Efficient Continual Top-$k$ Keyword Search in Relational Databases

YANWEI XU$^{1,a}$, YOSHIHARU ISHIKAWA$^{2,3,4,b}$, JIHONG GUAN$^{1,c}$

Received: April 11, 2011, Accepted: September 12, 2011

Abstract: Keyword search in relational databases has been widely studied in recent years because it requires users neither to master a certain structured query language nor to know the complex underlying database schemas. Most existing methods focus on answering snapshot keyword queries in static databases. In practice, however, databases are updated frequently, and users may have long-term interests on specific topics. To deal with such situations, it is necessary to build effective and efficient facilities in a database system to support continual keyword queries. In this paper, we propose an efficient method for answering continual keyword queries over relational databases. The proposed method consists of two core algorithms. The first one computes a set of potential top-$k$ results of keyword queries while the database is continually being updated. The second core algorithm uses these states to maintain the top-$k$ results of keyword queries while the database is continually being updated. Experimental results validate the effectiveness and efficiency of the proposed method.

Keywords: relational databases, information retrieval, keyword search, continual queries, results maintenance

1. Introduction

With the proliferation of text data available in relational databases, simple ways to explore such information effectively are of increasing importance. Keyword search in relational databases, with which a user specifies his/her information need by a set of keywords, is a popular information retrieval method because the user needs to know neither a complex query language nor the underlying database schemas. It has attracted substantial research efforts in recent years, and a number of methods have been developed, which can be categorized into two groups: graph-based methods [1], [2], [3], [4], [5] and schema-based methods [6], [7], [8], [9], [10].

Example 1 Consider a sample publication database shown in Fig. 1. Figure 1 (a) shows the three relations Papers, Authors, and Writes. In the following, we use the initial of each relation name (P, A, and W) as its shorthand. There are two foreign key references: $W \rightarrow A$ and $W \rightarrow P$. Figure 1 (b) illustrates the tuple connections based on the foreign key references. For the keyword query “James P2P” consisting of two keywords “James” and “P2P,” there are six tuples in the database that contain at least one of the two keywords (underlined in Fig. 1 (a)). They can be regraded as the results of the query. However, they can be joined with other tuples according to the foreign key references to form more meaningful results, several of which are shown in Fig. 1 (c). The arrows represent the foreign key references between the corresponding pairs of tuples. Finding such results which are formed by the tuples containing the keywords is the task of keyword search in relational databases. As described later, results are often ranked by relevance scores evaluated by a certain ranking strategy.

Most of the existing keyword search methods assume that the databases are static and focus on answering snapshot keyword queries. In practice, however, a database is often updated frequently, and the result of a snapshot query becomes invalid once the related data in the database is updated. For the database in Fig. 1, if publication data comes continually, new publication records are inserted into the three tables. Such new records may be more relevant to “James” and “P2P.” And some results may be removed due to deletions. Thus, a continual evaluation facility for keyword queries is essential in dynamic databases. Though various techniques have been proposed for answering snapshot keyword queries, there is no available method which is directly applicable to evaluating continual keyword queries in an on-line fashion.

A naïve solution is to reevaluate the keyword queries from scratch after the database is updated. The existing methods for snapshot top-$k$ keyword queries [8], [10], however, are rather expensive as they require costly join operations between relations. If the database has a high update rate, this naïve solution will incur heavy workload on the database server.

For continual keyword query evaluation, when the database is updated, two situations must be considered:

(1) Some top-$k$ results may be replaced by new ones that involve
the new tuples and some may expire due to deletions.

(2) The relevance scores of existing results may change because the underlying statistics (e.g., word frequencies) is changed.

This paper addresses the problem of continual top-k keyword search in relational databases with high update rates. We present an effective and efficient method to constantly report the top-k results of every monitoring query as the database is continually being updated. Our method consists of two core algorithms. The outline of the proposed approach is as follows:

- When a continual query is issued, the first algorithm Calc-State computes the range of the future relevance score for every query result, by assuming that at most a fixed number of tuples are inserted or deleted in every relation. Then, besides the current top-k results, the algorithm seeks the set of potential top-k results, whose upper bounds of relevance scores are higher than the minimal lower bounds of relevance scores of the current top-k results. The top-k results, potential top-k results, and the set of most relevant tuples of each relation are used to create a lightweight state.

- When new tuples enter the database, the second algorithm Insertion first updates the relevance scores of the results saved in the state, and then finds the new potential top-k results that contain the new tuples. In most cases, the tuples saved in the state are adequate to compute the new results. After tuples are deleted, results containing the deleted tuples are deleted. If too many top-k results are deleted from the state, it is re-calculated.

The rest of this paper is organized as follows. Section 2 discusses the related work. In Section 3, some basic concepts are introduced and the problem is defined. Section 4 introduces the framework for answering continual keyword search in relational databases. Section 5 presents the details of the proposed method and Section 6 gives the experimental results. Finally, in Section 7 we conclude and discuss future extensions of this paper.

2. Related Work

In this section, we survey the related work from three aspects: keyword search in relational databases, keyword search over relational data streams, and materialized view maintenance.

2.1 Keyword Search in Relational Databases

Given the l-keyword query \( Q = \{w_1, w_2, \ldots, w_l\} \), the task of keyword search in a relational database is to find structural information constructed from tuples in the database[12]. There are two approaches. The schema-based approaches in this area utilize the database schema to generate SQL queries which are evaluated to find the structures for a keyword query. After receiving a keyword query, they first utilize the database schema to generate a set of relation join templates, which can be interpreted as select-project-join views. Then, these join templates are evaluated by sending the corresponding SQL statements to the DBMS for finding the query results. The graph-based methods model and materialize the entire database as a directed graph where the nodes are relational tuples and the directed edges are foreign key references between tuples. Then for each keyword query, they find a set of structures (either Steiner trees[1], distinct rooted trees[3], r-radius Steiner graphs[5], or multi-center subgraphs[13]) from the database graph, which contain all or some of the query keywords and are connected by the paths in
database graph. For further details, please refer to the survey papers [12], [14]. The materialized data graph should be updated for any database changes; hence this model is not appropriate for databases that change frequently [14]. Therefore, this paper adopts the schema-based framework and can be regarded as an extension to continual keyword searching.

2.2 Keyword Search in Relational Data Streams

The most related projects to our paper are S-KWS [15] and KDynamic [16], which try to find new results or expired existing results for a given l-keyword query in order to monitor events that are implicitly interrelated over an open-ended, high-speed large relational data stream [12]. They adopt the schema-based framework since the database is not static. The main issue they addressed is how to reduce CPU cost (for evaluating joins) and memory overhead (for storing intermediate results) in the join template evaluation step while tuples are inserted/deleted at high speeds. The basic idea of their methods is to share the computational cost by using either operator mesh [15] or L-Lattice [16].

Our paper deals with a different problem from the two papers above, though all need to respond to continual queries in a dynamic environment. They focus on finding all valid query results. On the contrary, we maintain the top-k results, which is less sensitive to updates of the underlying databases because not every new or expired results change the top-k results. Therefore, our main concern is how to prune the database updates that do not change the top-k results.

2.3 Materialized View Maintenance

The problem considered in this paper can be viewed as the maintenance of materialized top-k views. To ensure the correctness of a query result, a materialized view must be up-to-date whenever it is accessed by a query. Materialized views maintenance has received considerable attention in the research community over the last two decades [17], [18], [19], [20]. References [18] and [19] address the problem of maintaining the top-k views that are defined on a relation, while Ref. [20] tackles the problem of top-k view maintenance in data streams. Although the result scores do not change once computed in the scenarios of Refs. [18], [19], [20], database updates in our context can change the relevance scores of existing results. Thus, a more sophisticated approach for incremental view updates is required.

3. Preliminaries

In this section, we introduce some important concepts for keyword querying evaluation in relational databases.

