リレーショナルデータベースにおける連続的トップkキーワード検索手法

許　延偉† 石川　佳治††

† 名古屋大学大学院情報科学研究科
†† 名古屋大学情報基盤センター
E-mail: †xuyanwei@db.itc.nagoya-u.ac.jp, ††ishikawa@itc.nagoya-u.ac.jp

あらまし　データベースに対する問合せの発行には，ユーザには問合せ言語やスキーマに関する知識が必要となる．このため，与えられたキーワード集合に対する適切な問合せ結果を返す，データベースに対するキーワード検索の研究が進んでいる．本研究では連続的問合せの状況におけるトップk問合せの効率的な処理方式について議論する．
キーワード　リレーショナルデータベース，キーワード検索，連続的問合せ，インクリメンタル管理

Efficient Continuous Top-*k* Keyword Search in Relational Databases

Yanwei XU † and Yoshiharu ISHIKAWA††

† Graduate School of Information Science, Nagoya University
†† Information Technology Center, Nagoya University
E-mail: †xuyanwei@db.itc.nagoya-u.ac.jp, ††ishikawa@itc.nagoya-u.ac.jp

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 efficiently answer continuous top-*k* keyword queries in relational databases. As involving a large number of join operations between relations when answering a keyword query, reevaluating the keyword query as long as 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 compute 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 growing continually. The experimental results show that our method can be used to settle the problem of answering continuous keyword search in 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 growing rapidly, the need for ordinary users to effectively search such information 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 the previous work [3] (shown in Figure 1). In the 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 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 their relevance scores computed by a ranking strategy.

In the literature, the reported 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] ones. 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.

In this paper, we study the problem of *continuous* top-*k* keyword search in relational databases. Assume you are a quality analysis staff at an international computer seller, and you are responsible for analyzing complaints of customers that are collected by cus-

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 from [3] (Query is "maxtor netvista". Matches are Underlined)

tomer service offices all over the world. Complaints of customers are coming continuously, and are stored in the database Complaints shown in Example 1. Suppose you want to find the information related to Panasonic notes, then you give a keyword query "panasonic note" and use one of the existing methods mentioned above to find related information. After observing some answers, you may doubt whether some newly arrived claims are more related to Panasonic notes, i.e., you want to continuously search the database using the keyword query. How should the system do to support such an query?

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

In this paper, we present a method to incrementally maintain answers for a top-k keyword search. Instead of full, non-incremental recomputation, our method performs incremental answer maintenance. Specifically, we keep the state of each query 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 incrementally maintain top-k answers 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 a state of a query evaluation process, our algorithm can handle the insertion of new tuples in most cases without restarting the keyword query.

The rest of this paper is organized as follows. Section 2 gives the problem definition. Section 3 briefly introduces the framework for answering continuous keyword queries in relational databases. Section 4 proposes the details of our method. Section 5 shows the experimental results. Section 6 presents the related work. Section 7 draws the conclusions.

2. Problem Definition

We first define some terms used throughout this paper, whose detailed definition 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 must contain 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 algebra 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 means 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. For example, the two tuple sets $Complaints^Q$ and $Products^Q$ in Example 1 are $\{c_1, c_2, c_3\}$ and $\{p_1, p_2\}$ respectively.

A *continuous keyword query* consists of (1) a set of distinct keywords, i.e., $Q = w_1, w_2, \cdots, w_{|Q|}$, and (2) a parameter k indicating that a user is only interested in top-k answers ranked by the relevance. The main difference of a continual keyword query to keyword queries in the 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. Let us return to the example in Section 1. After the evaluation of the three CNs, two JTTs with the highest relevance scores c_3 and $c_3 \leftarrow p_2$ are found. Then they are returned to the user as the top-2 answers. Suppose new tuples arrive continuously, as our computer dealer example mentioned in Section 1, the top-2 answers need to be updated if some arrived new claims are more related to "maxtor" and "netvista". For the following discussions, we summarize the notations we use in Table 1.

