Cluster Comput (2019) 22:S731-S747
https://doi.org/10.1007/s10586-017-1232-6

# Scalable top-k keyword search in relational databases

**Yanwei Xu**[1]

Received: 15 August 2017 / Revised: 18 September 2017 / Accepted: 25 September 2017 / Published online: 6 October 2017


**Abstract** Keyword search in relational databases has been widely studied in recent years because it does not require users neither to master a certain structured query language nor to know the complex underlying database schemas. There would be a huge number of valid results for a keyword query in a large database. However, only the top 10 or 20 most relevant matches for the keyword query—according to some definition of "Relevance"—are generally of interest. In this paper, we propose an efficient method which can efficiently compute the top-*k* results for keyword queries in a pipelined pattern, by incorporating the ranking mechanisms into the query processing method. Four optimization methods based on bounding the relevance scores of potential results, reusing and sharing the intermediate result are presented to improve the efficiency of the proposed algorithms. Compared to the existing top-*k* keyword search systems, the proposed methods can significantly reduce the number of computed query results with low relevance scores and the times for accessing databases, which result in the high efficiency in computing top-*k* keyword query results in relational databases. Extensive experiments on two real data sets are conducted to evaluate the effectiveness and efficiency of the proposed approach.

**Keywords** Relational databases · Keyword search · Top-k query

✉ Yanwei Xu
13816385269@163.com

[1] Shanghai Engineering Research Center for Broadband Technologies & Applications, 150 Honggu Road, Shanghai 200336, China

## 1 Introduction

With the proliferation of text data available in relational databases, simple ways to exploring 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. Given a $l$-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 [1].

*Example 1* Consider a sample publication database shown in Fig. 1. Figure 1a 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 1b 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. 1a). They can be regarded 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. 1b. The arrows represent the foreign key references between the corresponding pairs of tuples. Finding such results that are formed by the tuples containing the keywords is the task of a keyword search in relational databases. As described later, results are often ranked by relevance scores evaluated by a certain ranking strategy. □

There would be a huge number of valid results for a keyword query in a large database. However, only the top 10 or

**Papers**

| *pid* | *title* |
|---|---|
| $p_1$ | "Leveraging Identity-Based Cryptography for Node ID Assignment in Structured P2P Systems." |
| $p_2$ | "P2P or Not P2P?: In P2P 2003" |
| $p_3$ | "A System for Predicting Subcellular Localization." |
| $p_4$ | "Logical Queries over Views: Decidability." |
| $p_5$ | "A conservative strategy to protect P2P file sharing systems from pollution attacks." |
| ... | ... |

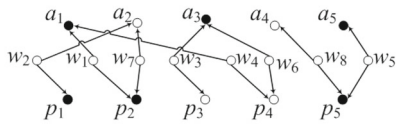
**Authors**

| *aid* | *name* |
|---|---|
| $a_1$ | "James Chen" |
| $a_2$ | "Saikat Guha" |
| $a_3$ | "James Bassingthwaighte" |
| $a_4$ | "Sabu T." |
| $a_5$ | "James S. W. Walkerdines" |
| ... | ... |

**Writes**

| *wid* | *aid* | *pid* |
|---|---|---|
| $w_1$ | $a_1$ | $p_2$ |
| $w_2$ | $a_2$ | $p_1$ |
| $w_3$ | $a_3$ | $p_3$ |
| $w_4$ | $a_1$ | $p_4$ |
| $w_5$ | $a_5$ | $p_5$ |
| $w_6$ | $a_3$ | $p_4$ |
| $w_7$ | $a_2$ | $p_2$ |
| $w_8$ | $a_2$ | $p_5$ |
| ... | ... | ... |

**(a)**




**(b)**

**Fig. 1** A sample database with a keyword query "James P2P". **a** Database (Matched keywords are underlined). **b** Examples of query results

20 most relevant matches for the keyword query—according to some definition of "Relevance"—are generally of interest [2]. Therefore, instead of finding all the results that containing the query keywords, proposing optimized methods which can efficiently find the top-$k$ results of the highest relevance scores are the main focus of a lot of existing work in this topic [2–4]. Although the proposed algorithms can avoid exhaustive processing by introducing some top-$k$ processing methods such as pipelined algorithms (hence can stop early before all the results are generated), they still suffer from a huge number of join checking which cannot produce results, and cannot effectively exploit the intermediate results of previous checking to facilitate the following computation.