3.1 Relational Database Model

We consider a relational database schema as a directed graph \( G_3(V, E) \), called a schema graph, where \( V \) represents the set of relation schemas \( \{ R_1, R_2, \ldots \} \) and \( E \) represents the foreign key references between pairs of relation schemas. Given two relation schemas, \( R_i \) and \( R_j \), there exists an edge in the schema graph, from \( R_j \) to \( R_i \), denoted \( R_i \leftarrow R_j \), if the primary key of \( R_i \) is referenced by the foreign key defined on \( R_j \). For example, the schema graph of the publication database in Fig. 1 is Papers \( \rightarrow \) Write \( \rightarrow \) Authors. A relation on relation schema \( R_i \) is an instance of \( R_i \) (a set of tuples) conforming to the schema, denoted \( r(R_i) \). A tuple can be inserted into a relation. Below, we use \( R_i \) to denote \( r(R_i) \) if the context is obvious.

3.2 Joint-Tuple-Trees (JTTs)

The results of keyword queries in relational databases are a set of connected trees of tuples, each of which is called a joint-tuple-tree (JTT for short). A JTT represents how the matched tuples, which contain the specified keywords in their text attributes, are interconnected through foreign key references. Two adjacent tuples of a JTT, \( t_i \in r(R_i) \) and \( t_j \in r(R_j) \), are interrelated if they can be joined based on a foreign key reference defined on relational schema \( R_i \) and \( R_j \) in \( G_3 \) (either \( R_i \leftarrow R_j \) or \( R_j \leftarrow R_i \)). To be a valid result of a keyword query \( Q \), each leaf of a JTT is required to contain at least one keyword of \( Q \). The number of tuples in a JTT \( T \) is called the size of \( T \), denoted by \( size(T) \).

Example 2 In Fig. 1 (c), tuples \( p_1, p_2, a_1 \) and \( a_2 \) are matched tuples to the keyword query as they contain the keywords. Hence, the four JTTs are valid results to the query. In contrast, \( p_1 \leftarrow w_2 \rightarrow a_2 \) is not a valid result because tuple \( a_2 \) does not contain any required keywords.

3.3 Candidate Networks (CNs)

Given a keyword query \( Q \), the query tuple set \( R_i^Q \) of relation \( R_i \) is defined as \( R_i^Q = \{ t \in r(R_i) \} \mid t \) contains some keywords of \( Q \). For example, the two query tuple sets in Example 1 are \( p^Q_1 = \{ p_1, p_2, p_3 \} \) and \( A^Q = \{ a_1, a_2, a_3 \} \). The free tuple set \( R_i^F \) of a relation \( R_i \) with respect to \( Q \) is defined as the set of tuples that do not contain any keywords of \( Q \). In Example 1, \( p^F = \{ p_3, p_4, \ldots \} \), \( A^F = \{ a_2, a_4, \ldots \} \). If a relation \( R_i \) does not contain text attributes (e.g., relation \( W \) in Fig. 1), \( R_i \) is used to denote \( R_i^F \) for any keyword query. We use \( R_i^{Q^F} \) to denote a tuple set, which may be either \( R_i^Q \) or \( R_i^F \).

Each JTT belongs to the result of a relational algebra expression, which is called a candidate network (CN) [8], [10], [21]. A CN is obtained by replacing each tuple in a JTT with the corresponding tuple set that it belongs to. Hence, a CN corresponds to a join expression on tuple sets that produces JTTs as results, where each join clause \( R_i^{Q^F} \bowtie R_j^{Q^F} \) corresponds to an edge \((R_i, R_j)\) in the schema graph \( G_3 \), where \( \bowtie \) represents a equijoin between relations. For example, the CNs that correspond to two JTTs \( p_2 \) and \( p_2 \leftarrow w_1 \rightarrow a_1 \) in Example 1 are \( p^Q \) and \( p^Q \bowtie W \bowtie A^Q \), respectively. In the following, we also denote \( p^Q \bowtie W \bowtie A^Q \) as \( p^Q \bowtie W \rightarrow A^Q \). As the leaf nodes of JTTs must be matched tuples, the leaf nodes of CNs must be query tuple sets. Due to the existence of \( m \) relations (for example, an article may be written by multiple authors), a CN may have multiple occurrences of the same tuple set. We use \( R_i^{Q^F} \) to denote the \( x \)-th occurrence of \( R_i^{Q^F} \). If there is only one occurrence of a tuple set, we omit the occurrence number \( x \). The size of CN \( C \), denoted as \( size(C) \), is defined as the number of tuple sets that it contains. Obviously, the size of a CN is the same as that of the JTTs it produces. Figure 2 shows the CNs corresponding to the four JTTs shown in Fig. 1 (c).

A CN can be easily transformed into an equivalent SQL state-
ment and executed by an RDBMS\(^2\).

### 3.4 Scoring Method

The problem of continual top-\(k\) keyword search we study in this paper is to continually report top-\(k\) JTTs based on a certain scoring function that will be described below. We adopt the scoring method employed in Ref.\,[8], which is an ordinary ranking strategy in the information retrieval area. The following function \(score(T, Q)\) is used to score JTT \(T\) for query \(Q\), which is based on the TF-IDF weighting scheme:

\[
score(T, Q) = \frac{\sum_{w|t\in Q} t\text{score}(t, Q)}{\text{size}(T)},
\]

where \(t \in T\) is a tuple (a node) contained in \(T\), \(t\text{score}(t, Q)\) is the tuple score of \(t\) with regard to \(Q\) defined as follows:

\[
t\text{score}(t, Q) = \sum_{w|t\in Q} \frac{1 + \ln (1 + \ln tf_w)}{1 + \ln s} \cdot \ln \frac{N}{d_{fw} + 1}
\]

where \(tf_w\) is the term frequency of keyword \(w\) in tuple \(t\), \(d_{fw}\) is the number of tuples in relation \(r(t)\) (the relation of tuple \(t\)) that contain \(w\). \(d_{fw}\) is interpreted as the document frequency of \(w\). \(dl_t\) represents the size of tuple \(t\), i.e., the number of letters in \(t\), and is interpreted as the document length of \(t\). \(N\) is the total number of tuples in \(r(t)\), \(asdl\) is the average tuple size (average document length) in \(r(t)\), and \(s (0 \leq s \leq 1)\) is a constant which is usually set to 0.2.

Table 1 shows the tuple scores of the six matched tuples in Example 1. Suppose all the matched tuples are shown in Fig. 1, and the numbers of tuples of the two relations are 150 and 180, respectively. Therefore, the top-3 results are \(T_1 = p_2\) (score = 6.48), \(T_2 = a_1\) (score = 4.35) and \(T_3 = p_2 \leftarrow w_2 \rightarrow a_1\) (score = 3.61).

### 4. Query Processing Framework

#### 4.1 Overview

Figure 3 shows the framework of continual keyword query processing in a relational database. The core of the framework is the continual query engine (CQE), which is built on the underlying RDBMS. The process of continual keyword query processing is summarized as follows:

1. The user issues a continual keyword query \(Q\) (a set of keywords) and specifies a \(k\) value to the processing engine.
2. The engine evaluates the initial top-\(k\) results and calculates the state of \(Q\). The top-\(k\) results are sent to the user and the state of \(Q\) is saved for continual execution.
3. After the underlying database is updated, the engine calculates new top-\(k\) results for \(Q\) and deletes the expired results. If the top-\(k\) results of \(Q\) are changed, the engine sends a message for updating the former results.
4. When the user decides to stop the query, the engine clears up its state.

In the steps above, Step 3 (the result maintenance step) is the most important one. It is very time-consuming if we calculate the new top-\(k\) results from scratch. Note that an update in the underlying database may not impose any changes to the current top-\(k\) results. Hence, a smart strategy of query processing with minimum cost is desirable.

In the next subsection, we briefly describe how to generate the set of CNs for a keyword query, which is the basis of the steps above. In Section 5, we present the continual keyword query processing approach in detail.