In keyword search in relational databases [3], answers of keyword queries are often ranked using an IR-Style ranking strategy. We first assign a *tscore* to each tuple in a given JTT by using a standard IR-ranking formula, which models each tuple in a relation as a document and all the tuples as a document collection. Then we combine the individual tuple scores together by using a monotonic aggregation function to obtain the final score. In this paper, we adopt the same ranking strategy. The relevance score of a JTT T is computed

Table. 1 Summary of Notations

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

using the following formulas based on the TF-IDF weighting. The scoring function is used in [3].

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

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 continuous query processing framework of continuous keyword search on relational databases.

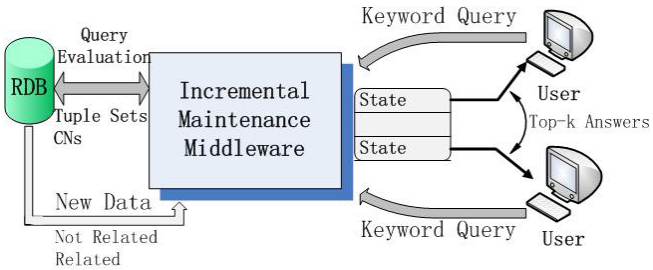


Fig. 2 Continuous Query Processing Framework

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 by [3], [5]. Instead of issuing a SQL query for each CN and combining them to find the top-k results, they issue multiple lightweight SQL queries and can stop the query execution immediately after the top-k answers are determined. Our method of CN evaluation is based on the method of [3], but can also find out the JTTs that have the potential to become top-k answers after some new tuples are inserted.

At the end of the CNs evaluation process, in order to achieve incremental maintenance of the query answers, the *state* of the process is computed and stored. A state consists of the found JTTs, the set of generated CNs, the tuples that have been joined, and the statistics of the keywords. After being notified new data, the Incremental Maintenance Middleware (*IMM*) start the answers 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.

We will present the details of the states and how to restart the query evaluation process in the next section.

4. Continuous Keyword Query Evaluation

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

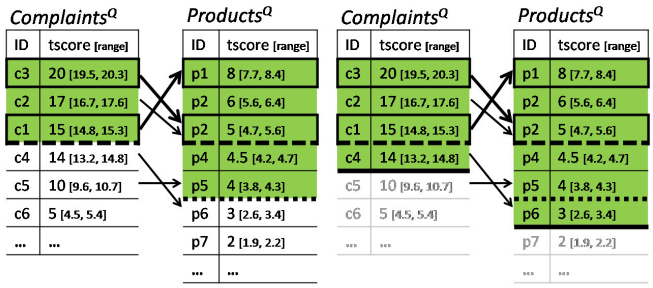
4.1 State of Continuous Keyword Query

Generally speaking, two tasks need to be done after newly tuples are inserted: arrived new tuples can change the values of df, N and $avdl$ in Eq. (1), hence change the tuple scores of existing tuples. Therefore the first task is to check whether some current top-k answers can be replaced by other JTTs whose relevance scores are increased. The arrived new tuples may lead new JTTs and new CNs. Therefore, the second task is to compute the new JTTs and check whether some 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 generated CNs 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 are 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 portion of JTTs. For this purpose, we use the two-phase CN evaluation method shown in Algorithm 1 to efficiently evaluate a set of candidate networks *CNSet* for keyword query Q, and create the state of Q. The first phase (line 1-11) is for computing the top-k answers, which is based on the method of [3]. The second phase (line 15-22) is for finding the JTTs that have the potential to become top-k answers.

The key idea of line 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) Compute the top-2 answers (b) Find potential top-2 answers

Fig. 3 Two-phase CN evaluation

a *cursor* for each tuple set of all the CNs that indicates the index of the tuple for next checking (line 3). All the combinations of tuples before cursor in each tuple set have been joined to find the JTTs. For each tuple set ts_i in a CN C, the algorithm use an upper bound function to bound the relevance scores of potential answers that contain the tuple $ts_i[cursor]$. The maximum upper bound scores of all the tuple sets of a CN is regarded as the *priority* of the CN (line 5), which ensures that any potential results from future execution of the CN will not have a higher score. At each loop iteration, the algorithm picks the next tuple of the "most promising" tuple set from the "most promising" CN for checking (line 8-10). The first phase will stop immediately after finding out the top-k answers, which can be identified when the score of the current top-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 as *checked* tuples since all the combinations of them have been joined to find the joining tuples trees, and call the tuples with indexes not smaller than the cursor as *unchecked* tuples.