In this paper, a novel algorithm is proposed to efficiently compute the top-$k$ results for keyword queries, which adopts the following three principles to achieve the high efficiency.

(1) *Bounding the relevance scores* computing the upper bound of the relevance scores of the results that containing a tuple $t$ is one of the main optimization directions of existing studies [2–4]. In this paper, we introduce a method to compute a tighter upper bound for each tuple, and the upper bound can be decreased in the query evaluating process; hence, more tuples can be filtered and the process can be terminated in a more early stage.
(2) *Reusing the query results* database accessing cost dominates the query evaluation cost for a keyword search. Reusing the results of previous checking can dramatically reduce the database accessing cost; however, this method has not been fully exploited in existing studies. This paper adopts the two ideas of caching and reusing the intermediate results, which can minimize the database accessing times and then achieve high efficiency.
(3) *Sharing the computation cost* although the results of keyword query in relational databases can have many *patterns* (see Fig. 1b), due to the query pattern enumeration style, there are many common sub-expressions between the query patterns of a keyword. Therefore, there is a great chance to share the computation cost among the query patterns in query evaluation. Based on the methods proposed in [5–7], this paper proposed a more experienced method to utilize this optimization direction.

This paper is an extended version of work published in [8]. We extend our previous work by more detailed description of the method, correctness proving of the algorithm, two crucial optimization methods that can highly improve the efficiency of the algorithm and abundant experiments to study the efficiency of the proposed methods on two real data sets. We summarize the key contributions of this paper as follows: (1) by incorporating the ranking mechanisms into the query processing method, an algorithm which can efficiently compute the top-$k$ results for keyword queries in a pipelined pattern is presented. (2) Four optimization methods which can highly improve the efficiency of the proposed algorithms are presented; and finally, (3) extensive experiments are conducted to evaluate the proposed approach.

The rest of this paper is organized as follows. In Sect. 2 some basic concepts are introduced. Section 3 discusses related work. Section 4 presents the details of the proposed algorithm. And four optimization methods that can highly improve the efficiency of the proposed algorithm are given in Sect. 5. Section 6 gives the experimental results. Finally, in Sect. 7 we conclude this paper.

## 2 Preliminaries and main challenges

In this section, we introduce some important concepts for top-$k$ keyword querying evaluation in relational databases, common in most of the existing keyword search systems [2,4,6,7].

### 2.1 Relational database model

We consider the schema of a relational database as a directed graph $G_S(V, E)$, called a schema graph, where $V$, the set of nodes of $G_S$, represents the set of relation schemas $\{R_1, R_2, \ldots, \}$ and $E$, the set of edges of $G_S$, represents the

foreign key references between pairs of relation schemas. Given two relation schemas, $R_i$ and $R_j$, $\langle R_i, R_j \rangle \in E$ if the primary key of $R_i$ is referenced by a foreign key defined on $R_j$. $\langle R_i, R_j \rangle$ can also be denoted as $R_i \leftarrow R_j$, where the arrow indicates the direction of the edge. For example, the schema graph of the publication database in Fig. 1 is $Papers \leftarrow Writes \rightarrow Authors$. A relation on relation schema $R_i$ is an instance of $R_i$ (a set of tuples) conforming to it, denoted as $r(R_i)$. In the following, we do not distinguish $R_i$ from $r(R_i)$ if the context is obvious.