#### 4.2 Generation of Candidate Networks

When a continual keyword query \(Q = \{w_1, w_2, \ldots, w_l\}\) is specified, CQE first generates the non-empty query tuple set \(RQ_i^Q\) for each relation \(R_i\) in the target database using full-text indices. Then all the non-empty query tuple sets and the database schema graph are utilized to generate the set of valid CNs, whose basic idea is to expand each partial CN by adding a \(R_i^0\) or \(R_j^0\) at each step (\(R_i\) is adjacent to one relation in the partial CN in \(GS_k\)), beginning from the set of non-empty query tuple sets. Hence, the CNs are generated in increasing order of size and shall be sound, complete and duplicate-free. In the implementation, we adopt the state-of-the-art CN generation algorithm proposed in Ref.\,[22].

**Example 3** In Example 1, there are two non-empty query tuple sets \(PQ^Q\) and \(A^Q\). Using them and the database schema graph, the CNs are generated as: \(CN_1 = P^Q, CN_2 = A^Q, CN_3 = P^Q \leftarrow W \rightarrow A^Q, CN_4 = P^1 \leftarrow W^1 \rightarrow A^0 \leftarrow W^2 \rightarrow P^{2,0}, CN_5 = P^2 \leftarrow W^1 \rightarrow A^0 \leftarrow W^2 \rightarrow P^{2,0}, CN_6 = A^1 \leftarrow W^1 \rightarrow P^0 \leftarrow W^2 \rightarrow A^{2,0}, CN_7 = A^1 \leftarrow W^1 \rightarrow P^0 \leftarrow W^2 \rightarrow A^{2,0}, etc.\)

Then, the generated CNs of size smaller than a certain number are evaluated to obtain the top-\(k\) results. The detail is given in the next section. After the evaluation of the CNs, the state of each query is saved in the server for the future use. Note that new non-empty query tuple sets may be produced in the following result maintenance step; hence new CNs need to be generated. In order to achieve high efficiency in maintaining the top-\(k\) results, we generate a set of CNs in a pre-processing step by assuming that

\(\frac{\text{cost}}{\text{length}}\) for example, we can transform CN \(P^Q \leftarrow W \rightarrow A^0\) as follows using the full-text search facility of MySQL: SELECT * FROM W w, P p, A a WHERE w.pid = p.pid AND W.aid = a.aid AND MATCH(title, name) AGAINST (‘James P2P’).
5. Evaluation of Continual Keyword Queries

5.1 Two Effects of Database Updates

Database updates bring two orthogonal effects on the current top-k results:

1. They change the values of \(d_{ref}, N\), and \(awdl\) in Eq. (2) and hence change the relevance scores of existing results.

2. New JTTs may be generated due to insertions. Existing top-k results may be expired due to deletions.

Although the second effect is more drastic, the first one is not negligible for long-term incremental updates. For example, assume now that 120 new non-matched tuples are inserted into relation \(A\), which changes \(A^0.N\) to 300. Thus, the tuple scores of \(a_1, a_3\), and \(a_5\) are changed to 4.94, 3.68 and 3.98, respectively. And the relevance score of \(p_2 \leftarrow w_2 \rightarrow a_1\) is changed to 3.80. Therefore, the top-3 results are changed to \(p_2, a_1, a_5\).

The method proposed in this paper efficiently maintains top-k results by considering the following characteristics. First, merely a small portion of existing JTTs would potentially enter the top-k results because the amount of changes of their relevance scores is limited. Since such JTTs are identified by the algorithm described in Section 5.2, we can avoid computing and storing a large number of JTTs. Second, it is rare for a database update to change the top-k result. Since the effect of each database update can be efficiently calculated by the methods shown in Section 5.3 and Section 5.4, we can achieve considerable reduction in running time.

5.2 State Calculation

Algorithm 1 outlines our algorithm to calculate the state for a continual top-k keyword query, which uses a two-phase CN evaluation method to evaluate the set of generated CNs. The first phase (lines 1–2) employs an existing algorithm, for example, the Global Pipelined Algorithm [8] explained in Section 5.2.1, to find the top-k results. Then the second phase (procedure GetPotentialResults) finds the potential top-k results based on the ranges of relevance score of JTTs, which are computed by the method explained in Section 5.2.2. The procedure is described in Section 5.2.3.

5.2.1 Finding Top-k Results

In Refs. [8] and [10], several algorithms are proposed to efficiently find the top-k results of keyword queries. The aim of all the algorithms is to find a proper order for generating JTTs in order to stop early before all JTTs are generated [23]. Since Ref. [10] adopts a different ranking function, we focus on the two algorithms proposed in Ref. [8], namely, the Sparse algorithm and the Global Pipelined (GP) algorithm. The Sparse algorithm computes an upper-bound score for each CN, which bounds the maximum score of JTTs of the CN. The CNs with high upper-bound scores are fully evaluated, while the CNs with low upper-bound scores are not evaluated if their bounds are not larger than the relevance score of \(k\)-th largest found result. The GP algorithm calculates the top-k results from each query tuple set \(prefix\), which is the subset of tuples that have the highest relevance scores, for estimating the future scores in a pipelined way [14], thus each CN can avoid being fully evaluated. The following discussion assumes the use of the GP algorithm. However, our method works well when the Sparse or other algorithm is employed to find the top-k results.

Figure 4 presents the process of finding the top-3 results using the GP algorithm. In order to simplify the presentation, we suppose only the three CNs generated in Example 3 of size no more than three are evaluated. The left part of Fig. 4 shows the main data structure for evaluating them. Their tuple scores are in Table 1. Tuples in each query tuple set are sorted in non-increasing order according to their tuple scores. The upper-bound score of a CN, denoted as \(\text{MPFS} \[8\], is the maximum relevance score of the hypothetical JTTs in the future evaluation of the CN. The \(\text{MPFS}\) value of \(CN_1\), for example, is computed as \((p_2 \cdot \text{score} + a_1\cdot \text{score})/3 = 3.61\), because among the hypothetical JTTs in its future evaluation, \(p_2\) joined with \(a_1\) has the maximum relevance score. The right part of Fig. 4 is obtained after finding the top-3 results. We define the prefix of each \(R^i_r\) in a CN, which are joined with the other prefixes of the CN to find valid JTTs, as...
the checked tuples of $R_{i}^{Q}$. The three JTTs $T_{1} = p_{2}$, $T_{2} = a_{1}$, and
$T_{3} = p_{2} \leftarrow w_{2} \rightarrow a_{1}$ are identified as the top-3 results because their relevance scores are not smaller than all of the three current MPFS values. The current MPFS of CN$_{1}$ is computed as \( (p_{2}, tscore + a_{1}, tscore) / 3 \) = 3.33, because among the hypothetical JTTs in its future evaluation, $p_{2}$ joined with $a_{1}$ has the highest relevance score. For the details, please refer to Ref. [8].

5.2.2 Computing the Range of Relevance Scores

Let us recall the function for computing tuple scores given in Eq. (2):

\[
tscore(t, Q) = \sum_{w \in \partial Q} \frac{1 + \ln(1 + \ln(f_{sw}))}{1 - s + s \cdot \frac{df_{w}}{df_{w} + 1}} \cdot \ln \left( \frac{N}{df_{w} + 1} \right).
\]

For each relation, we assume that (a) all tuples at most $N \cdot \Delta N$ tuples are inserted or deleted, and (b) the document frequencies change slightly due to the insertion. $\Delta df_{w}$ is used to denote the maximum increased or decreased ratio of the document frequency for keyword $w$. Note that the $\Delta df_{w}$ values of different $w$ can be different. We further assume that the average document length (adll) is a constant to simplify computation. This assumption is reasonable because the changes to adll in a large database are negligible. For example, in the database used in the experimental section, the maximum varying degree of adll is smaller than 0.5%. Thus, the resulting varying degree of the relevance scores is smaller than 0.1% (s = 0.5%).