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

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

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

We consider the situation where a) at most ΔN new tuples are inserted; and b) document frequency slightly changes due to the insertion. Δdf denotes the maximum increased count of the document frequency for every term before ΔN new tuples are inserted. Note that 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 notations $A(t,w) = \frac{1+\ln(1+\ln(tf_{t,w}))}{1-s+s\cdot\frac{dl_t}{avdl}}$ 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 the upperbound and the lowerbound of Eq.(2) which are valid while the two constraints ΔN and Δdf are fulfilled. First, we compute the maximum score for the existing tuples t. The situation occurs when all the terms in $t \cap Q$ do not appear in 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 contains w: $B(t,w)^{\min} =$

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 in descending $score(T,Q)$; *Results*: a queue for outputted results in descending $score(T,Q)$
2: Sort tuples of each tuple set according to *tscore* in descending order
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 the JTTs in *RTemp* to *Results* with score 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* = *cursor*2 of 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* $\leftarrow$ 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 *cursor*2 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*

$A(t,w) \cdot \ln\left(\frac{N+1+\Delta df_w}{df_w+\Delta df_w}\right)$. Therefor the lowerbound 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_ws are equal. Using such ranges, the range of relevance score 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 watch the change of statistics to monitor 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 sequel, we consider the situation 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 potential to become top-k answers in the future. We find such tuples in line 18-19, and join them with the tuples before *cursor*2 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 JTTs that have the potential to become top-k answers. Tuples can form the current or potential top-k answers are before the *cursor*2 in each tuple set.

In line 14, the state for Q is created based on the snapshot of *CNEvaluation*. The state mainly 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*: its each entry contains the tuple ID and the *tscore*.

Note that the tuples before *cursor*2 in each tuple set can be considered as highly related to the keyword query and have high possibilities to form JTTs with newly inserted tuples, hence they need to be stored in the state for the second task. However, checked tuples in the tuple sets that reference another tuple sets through consider the situation that foreign key to primary key relationships need not to be stored since they can not reference new inserted tuples. We need to store the statistics value $\sum_{w \in t \cap Q} A(t,w)$ for each tupe 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) presents 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 keep the assumption, i.e., the differences between the current values of df and N and the values when the state was firstly created are smaller than Δdf and ΔN, respectively. If the assumption is fulfilled, the algorithm *Insertion* shown in Algorithm 2 is used to incrementally maintain the top-k answers list for a keyword query, otherwise the query must be reevaluated.

In Algorithm 2, line 1-3 are for the first task, and line 5-18 are used to compute the the new JTTs that contain the new tuples. In line 1, the values of of N and df of 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 of 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 set ts of C do
15:                    query the unchecked tuples of ts from the database
16:                    delete the new inserted tuples from ts that have not been
                       processed
17:                call FindPotentialAnswers({C}, S.Queue) while regarding ts
                   only contains 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* of the new tuple is computed using the actual values of df and N, but the values of df and N 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 of 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* and *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 keywords may

be related. However, new tuple t that contains the keywords cannot be related in case that 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 regarding $R(t)^Q$ only contains the new tuple t (line 17). Note that the relevance scores of the new JTTs produced in line 12 and 17 should be computed using the actual values of dfs and Ns.

