foreign keys of the relations, respectively. The numbers of tuples for the 8 relations are, 1092K, 109K, 658K, 2752K, 730, 11K, 12K, 23K. The DBMS used is MySQL (v5.1.44) with the default configurations. Indexes were built for all primary key and foreign key attributes, and full-text indexes were built for all text attributes.

We manually picked a large number of queries for evaluation. We attempted to include a wide variety of keywords and their combinations in the query set, taking into account factors such as the selectivity of keywords, the size of the relevant answers, and the number of potential relevant answers. We focus on the 20 queries with query lengths ranging from 2 to 3, which are listed in Table 2.

Exp-1 (Parameter tuning) In this experiment, we want to study the effects of the two parameters on computing the range of future tuple scores. The number of tuples that need to be joined in the second phase of CN evaluation is determined by ΔN and Δdf. Small values of ΔN and Δdf result in small numbers of tuples be joined, but a

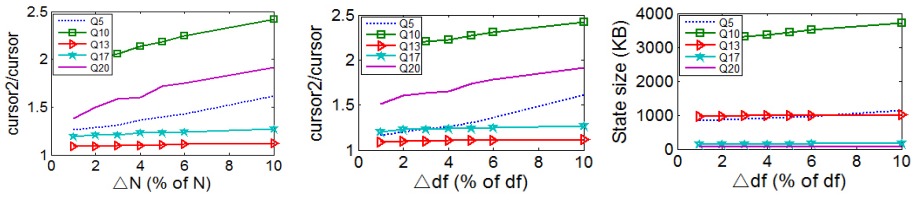
[4] http://dblp.mpi-inf.mpg.de/dblp-mirror/index.php

Table 2. Queries

QID	Keywords	QID	Keywords	QID	Keywords	QID	Keywords
Q1	bender, p2p	Q6	sigmod, xiaofang	Q11	Hardware, luk, wayne	Q16	Staab, Ontology, Steffen
Q2	Owens, VLSI	Q7	constraint, nikos	Q12	intersection, nikos	Q17	query, Arvind, parametric
Q3	p2p, Steinmetz	Q8	fagin, middleware	Q13	peter, robinson, video	Q18	search, SIGMOD, similarity
Q4	patel, spatial	Q9	fengrong, ishikawa	Q14	ATM, demetres, kouvatsos	Q19	optimal, fagin, middleware
Q5	vldb, xiaofang	Q10	hong, kong, spatial	Q15	Ishikawa, P2P, Yoshiharu	Q20	hongjiang, Multimedia, zhang

large frequency of recomputing the state because the increases of N and df will soon exceed ΔN and Δdf, respectively, due to the insertion of tuples. Therefore, the values of ΔN and Δdf represent a tradeoff between the storage space for the state and the efficiency for top-k answer maintenance. In our experiments, the values of ΔN and Δdf are set to be a *fraction* of the values of N and df, respectively. For each query, we run the two-phase CN evaluation algorithm with different values of ΔN and Δdf. The main experimental results for five queries are shown in Figure 4.

We use two metrics to evaluate the effects of the two parameters. The first is $cursor2/cursor$ where *cursor* and *cursor2* indicate the summation of numbers of checked tuples after the first and second phases of CN evaluation, respectively. Small values of $cursor2/cursor$ imply a small number of tuples are joined in the second phase for computing the potential top-k answers. The second metric is the size of the state. Figures 4(a) and 4(b) show the changes of $cursor2/cursor$ for different ΔN and Δdf while fixing the other parameter to 10%. Figure 4(a) and 4(b) show that only a small number of tuples are joined in the second phase of CN evaluation, which implies that the range of tuple scores computed by our method is very tight. The curves in Figure 4(a) and 4(b) are not very steep. Hence, we can use some relatively large value of ΔN and Δdf when creating the state for a continuous keyword query. Note that the values of N in a database are always very large. Therefore, even a small value of ΔN (like 10%) can result in the state being valid until a large number of new tuples (100,000 in our experiment) have been inserted, as long as the Δdfs condition is not violated. Figure 4(c) shows the change of the state size for a query when varying Δdf while keeping $\Delta N = 10\%$. The data size of the state of a continuous keyword query is quite small (several MBs at most); hence, the IMM can easily load the state of a query for answer maintenance.



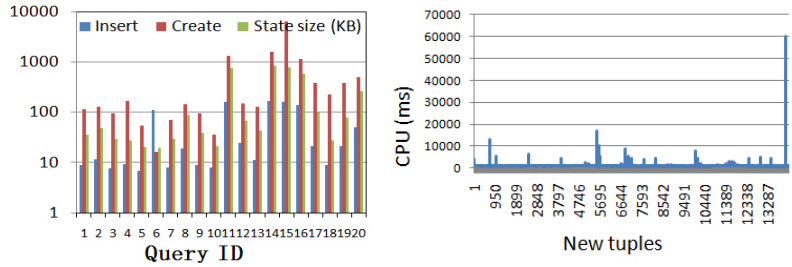
(a) Varying ΔN when $\Delta df = 10\%$ (b) Varying Δdf when $\Delta N = 10\%$ (c) Change of state size

Fig. 4. Effect of ΔN and Δdf

Exp-2 (Efficiency of answer maintenance) In this experiment, we first create states for the 20 queries. Then we sequentially insert 14,223 new tuples into the database. The CPU times for maintaining the top-k answers for the 20 queries after each new tuples being inserted are recorded. All the experiments are done after the DBMS buffer has been warmed up. The values of ΔN and Δdf are both set to 1%. As the values of ΔN and Δdf are very small, the cost for creating the state of a query is essentially as the cost for the first phase of CN evaluation.