Then, for the future value of each $\ln \left( \frac{N}{df_{w} + 1} \right)$, we can derive an upper bound $ln^{\max} \left( \frac{N}{df_{w} + 1} \right)$ and a lower bound $ln^{\min} \left( \frac{N}{df_{w} + 1} \right)$ as:

\[
ln^{\max} \left( \frac{N}{df_{w} + 1} \right) = \ln \left( \frac{N(1 + \Delta N)}{df_{w}(1 - \Delta df_{w}) + 1} \right)
\]

and

\[
ln^{\min} \left( \frac{N}{df_{w} + 1} \right) = \ln \left( \frac{N(1 - \Delta N)}{df_{w}(1 + \Delta df_{w}) + 1} \right)
\]

Thus, the upper bound and lower bound of the relevance score for tuple $t$, denoted as $tscore^{\max}$ and $tscore^{\min}$, are computed by replacing $\ln \left( \frac{N}{df_{w} + 1} \right)$ of Eq. (2) with $ln^{\max} \left( \frac{N}{df_{w} + 1} \right)$ and $ln^{\min} \left( \frac{N}{df_{w} + 1} \right)$, respectively.

Using the tuple score bounds above, the range of relevance score of a JTT $T$ is:

\[
\left[ \sum_{t \in T} tscore^{\min}, \sum_{t \in T} tscore^{\max} \right] \cdot \frac{1}{\text{size}(T)}
\]

We use $Tscore^{\max}$ and $Tscore^{\min}$ to denote the upper bound and lower bound of $Tscore$, respectively.

Example 4 By setting $\Delta N = 1 / 10$ and $\Delta df_{w} = 1 / 10$, the ranges of the relevance score for the six matched tuples are shown in Table 2. Hence, $T_{1}.score \in [6.16, 6.79]$, $T_{2}.score \in [4.15, 4.55]$ and $T_{3}.score \in [3.44, 3.78]$.

5.2.3 Finding the Potential Top-k Results

A JTT is a potential top-k result of a query if its $score^{\max}$ value is higher than the minimal $score^{\min}$ of the initial top-k results, which is saved as the variable $\theta$. Note that we do not need to find all the valid JTTs and then check whether their $score^{\max}$ values are larger than $\theta$. In contrast, the potential top-k results are computed by joining all the tuples in each CN that have the possibilities to form JTTs with $score^{\max} > \theta$. For each tuple $t$ of $R_{i}^{Q}$ in CN $C$, we calculate the maximum $score^{\max}$ of the possible JTTs of $C$ that contain $t$:

\[
MaxScore(t, R_{i}^{Q}, C) = \left( tscore^{\max} + \sum_{R_{j}^{Q} \in C} \Delta R_{j}^{Q}, \max_{\text{score}(R_{j}^{Q})} \right) \text{size}(C)
\]

where $\max_{\text{score}(R_{j}^{Q})}$ indicates the maximum $tscore^{\max}$ of tuples in $R_{j}^{Q}$. If $MaxScore(t, R_{i}^{Q}, C)$ is larger than $\theta$, $t$ has the possibility to form potential top-k results, and thus $t$ needs to be joined. Notice that $MaxScore(t, R_{i}^{Q}, C)$ is an overestimate because there is no guarantee that $t$ can join the tuple with maximum $tscore^{\max}$ in each $R_{j}^{Q} \neq R_{i}^{Q}$. However, $MaxScore(t, R_{i}^{Q}, C)$ is the best estimate that we can produce efficiently without accessing the database. A tuple $t$ has equal $MaxScore(t, R_{i}^{Q}, C)$ values for each occurrence of $R_{j}^{Q}$ in $C$; hence they are denoted uniformly using $MaxScore(t, C)$.

Algorithm 2 first identifies the potential top-k results from $JTT_{list}$ (line 2). Then for each non-checked tuple $t$ of $MaxScore(t, C) > \theta$ in every CN $C$, it executes the following two operations:

1. Join $t$ with checked tuples in other query tuple sets of $C$ to find the JTTs with $score^{\max} > \theta$ (line 6).
2. Mark $t$ as checked (line 7).

Example 5 For the snapshot shown in the right part of Fig. 4, $\theta = 3.44$ since the $score^{\max}$ values of $T_{1}$, $T_{2}$ and $T_{3}$ are 6.16, 4.15, and 3.44 (the tuple score ranges are shown in Table 2), respectively. For every non-checked tuple $t$ of CN$_{1}$ and CN$_{2}$, $MaxScore(t, CN_{1}) = MaxScore(t, CN_{2}) = tscore^{\max}$. Thus, we get two potential top-3 results $a_{3}$ and $p_{5}$. For the four non-checked tuples in CN$_{3}$, $MaxScore(p_{1}, CN_{3}) = 2.65$, $Max(p_{5}, CN_{3}) = 2.67$, $MaxScore(a_{3}, CN_{3}) = 3.39$ and $Max(a_{5}, CN_{3}) = 3.49$. Therefore, only $\{a_{3}, p_{5}\}$ is calculated but no JTTs are found. Figure 5 shows the status of the three CNs after finding the potential top-3 results.

5.2.4 Data in States

After the two-phase CN evaluation, the state for each query is created in line 5 of Algorithm 1. The state can be saved to the database or in the main memory. It contains three kinds of data:

- The keyword statistics: the $N$ and $df_{w}$ values of each relation.
- The CNs and the checked tuples in each CN $C$.
- The top-k and the potential top-k results as well as the variable $\theta$.

The checked tuples, i.e., tuples of $MaxScore(t, C) > \theta$, are highly related; hence they have a high probability to form valid JTTs with the inserted tuples. Moreover, they must be joined for

### Table 2 Ranges of the tuple scores for the six matched tuples.

<table>
<thead>
<tr>
<th>Tuple</th>
<th>$a_{1}$</th>
<th>$a_{3}$</th>
<th>$a_{5}$</th>
<th>$p_{1}$</th>
<th>$p_{3}$</th>
<th>$p_{5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tscore^{\min}$</td>
<td>4.15</td>
<td>3.09</td>
<td>3.35</td>
<td>3.08</td>
<td>6.16</td>
<td>3.13</td>
</tr>
<tr>
<td>$tscore^{\max}$</td>
<td>4.53</td>
<td>3.39</td>
<td>3.67</td>
<td>3.39</td>
<td>6.79</td>
<td>3.45</td>
</tr>
</tbody>
</table>

---

*For presentation purposes, lines 4–7 are simplified. In the real implementation, we use set-based processing for efficiency reasons.*
finding the new JTTs with $score^{\text{max}} > \theta$ in the top-$k$ result maintenance step. Hence, they are saved to accelerate the computation of new results. Note that we only need to save the tuple ID for each checked tuple.

5.3 Handling Insertions of Tuples

Algorithm 3 outlines our algorithm for handling new tuples. After receiving a new tuple $t_{\text{new}}$ of relation $R_i$, we first check whether each $\ln \left( \frac{N}{|\Delta N|} \right)$ value of $R_i$ exceeds its two bounds (line 2). This is very easy: monitoring $\Delta N$ is straightforward; $\Delta f_w$ for all the terms $w$ can be accumulated in the process of handling database updates. If any $\ln \left( \frac{N}{|\Delta N|} \right)$ value exceeds its two bounds, the keyword query is re-evaluated (line 3); otherwise the two effects mentioned in Section 5.1 are calculated. First, the relevance scores of the saved results are updated in line 5. Second, if $t_{\text{new}}$ is a matched tuple for $Q$, data in the query state is updated in lines 6–7. In lines 6–7, the new CNs are added if $R_i^0$ is a new non-empty query tuple set.