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

5. Experimental Study

For the evaluation,we use the DBLP(*1) 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 of the 8 relations are, 1092K, 109K, 658K, 2752K, 730, 11K, 12K, 23K. The DBMS used is SQL Server 2005 Developer Edition with default configurations. Indexes were built on all primary key and foreign key attributes, and fulltext indexes are built on 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, such as the selectivity of keywords, the size of the relevant answers, the number of potential relevant answers, etc. We focus on 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 of computing the range of future tuple scores. The amount of tuples 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 number of tuples be joined, but large frequency of recomputing the state because the increases of

(*1)：http://dblp.mpi-inf.mpg.de/dblp-mirror/index.php

Table. 2 Queries

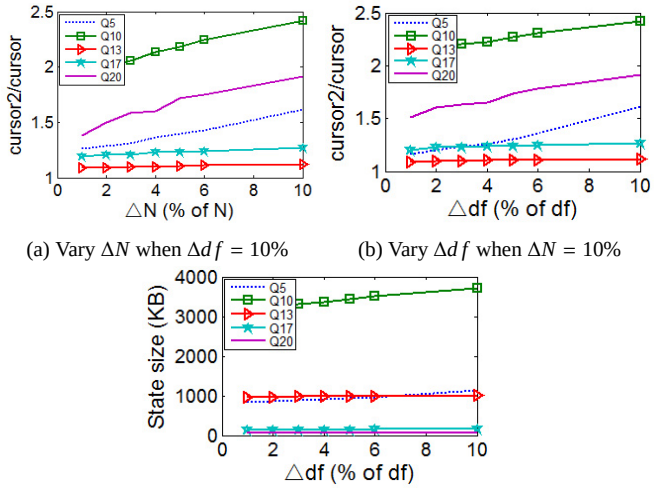
QID	Keywords	QID	Keywords
Q1	bender, p2p	Q11	Hardware, luk, wayne
Q2	Owens, VLSI	Q12	intersection, nikos
Q3	p2p, Steinmetz	Q13	peter, robinson, video
Q4	patel, spatial	Q14	ATM, demetres, kouvatsos
Q5	vldb, xiaofang	Q15	Ishikawa, P2P, Yoshiharu
Q6	sigmod, xiaofang	Q16	Staab, Ontology, Steffen
Q7	constraint,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

values of N and df will soon exceed ΔN and Δdf, respectively, due to insertion of tuples. Therefore, the values of ΔN and Δdf are a tradeoff between the storage space for the state and the efficiency for top-k answers maintenance. In our experiments, the values of ΔN and Δdf are set to be the *percent* 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 experiment results of five queries are shown in Figure 4.

We use two metrics to eualuate the effects of the two parameters. (1) $cursor2/cursor$. For each keyword query, we sum up the values of $cursor2$ and $cursor$ of all the tuple sets respectively after the two-phase CN evaluation, and then compute the ratio of the two summations. (2) The size of the state. Figures 4(a) and 4(b) show the changes of $cursor2/cursor$ to different ΔN and Δdf while fixing Δdf and ΔN to 10%, respectively. 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 score 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, therefor even a small value of ΔN (like 10%) can results in the state being valid before a large number of new tuples (100, 000 in our experiment) are inserted as long as Δdfs are not violated. Figure 4(c) shows the change of the state size for a query when varying Δdf while ΔN = 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 answers maintenance.