Figure 5(a) shows the time cost to create states (*Create*) and the average time cost of the 20 queries to handle the 14,223 new tuples (*Insert*). Note that the times are displayed using a log scale. From Figure 5(a), we can see that the more time used to create the state of a query, the more time is used to maintain answers for the query. In our experiment, the states of the 20 queries are stored in the database. The states of the queries are read from the database after the IMM receives a new tuple. The time for maintaining the new tuples also contains the time cost of reading the states from database and writing them back to database after handing new tuples. Hence, such time costs represent a large proportion of the total time cost for handling new tuples when they are not related. In order to reveal this relationship, we also plot the state sizes for the 20 queries in Figure 5(a). The cost of reading and writing a state is clearly revealed by the data for Q6. The data of Q6 appears to be an exception because the value of *Insert* is larger than *Create*. The main reason for this is that Q6 is very easy to answer. Hence, the time used to load and write back the state is the majority for handling new tuples for Q6.

Figure 5(b) shows the total time for handing each inserted tuple. In most cases, the time used to handle a new tuple is quite small, which corresponds to the situation that the new tuple does not contain any keyword from the 20 queries. Hence, the algorithm only needs to update the scores of JTTs in the JTT queue of the states. The peaks of the data in Figure 5(b) correspond to the situations in which some queries need to be reevaluated because of violation of the Δdf. Eventually, ΔN is violated, hence several queries need to be reevaluated at the same, this results in the highest peak in Figure 5(b).



(a) Time for creating states and the average time for handling new tuples

(b) Total time for handling each new tuple

Fig. 5. Efficiency of maintaining the top-k answers

6 Related Work

Keyword search in relational databases has recently emerged as a new research topic [11]. Existing approaches can be broadly classified into two categories: ones based on candidate networks [2, 3, 7] and others based on Steiner trees [1, 8, 10].

DISCOVER2 [3] proposed ranking tuple trees according to their IR relevance scores to a query. Our work adopts the *Global Pipelined* algorithm of [3], and can be viewed as a further improvement to the direction of continual keyword search in relational databases. SPARK [5] proposed a new ranking formula by adapting existing IR techniques based on the natural idea of a virtual document. They also proposed two algorithms, based on the algorithm of [3], that minimize the number of accesses to the database. Our method of incremental maintenance of top-k query answers can also be applied to these algorithms, which will be a direction of future work.

7 Conclusion

In this paper, we have studied the problem of finding the top-k answers in relational databases for a continuous keyword query. We proposed storing the state of the CN evaluation process, which can be used to restart the query evaluation after the insertion of new tuples. An algorithm to maintain the top-k answer list on the insertion of new tuples was presented. Our method can efficiently maintain a top-k answers list for a query without recomputing the keyword query. It can, therefore, be used to solve the problem of answering continual keyword searches in a database that is updated frequently.

Acknowledgments

This research is partly supported by the Grant-in-Aid for Scientific Research, Japan (#22300034), the National Natural Science Foundation of China (NSFC) under grant No.60873040, 863 Program under grant No.2009AA01Z135 and Open Research Program of Key Lab of Earth Exploration & Information Techniques of Ministry of China (2008DTKF008). Jihong Guan was also supported by the Program for New Century Excellent Talents in University of China (NCET-06-0376) and the "Shu Guang" Program of Shanghai Municipal Education Commission and Shanghai Education Development Foundation.

References

1. Aditya, B., Bhalotia, G., Chakrabarti, S., Hulgeri, A., Nakhe, C., Parag, Sudarshan, S.: BANKS: Browsing and keyword searching in relational databases. In: VLDB. (2002) 1083–1086
2. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: Enabling keyword search over relational databases. In: ACM SIGMOD. (2002) 627
3. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style keyword search over relational databases. In: VLDB. (2003) 850–861
4. Liu, F., Yu, C., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. In: ACM SIGMOD. (2006) 563–574
5. Luo, Y., Lin, X., Wang, W., Zhou, X.: SPARK: Top-k keyword query in relational databases. In: ACM SIGMOD. (2007) 115–126
6. Li, G., Zhou, X., Feng, J., Wang, J.: Progressive keyword search in relational databases. In: ICDE. (2009) 1183–1186
7. Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword search in relational databases. In: VLDB. (2002) 670–681
8. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB. (2005) 505–516
9. He, H., Wang, H., Yang, J., Yu, P.S.: Blinks: Ranked keyword searches on graphs. In: ACM SIGMOD, New York, NY, USA, ACM (2007) 305–316
10. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: EASE: An effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In: ACM SIGMOD. (2008) 903–914
11. Wang, S., Zhang, K.: Searching databases with keywords. J. Comput. Sci. Technol. 20(1) (2005) 55–62
12. Burns, A.: Preemptive priority based scheduling: An appropriate engineering approach. In Advances in Real Time Systems. S. H. Son, Prentice Hall. (1994) 225–248