5.3.1 Finding New Results

Procedure FindNewJTTs (see Algorithm 4) first constructs the set of CNs to be evaluated in lines 1 to 8. For finding all the JTTs involving $t_{\text{new}}$ with $score^{\text{max}} > \theta$, $t_{\text{new}}$ needs to be joined with the checked tuples in each CN. Thus, $R_i^{Q|\text{CNSet}}$ of the CNs in $S$.CNSet are replaced with $\{t_{\text{new}}\}$, where $R_i^{Q|\text{CNSet}}$ is $R_i^0$ if $t_{\text{new}}$ is a matched tuple or $R_i^0$ if $t_{\text{new}}$ is a non-matched tuple. If a CN contains $n R_i^{Q|\text{CNSet}}$ occurrences, it is added into $CS$ $n$ times in the loop (line 4); each iteration replaces an occurrence of $R_i^{Q|\text{CNSet}}$ with $\{t_{\text{new}}\}$. Then, if $t_{\text{new}}$ is a matched tuple, the CNs with $MaxScore(t_{\text{new}}, C) \leq S \theta$ are pruned because they cannot produce JTTs involving $t_{\text{new}}$ with $score^{\text{max}} > S \theta$; even if $MaxScore(t_{\text{new}}, C) > S \theta$, not all the checked tuples in $C$ can form JTTs with $score^{\text{max}} > S \theta$. Hence, the checked tuples with $MaxScore(t, C) \leq S \theta$ are deleted in line 8, where $MaxScore(t, C)$ is computed with $max_score(R_i) = t_{\text{new}}, t_{\text{score}}^{\text{max}}$. Note that if $t_{\text{new}}$ is a non-matched tuple, no pruning based on $MaxScore$ values are possible.

Example 6 Suppose the seven CNs shown in Example 3 form the initial state. Then for a new tuple $t_{\text{new}}$ of relation $W$, ...
would produce the same results involving instantiations of CN. Note that the duplicated CNs are excluded. For instance, the two example, the subexpression common subexpressions due to how they are generated [7]. For the key optimization opportunity is that the CNs share many intermediate results which are not output eventually. For instance, the two Table 3 Process of DISCOVER for evaluating the CNs in Example 6. 

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Subexpression</th>
<th>CNs</th>
<th>Iteration</th>
<th>Subexpression</th>
<th>CNs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(t_{\text{new}} \models P^0)</td>
<td>(C_1, C_2, C_3, C_4)</td>
<td>4</td>
<td>(W \models P^2)</td>
<td>(C_2, C_3, C_4)</td>
</tr>
<tr>
<td>2</td>
<td>(t_{\text{new}} \models A^0)</td>
<td>(C_1, C_2, C_3, C_4)</td>
<td>5</td>
<td>(t_{\text{new}} \models A^1 \land W)</td>
<td>(C_1)</td>
</tr>
<tr>
<td>3</td>
<td>(W \models A^1)</td>
<td>(C_2, C_3, C_4)</td>
<td>6</td>
<td>(t_{\text{new}} \models P^2 \land W)</td>
<td>(C_1)</td>
</tr>
</tbody>
</table>

the CNs in \(CS\) are \(CS = \{C_1, C_2, C_3, C_4, C_5\}\), where \(C_1 = P^0 \leftarrow \{t_{\text{new}}\} \rightarrow A^0, C_2 = P^1 \leftarrow \{t_{\text{new}}\} \rightarrow A^0 \leftarrow W^2 \rightarrow P^2\), \(C_3 = P^1 \leftarrow \{t_{\text{new}}\} \rightarrow A^0 \leftarrow W^2 \rightarrow P^2\), \(C_4 = A^1 \leftarrow \{t_{\text{new}}\} \rightarrow P^2 \leftarrow W^2 \rightarrow A^0\). Note that the duplicated CNs are excluded. For instance, the two instantiations of \(CN_4\) \(P^1 \leftarrow \{t_{\text{new}}\} \rightarrow A^0 \leftarrow W^2 \rightarrow P^2\) and \(P^1 \leftarrow W^1 \rightarrow A^0 \leftarrow \{t_{\text{new}}\} \rightarrow P^2\) are duplicated because they would produce the same results involving \(t_{\text{new}}\).

Full evaluation of the CNs in \(CS\) individually is rather costly. The key optimization opportunity is that the CNs share many common subexpressions due to how they are generated [7]. For example, the subexpression \(P^0 \models \{t_{\text{new}}\}\) is contained in the first four CNs computed in Example 6. Efficient execution plans store the common join expressions as intermediate results and reuse them in evaluating the CNs. DISCOVER [7] shows that the selection of the optimal execution plan (i.e., how to reuse common subexpressions) is NP-complete and proposes a greedy algorithm, which selects the common subexpression with the maximum frequency in the evaluated CNs to execute in each iteration. Its process for evaluating the five CNs in Example 6 is shown in Table 3. Table 3 may compute a large number of unnecessary intermediate results which are not output eventually. For instance, the intermediate results of \(W \models A^1\) and \(W \models P^0\) can be large in size. However, such results are seldom can be involved in the final JTTs output because the probability that they are joined with \(t_{\text{new}}\) is small.

In order to overcome the above drawback of DISCOVER, procedure FindNewJTTs constructs a rooted tree \(T\) for the CNs in \(CS\) (line 9). Each CN \(C_i\) is firstly modeled as a rooted tree rooted at \(t_{\text{new}}\), whose nodes are labeled by \(i\). For example, the rooted tree corresponding to \(C_2 = P^1 \leftarrow \{t_{\text{new}}\} \rightarrow A^0 \leftarrow W^2 \rightarrow P^2\) has two rooted subtrees: \(P^0\) and \(A^0 - W - P^0\). Then the rooted trees of all the CNs merge their roots to form the rooted tree \(T\). Figure 6 shows the rooted tree \(T\) for the CNs in Example 6, which shows the labels of nodes in their left. Each edge \(\{R^i_{\alpha \omega F}, R^j_{\beta \alpha F}\}\) in \(T\) represents a join operation \(R^i_{\alpha \omega F} \leftarrow R^j_{\beta \alpha F}\) in a CN, which joins tuples of the father node \(R^i_{\alpha \omega F}\) to tuples of the child node \(R^j_{\beta \alpha F}\). Thus, the joins of the CNs get the father-child relationship induced by the father-child relationship in \(T\). Below, we refer to the edges in \(T\) as joins if the context is obvious. Two joins in each level are similar if their father nodes and child nodes are tuple sets of the same relations, respectively. The similarity relation partitions joins at each level. The sets of similar joins in each level are indicated by rectangles in Fig. 6.

All the joins in the rooted tree \(T\) are evaluated in a top-to-bottom, breadth-first fashion. Thus, all the found intermediate results are joined with \(t_{\text{new}}\). Each set of similar joins in every level are evaluated by a common join operation, which joins the union of tuples in their father nodes \(\cup R^i_{\alpha \omega F}\) with the union of tuples in their child nodes \(\cup R^j_{\beta \alpha F}\) (lines 11–12). Thus, the result of the common join operation, indicated by \(U\) in line 12, contains the

### Algorithm 4: FindNewJTTs \(x(S, t_{\text{new}}, R_i)\)

1. \(CS \leftarrow \emptyset;\) // the set of new CNs to be evaluated
2. \(R^i_{\alpha \omega F} \leftarrow\) the tuple set corresponding to \(t_{\text{new}}\);
3. foreach CN \(C\) in \(S\) \{CNSet\} that has checked tuples do
   4. foreach occurrence \(R^i_{\alpha \omega F} \rightarrow C\) do
      5. Add \(C\) into \(CS\) after replacing \(R^i_{\alpha \omega F} \leftarrow \{t_{\text{new}}\}\);
   6. if \(t_{\text{new}}\) is a matched tuple then // Prune CNs and checked tuples
      7. Delete each \(C\) in \(CS\) with \(\text{MaxScore}(t_{\text{new}}, C) \leq S.\theta\);
      8. Delete each tuple \(t\) with \(\text{MaxScore}(t, C) \leq \text{from each } C \in CS;\)
   9. Construct a rooted tree \(T\) for the CNs in \(CS;\)
      10. // Evaluate all the joins in \(T\) in a breadth-first fashion
      11. for \(i = 1\) to \(n\) // \(n\) is the maximum level of edges in \(T\)
          12. foreach set of similar joins \(\{R^i_{\alpha \omega F} \leftarrow R^j_{\beta \alpha F}\}\) in level \(i\) of \(T\) do
              13. \(U \leftarrow\) results of evaluating \(\{\{R^i_{\alpha \omega F} \leftarrow R^j_{\beta \alpha F}\};\) // partial JTTs
              14. foreach join \(J\) in \(\{R^i_{\alpha \omega F} \leftarrow R^j_{\beta \alpha F}\}\) do
                  15. if can find results of \(J\) in \(U\) then
                      16. Save the results of \(J\) and delete the non-involved tuples from \(J\)’s two nodes;
                  else // \(J\) has no results
                      17. Delete the nodes with the label of \(J\)’s nodes from \(T;\)
              18. Construct JTTs using the results of the joins in \(T\) and add those with \(\text{score}_{\text{new}} > S.\theta\) into \(S;\)