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

Figure 5(a) shows the time cost to create states (*Create*) and the



(a) Vary ΔN when $\Delta df = 10\%$ (b) Vary Δdf when $\Delta N = 10\%$

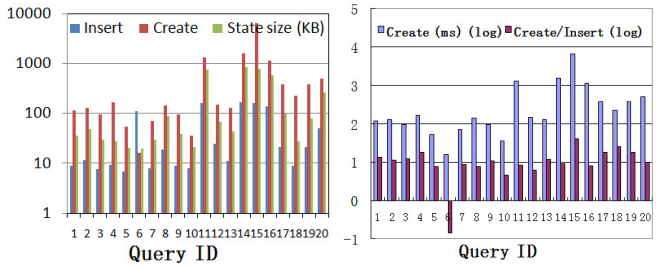
(c) Change of the state size

Fig. 4 Effect of ΔN and Δdf

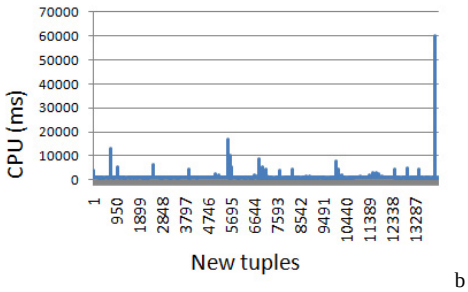
average time cost of the 20 queries to handle the 14,223 new tuples (*Insert*). Note that the are in log scale. From Figure 5(a), we can find that the more time used to create a state of a query, the more time used to maintain answers for the query. In our experiment, the states of the 20 queries are stored to the database. The states of the queries are read from the database after the IMM receive 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 occupy a large percent of the time cost for handling new tuples when they are not related. In order to reveal such relationship, we also plot the state sizes of the 20 queries in Figure 5(a). The cost of reading and writing back a state can be highly revealed by the data of Q6. The data of Q6 seems as an exception because the value of *Insert* is larger than *Create*. The main reason is that Q6 is very easy to answer. Hence the time used to load and write back the state is the majority of time for handling new tuples for Q6.

Figure 5(b) presents the ratio of *Create* to *Insert*, which shows that the more time cost to create a state of a query, the more speed-up ratio is achieved. Figure 5(c) shows the total time for handing each inserted new tuple. In most cases, the time used to handle a new tuple is quite small, which corresponding to the situation that the new tuple do not contain any keyword of the 20 queries. Hence the algorithm only need to update the scores of JTTs in the JTT queue of the states. The peaks of the data in Figure 5(c) correspond to the situations that some queries need to be reevaluated due to violation of Δdf. At last, ΔN is violated, hence several queries need to be reevaluated at the same, which results in the highest peak in Figure 5(c).

Table 3 presents the executed times of lines of algorithm *Insertion* when handling the 14, 223 new tuples for each query. Different lines corresponding to different relevance of new tuples. We omit the times that the new tuple is not related, i.e., only lines 1-3 are executed. If algorithm *Insertion* execute to line 5, the new tuple contains some keywords. If algorithm *Insertion* execute to line 11, the new tuple is related and is joined with stored tuples in the state. If the new tuple is more related, it need to be joined with unchecked tuples, which results in the execution of line 14 of *Insertion*. The *violate* line in Table 3 are the times of Δdf is violated. Table 3 shows that most of the highly related tuples are stored in the state for each keyword, which results in small numbers of execution of line 14 and violation of Δdf.



(a) Time for creating states and the average time for handling new tuples (b) Speed-up ratios of each query



(c) Total time for handling each new tuple

Fig. 5 Efficiency of maintaining top-k answers

Table. 3 Executed times of lines of Algorithm *Insertion*

QID	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
line 5	14	2	14	300	0	0	70	17	0	320
line 11	1	0	0	0	0	0	0	1	0	2
line 14	0	1	0	0	1	0	0	0	1	2
violate	1	1	1	5	1	1	1	1	1	6

QID	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
line 5	7	8	98	0	0	5	215	295	85	267
line 11	9	1	0	13	14	24	2	0	0	54
line 14	0	0	0	0	0	3	0	0	0	0
violate	1	1	1	1	1	10	2	2	1	1

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: those based on candidate networks [2], [3], [7] and others based on Steiner trees [1], [8], [10].

DISCOVER2 [3] proposed ranking of tuple trees according to their IR relevance scores to a query. Our work adopt 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 a natural notion of a virtual document. They also proposed two algorithms that have minimal accesses to the database, which are based on the algorithm of [3]. Our method of incremental maintenance of top-k query answers can also be applied to them, which will be a direction of the future work.

7. Conclusion

In this paper, we studied the problem of answer continuous top-k keyword query in relational databases. We proposed to store the state of the CN evaluation process, which can be used to restart the query evaluation after the insertion of new tuples. An algorithm was presented to maintain the top-k answer list on the insertion of new tuples. Our method can efficiently maintain a top-k answers list for a query without recomputation the keyword query, which can be used to settle 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 (#19300027, #21013023).

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