The CNs in \(CS\) are \(CS = \{C_1, C_2, C_3, C_4, C_5\}\), where \(C_1 = P^0 \leftarrow \{t_{\text{new}}\} \rightarrow A^0, C_2 = P^1 \leftarrow \{t_{\text{new}}\} \rightarrow A^0 \leftarrow W^2 \rightarrow P^2\), \(C_3 = P^1 \leftarrow \{t_{\text{new}}\} \rightarrow A^0 \leftarrow W^2 \rightarrow P^2\), \(C_4 = A^1 \leftarrow \{t_{\text{new}}\} \rightarrow P^2 \leftarrow W^2 \rightarrow A^0\). Note that the duplicated CNs are excluded. For instance, the two instantiations of \(CN_4\) \(P^1 \leftarrow \{t_{\text{new}}\} \rightarrow A^0 \leftarrow W^2 \rightarrow P^2\) and \(P^1 \leftarrow W^1 \rightarrow A^0 \leftarrow \{t_{\text{new}}\} \rightarrow P^2\) are duplicated because they would produce the same results involving \(t_{\text{new}}\).
results of each join in \(R^Q_i \cup F \Rightarrow R^Q_i \cup F\). Note that, the checked tuples of \(R^Q_i \cup F\) and \(R^Q_i \cup F\) in different joins may be different. From \(U\), each join \(J\) in \(\{R^Q_i \cup F \Rightarrow R^Q_i \cup F\\}\) finds its results using the indexes on the tuples of its father and child nodes, and then deletes the tuples in its two nodes which are not involved in the results (line 15). All the joins in Fig. 6 are evaluated by the six common join operations shown in it, where \(\mathbf{A}, \mathbf{P}\) and \(\mathbf{W}\) indicate that they only contain the tuples that can join \(\mathbf{t}_{\text{new}}\). For example, \(\mathbf{A}\) is the union of tuples that can join \(\mathbf{t}_{\text{new}}\) in \(A^\ell\) labeled by 2 and \(A^\ell\) label by 3. If a join has no results, all the nodes of the CN containing it are deleted from \(T\) (lines 16–17). After evaluating all the joins, if a CN \(C_i\) is not deleted from \(T\), its JTTs involving \(\mathbf{t}_{\text{new}}\) are constructed using the results of the joins of labeled nodes \(l_i\).

### 5.3.2 Complexity Analysis

If the matched tuples are distributed evenly, the \(\ln\left(\frac{N}{\pi_{\text{new}}}\right)\) values will vary in small degrees. Even in the worst case\(^4\) (i.e., no matched tuples in the new tuples), moderate values of \(\Delta N\) and \(\Delta f_{\text{new}}\) of around 0.1 will rarely cause query re-evaluation. The first upper bound violation occurs when \(\ln\left(\frac{N_{\text{new}}}{\pi_{\text{new}}}\right) = \ln\left(\frac{N_{\text{new}}}{\pi_{\text{new}}}\right) = \ln\left(\frac{N_{\text{new}}}{\pi_{\text{new}}}\right)\), i.e., \(N_{\text{new}} \approx N \left(\frac{\Delta N}{1 + \Delta f_{\text{new}}}\right)^{1}\), where \(N_{\text{new}}\) indicates the new number of tuples. Hence the \(\Delta\)th upper bound violation occurs when \(N_{\text{new}} = N \left(\frac{\Delta N}{1 + \Delta f_{\text{new}}}\right)^{\Delta}\). Thus, before the database is doubled, there are less than four times (log\(1.1/0.9\) 2 < 4). Similar conclusions can be drawn if deletions are considered. Therefore, we can omit query re-evaluations when discussing the time complexity.

A new matched tuple \(\mathbf{t}_{\text{new}}\) is pruned for CN \(C\) if \(\text{MaxScore}(t_{\text{new}}, C) < S\theta\). The probability \(\Pr(\text{MaxScore}(t_{\text{new}}, C) > S\theta)\) can be approximated by \(\frac{\#C}{\#M}\), where \#\(C\) is the number of checked tuples of \(R^Q_i\) and \#\(M\) \(\approx |R^Q_i|\). In our experiments, the common value of \(\frac{\#C}{\#M}\) is smaller than 0.3. Hence, roughly speaking, 70\% matched tuples are pruned for each CN \(C\). Lines 11–12 of Algorithm 3, which need to query non-checked tuples from the database, show negligibly low execution frequency in our experiment, because it is very rare that \(t_{\text{score}}\) of \(t_{\text{new}}\) is larger than those of all the tuples in \(R^Q_i\).

We use \(M_1\) and \(M_2\) to indicate the maximum number of adjacent relations that a relation \(R_i\) can have in the database schema graph \(G_S\) and the maximum number of tuples of a tuple \(t_i\) of \(R_i\) can join in its each adjacent relation, respectively. Note that \(M_1\) is often rather small compared to the number of CNs. Then in the worst case, the number of different sets of similar joins in level \(l\) is \(O(M_1^l)\), and the result size of each join in level \(l\) is \(O(M_2^l)\). The cost to evaluate a join is proportional to the size of its result. The maximum level of joins of the rooted tree \(T\) is \(CN_{\text{max}} - 1\), where \(CN_{\text{max}}\) indicates the maximum size of the CNs. Thus, the maximum cost to evaluate all the joins in \(T\) is:

\[
O\left(\sum_{i=1}^{CN_{\text{max}}-1} (M_1^i \cdot M_2^i)^{CN_{\text{max}}}\right) = O((M_1 \cdot M_2)^{CN_{\text{max}}})
\]

However, as can be seen from the experimental results, the average-case behavior seems to be much lower than the analytical worst-case. This is because the probability that there are checked tuples in a query tuple set \(R^Q_i\) of level \(l\) that can join with \(t_{\text{new}}\) can be approximated as: \(\rho = M_1^l \cdot \frac{\frac{\Delta N}{\pi_{\text{new}}}}{\#C} \cdot \frac{\Delta f_{\text{new}}}{\#M}\), where \(M_1^l\) is the maximum number of tuples in relation \(R_i\) that can join with \(t_{\text{new}}\). \(\frac{\Delta N}{\pi_{\text{new}}}/\#C\) indicates the probability of a tuple being a matched tuple and \(\Delta f_{\text{new}}/\#M\) indicates the probability for a matched tuple being checked. \(\rho\) is small because the \(\frac{\Delta N}{\pi_{\text{new}}}/\#C\) values are often very small (< 0.01). Moreover, since the joins of \(T\) are evaluated in the top-down manner, tuples in \(R^Q_i\) that can join \(t_{\text{new}}\) are required to join some tuples of all the ancestors of \(R^Q_i\). Therefore, for most of the new tuples, every CN in \(CS\) is deleted from \(T\) due to the empty result of its one join.

### 5.4 Discussion on Handling Deletions

Since updates can be modeled as deletions followed by insertions, we only discuss handling of deletions. Deleting a tuple \(t\) from the query state is straightforward: \(t\) is deleted from all the query tuple sets that containing it and all the results involving \(t\) are deleted. However, when handling deletions, beside checking whether any \(\ln\left(\frac{N_{\text{new}}}{\pi_{\text{new}}}\right)\) exceeds its two bounds similar to handling insertions, we also need to check whether the \(k\) results in the state that have the highest relevance scores are still the top-\(k\) results, which can be false due to deletions.

Recall that all the JTTs that are not in the state satisfy \(\text{score}_{\max} \leq \theta\). Therefore, the top-\(k\) results fulfill \(\text{score} > \theta\). After a query state is computed, there are at least \(k\) JTTs with \(\text{score} > \theta\) in it since the initial top-\(k\) results are with \(\text{score}_{\max} \geq \theta\). The number of such JTTs in the state can be decreased due to deletions. If the number of JTTs with \(\text{score} > \theta\) in the query state after handling deletions is smaller than \(k\), we need to re-evaluate the query state. Since the value \(k\) is rather small compared to the huge number of valid JTTs, the possibility of deleting a result with \(\text{score} > \theta\) is rather small. It is worth noting that the set of potential top-\(k\) results also contain results with \(\text{score} > \theta\). In addition new JTTs with \(\text{score} > \theta\) can also be formed by new tuples. Thus, if the insertion rate is not smaller than the deletion rate, the possibility that the number of JTTs with \(\text{score} > \theta\) falling below \(k\) would be very small. However, if the deletion rate is much larger than the insertion rate, the possibility that the number of JTTs with \(\text{score} > \theta\) is decreased is high. Then, maintaining the top-\(k\) results would suffer from frequent query re-evaluations. In such cases, we can use larger \(\Delta N\) and \(\Delta f_{\text{new}}\) values, which will result in broader ranges of relevance scores, hence more results with \(\text{score} > \theta\) in the set of potential top-\(k\) results. However, the experimental results show that even small values of \(\Delta N\) and \(\Delta f_{\text{new}}\) (< 0.10) can achieve very low query re-evaluation frequencies caused by this reason.

### 6. Experimental Study

We conducted extensive experiments to test the efficiency of our methods. We use the DBLP dataset\(^5\). Note that DBLP is not continuously growing and is updated on a monthly basis. The reason we use DBLP to simulate a continuously growing relational dataset is because there is no real growing relational datasets in the public domain, and many studies [8], [10] on top-k

\[^4\] The possibility that all the new tuples are matched tuples is very small because \(\frac{\Delta N}{\pi_{\text{new}}}\) is often very small (< 0.01).

\[^5\] http://dblp.mpi-inf.mpg.de/dblp-mirror/index.php/
keyword queries over relational databases use DBLP. The downloaded XML file is decomposed into relations according to the schema shown in Fig. 7. The two arrows from PaperCite to Papers denote the foreign-key-references from paperID to paperID and citedPaperID to paperID, respectively. The DBMS used is MySQL (v5.1.44) with the default “Dedicated MySQL Server Machine” configuration. All the relations use the MyISAM storage engine. Indexes are built for all primary key and foreign key attributes, and full-text indexes are built for all text attributes. All the algorithms are implemented in C++.

We conducted all the experiments on a 2.53 GHz CPU PC with 4 GB memory running Windows 7.

### 6.1 Parameters

We use the following four parameters in the experiments: (1) $\Delta N$: the maximum varying ratio of $N$ and $df_w$ values; (2) $l$: the number of keywords in a query; (3) $IDF$: the ratio of the number of matched tuples to the number of total tuples, i.e., $\frac{df_w}{N}$; and (4) $CN_{\text{max}}$: the maximum size of the generated CNs. The parameters with their default values (bold) are shown in Table 4. We use a single parameter $N$ to denote the same value of $\Delta N$ and $\Delta df_w$ because the impact of two different $\Delta N$ and $\Delta df_w$ to the ranges of relevance scores can be provided by setting $\Delta N = \Delta df_w = \sqrt{\Delta N^2 - \Delta df_w^2}$. The keywords selected are listed in Table 5 with their $IDF$ values, where the keywords in bold fonts are author names which are popular keywords. Ten queries are constructed for every $IDF$ value, each of which contains three selected keywords. For each $l$ value, ten queries are constructed by selecting $l$ keywords from the row of $IDF = 0.013$ in Table 5. To avoid generating a small number of CNs for each query, one author name keyword of each $IDF$ value is always selected for each query.

Smaller $\Delta N$ values result in less checked tuples and potential top-$k$ results, but have a larger query re-evaluation frequency because $\ln\left(\frac{N'}{df_w}\right)$ will soon exceed their bounds. Therefore, the chosen $\Delta N$ values represent a trade-off among several factors. $CN_{\text{max}}$ has a great impact on keyword query processing because the number of generated CNs increases exponentially while $CN_{\text{max}}$ increases. And the number of matched tuples increases as $IDF$ and $l$ increase. Hence, the latter three parameters $l$, $IDF$ and $CN_{\text{max}}$ affect the scalability of our method.

### 6.2 Exp-1: State Calculation

In this experiment, we want to study the effects of the five parameters on state calculation. We retrieve the data in the XML file sequentially until the number of tuples in the relations reach the numbers shown in Table 6. Then we run the algorithm Calc-PR on different values of each parameter while keeping the other three parameters at their default values. For example, when varying $\Delta N$ from 0.01 to 0.3, the other three parameters are $l = 3$, $IDF = 0.13$ and $CN_{\text{max}} = 6$. We use three measures to evaluate the effects of the parameters. The first is #Check, which indicates the number of checked tuples for all the CNs. The second measure, #PR, is the number of found potential top-$k$ results. The third measure is State, the data size of the query states. The major life-cycle of a continual top-$k$ keyword query corresponds to Step 3 mentioned in Section 4.1. Therefore, we do not report the running time of the state calculation. Ten top-$k$ queries are selected for each combinations of parameters, and the following results are obtained by averaging the metrics used.

The main results of this experiment are given in Fig. 8. Note that the units for the $y$-axis are different for the three measures. Figure 8(a) shows that #Check and #PR grow fast as $\Delta N$ increases because the ranges of relevance scores of the results are enlarged. However, as shown in the next subsection, a small $\Delta N$ (e.g., 0.1) is adequate for the maintenance of the top-$k$ results. The three measures do not show rapidly increase in Fig. 8(b)–(d) while $df_w$, $IDF$ and $CN_{\text{max}}$ are growing, which imply the good scalability of our method. Figure 8 shows that the state sizes are quite small under all settings (1.4 MB at most). The large state size when $CN_{\text{max}} = 7$ is mainly caused by the large number of CNs ($\approx 400$).

From Fig. 8(c), we can see that the #PR values are equal for different $CN_{\text{max}}$ values. This is because the newly added CNs rarely produce top-$k$ results as $CN_{\text{max}}$ increases: (1) the top-$k$ re-
sults are only distributed in a few CNs [24]; and (2) the adopted ranking strategy prefers the JTTs with small sizes. Figure 8 (b) and (d) show that the effects of IDF and l seem more complicated: #Check and #PR are very large when IDF and l have the minimum values 0.003 and 2, respectively. This is because the probability that the keywords co-appear in a JTT is small when they have small values. Thus, all the matched tuples and results have roughly equally small relevance scores, which results in smaller \( l \) values and larger \( \theta \) values. This also explains why \( \frac{\text{Check}}{\text{PR}} \) reaches its two peak values when \( l = 2 \) (0.254) and IDF = 0.003 (0.247), although its values are very small (< 0.1) in most cases.

### 6.3 Exp-2: Top-\( k \) Result Maintenance

Firstly, we want to study the efficiency of Algorithm 3 in handling new tuples. The query states used are the states calculated in Exp-1. We sequentially insert 402,270 additional tuples into the database by retrieving data from the DBLP XML file. The new data is roughly 50 percent of the data used in Exp-1. At the same time, Algorithm 3 is used to maintain the top-50 results for the queries. The composition of the additional tuples is shown in Table 7. The database update records are read from the database log file; hence the new tuples inserting rate has no direct impact on the efficiency of the maintenance of the top-\( k \) results because the database is updated by another process. We also compare the efficiency of procedure \( \text{FindNewJTTs} \) in evaluating the CNs instantiated using new tuples against that of DISCOVER and the naive method that fully evaluates the CNs individually.

The main experimental results are given in Fig. 9. Figure 9 (a) shows the change in average execution times of Algorithm 3 in handling additional tuples while varying the four parameters, which presents the efficiency of Algorithm 3. Note that the units for the \( x \)-axis are different for the four measures, whose minimum and maximum values are labeled in Figure 9 (a), and their other values can be found in Table 4. Comparing Figure 9 (a) with the curves for the measure #Check in Fig. 8 (especially the curves in Fig. 8 (b) and Fig. 8 (d)), we can find that the time to handle a new tuple for a query is mainly affected by the number of checked tuples. This is because more time is required to execute line 12 of procedure \( \text{FindNewJTTs} \) when the number of checked tuples grows.

![Fig. 8](image-url) Effects of parameters in calculating query states (\( k = 50 \)).

<table>
<thead>
<tr>
<th>Papers</th>
<th>PaperCite</th>
<th>Write</th>
<th>Authors</th>
<th>Proceedings</th>
<th>ProEditors</th>
<th>ProcEditor</th>
</tr>
</thead>
<tbody>
<tr>
<td>84,541</td>
<td>9,786</td>
<td>210,154</td>
<td>90,147</td>
<td>2,786</td>
<td>2,091</td>
<td>2,765</td>
</tr>
</tbody>
</table>

Figure 9 (b) shows the average execution times of the three methods for evaluating the CNs constructed for new tuples while varying \( CN_{\text{max}} \). DISCOVER and procedure \( \text{FindNewJTTs} \) achieve great improvement in terms of efficiency when compared to the naive method for reusing the results of common subexpressions. Procedure \( \text{FindNewJTTs} \) outperforms DISCOVER in all cases because: (1) it can avoid computing a large number of unnecessary intermediate results because all the found intermediate results can join with \( t_{\text{new}} \); and (2) evaluating the similar joins by a common join operation can achieve higher efficiency. For example, for evaluating the left five joins in the first level of \( T \) shown in Fig. 6, procedure \( \text{FindNewJTTs} \) needs to evaluate a join operation \( [t_{\text{new}}] \bowtie r(A) \), whereas DISCOVER needs to evaluate two join operations \( [t_{\text{new}}] \bowtie A^F \) and \( [t_{\text{new}}] \bowtie A^Q \). Although the time costs of the three methods all increase while \( CN_{\text{max}} \) increases, our method shows much better scalability. For instance, the increase in time cost is less than 1 ms when \( CN_{\text{max}} \) grows from 6 to 7, despite the fact that the number of CNs grows from 135 to 412. This is because, for most new tuples, only the first one (at most two) level(s) of joins in the rooted tree \( T \) are evaluated. Figure 9 (c) shows the peak memory of the three compared methods. We only
report the memory consumed by the continual query engine. As expected, the naïve method consumes the least memory since it only needs to store the query state. Procedure FindNewJTTs uses a little more memory than the naïve method since it needs to store the rooted trees. DISCOVER uses the largest amount of memory due to the large number of intermediate results.

As explained in Section 5.3.2 and as can be seen in Fig. 9 (a), the actual cost to handle insertions is rather small compared to the upper bound computed by Eq. (6); hence we do not study the effects of the variants in Eq. (6) quantitatively, but only qualitatively. The effect of $CN_{\text{max}}$ can be observed from all the figures in Fig. 9. Then, we manually increase $M_2$ (the number of tuples that a tuple can join with) in Eq. (6) to study its effect. Specifically, the number of tuples that a Papers tuple can join is increased from 2 to 10. Figure 9 (d) shows the changes in time cost of different $CN_{\text{max}}$ values while increasing $M_2$. We can observe the increase in time cost for all the $CN_{\text{max}}$ values. Studying the effect of $M_1$ is future work since the database schema studied in this paper is static.

Lastly, the efficiency of our method in handling deletions is studied. Different from the above discussions, the average time for handling deletions is not listed, since it is very small (≈ 0.3 ms) and has small changes under all settings because database access is not required in these tasks (updating the relevance scores of the saved results and deleting the expired results). We need to be done if the query re-evaluation is not activated. However, deletions can incur the type of query re-evaluations caused by deficiency of top-$k$ results, whose frequencies are shown in Table 8 while varying $\Delta N$, under different insertion and deletion rates. The first row shows the results when no deletions are involved; the second row shows the results when 402,270 tuples are randomly deleted while new tuples are inserted at the same rate; and the third row shows the results when only the 402,270 tuples are deleted. The number in parenthesis in each cell is the number of query re-evaluations triggered by exceeding the $\ln \frac{N}{N_{t+1}}$ ranges in the experiment. Table 8 shows that although the frequencies of query re-evaluations are very large when the deletion rate is not smaller than the insertion rate, we can find fast decreases while $\Delta N$ increases. When $\Delta N = 0.10$, there are less than 10 occurrences of query re-evaluations; hence a small $\Delta N$ is adequate to produce long-lasting query states.

7. Conclusions and Future Work

In this paper, we have studied the problem of finding the top-$k$ results in relational databases for a continual keyword query. We proposed to store the state of each keyword query, which is used to maintain the top-$k$ results after new tuples are inserted into the database. Algorithms for maintaining the top-$k$ results upon insertion of new tuples are developed. The proposed method can efficiently maintain the top-$k$ results of a keyword query without re-evaluation. Therefore, it can be used to solve the problem of answering continual keyword search in databases that are updated frequently.

In the future, we would like to extend the method proposed in this paper to maintain the top-$k$ results for a large number of keyword queries. Currently, the query states are kept in main mem-

![Fig. 9 Efficiency of top-$k$ result maintenance.](image)

<table>
<thead>
<tr>
<th>$\Delta N$</th>
<th>0.01</th>
<th>0.05</th>
<th>0.1</th>
<th>0.15</th>
<th>0.2</th>
<th>0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete=0</td>
<td>1 (96.9)</td>
<td>0 (15.6)</td>
<td>0 (6.0)</td>
<td>0 (3.4)</td>
<td>0 (2.6)</td>
<td>0 (2.2)</td>
</tr>
<tr>
<td>Insert=Delete</td>
<td>405.7 (17.2)</td>
<td>4.1 (15.6)</td>
<td>1.1 (7.8)</td>
<td>1 (5.6)</td>
<td>0.4 (3.6)</td>
<td>0.3 (2.7)</td>
</tr>
<tr>
<td>Insert=0</td>
<td>641.6 (5.7)</td>
<td>35.1 (4.2)</td>
<td>3.6 (4.8)</td>
<td>2.8 (2.8)</td>
<td>1.6 (0.6)</td>
<td>1.4 (0)</td>
</tr>
</tbody>
</table>


Jihong Guan is now a professor of Department of Computer Science & Technology, Tongji University, Shanghai, China. She received her Bachelor degree from Huazhong Normal University in 1991, her Master degree from Wuhan Technical University of Surveying and Mapping (merged into Wuhan University since August 2000) in 1991, and her Ph.D. from Wuhan University in 2002. Before joining Tongji University, she served in Department of Computer, Wuhan Technical University of Surveying and Mapping from 1991 to 1997, as an assistant professor and an associate professor (since August 2000) respectively. She was an associate professor (August 2000–October 2003) and a professor (Since November 2003) in the School of Computer, Wuhan University. Her research interests include databases, data mining, distributed computing, bioinformatics, and geographic information systems. She has published more than 100 research papers in domestic and international journals and conferences.