### 2.2 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 interconnected if they can be joined based on a foreign key reference defined on relational schema $R_i$ and $R_j$ in $G_S$ (either $R_i \leftarrow R_j$ or $R_i \rightarrow R_j$). The foreign key references between tuples in a JTT can be denoted using arrows or notation $\bowtie$ . For example, the second JTT in Fig. 1b can be denoted as $a_1 \leftarrow w_1 \rightarrow p_2$ or $a_1 \bowtie w_1 \bowtie p_2$. 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. But the non-leaf tuples may not contain any keywords. In Fig. 1b, tuples $p_1$, $p_2$, $a_1$, $a_3$ are matched tuples to the keyword query as they contain the keywords. Hence, the four individual tuples and $a_1 \leftarrow w_1 \rightarrow p_2$ are valid results to the query. In contrast, $p_1 \leftarrow w_2 \rightarrow a_2$ is not valid because $a_2$ is not a matched tuple. The number of tuples in a JTT $T$ is called the *size* of $T$, denoted by $size(T)$.

Note that although a JTT is not required to contain all the keywords of a query (i.e., we adopt the *OR*-semantic), the scoring method, which is introduced later, ensures that the JTTs containing all the keywords would have higher relevance scores than those containing only a portion of keywords. The OR-semantic is adopted by all the top-$k$ keyword search studies [2,3]. In contrast, the *AND*-semantic requires each query result to contain all the keywords of a query, and is adopted by the studies aiming to find all the results for a keyword query [5–7].

### 2.3 Candidate networks (CNs)

Given a keyword query $Q$, the query tuple set $R_i^Q$ is defined as $R_i^Q = \{t \in r(R_i) | t$ contains some keyword of $Q\}$. For example, the two query tuple sets in Example 1 are $P^Q = \{p_1, p_2, p_5\}$ and $A^Q = \{a_1, a_3, a_5\}$, respectively. 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 $R_i^{Q \text{or} F}$ to denote a *tuple set*, which may be either $R_i^Q$ or $R_i^F$.

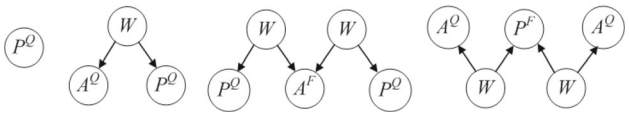



**Fig. 2** Examples of candidate networks

Each JTT belongs to the result of a relational algebra expression, which is called a *candidate network* (*CN*) [2,3,10]. 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 $R_i^{Q \text{or} F} \bowtie R_j^{Q \text{or} F}$ corresponds to an edge $\langle R_i, R_j \rangle$ in the schema graph $G_S$, where $\bowtie$ represents an *equi-join* between relations. For example, the CNs that correspond to two JTTs $p_2$ and $a_1 \leftarrow w_1 \rightarrow p_2$ 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 $P^Q \leftarrow W \rightarrow A^Q$. 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 $m$:$n$ relationships (for example, an article may be written by multiple authors), a CN may have multiple occurrences of the same tuple set. The size of a CN $C$, denoted as $size(C)$, is the number of its tuple sets, that is, the sizes of the JTTs it produces. Figure 2 shows the CNs corresponding to the four JTTs shown in Fig. 1b. A CN can be easily transformed into an equivalent SQL statement and executed by an RDBMS. For example, we can transform CN $P^Q \leftarrow W \rightarrow A^Q$ as:

SELECT * FROM W w, P p, A a WHERE w.pid = p.pid AND w.aid = a.aid AND p.pid in ($p_1$, $p_2$, $p_5$) and a.aid in ($a_1$, $a_3$, $a_5$).

When a keyword query $Q = \{w_1, w_2, \ldots, w_l\}$ is specified, the non-empty query tuple set $R_i^Q$ for each relation $R_i$ in the target database is firstly computed using full-text indices. Then all the non-empty query tuple sets and the database schema are used to generate the set of valid CNs. The first algorithm of CN generation is proposed in [5], whose basic idea is to expand each partial CN by adding an $R_i^Q$ or $R_i^F$ at each step ($R_i$ is adjacent to one relation of the partial CN in $G_S$), beginning from the set of non-empty query tuple sets. The set of CNs should be sound/complete and duplicate-free. There is always a constraint $CN_{max}$, which denotes the maximum size of CNs, to avoid generating complicated but less meaningful CNs. Luo [11] proposed a more efficient CN generating algorithm which can avoid the isomorphism checking of the enumerated CNs by defining a new canonical form for

CNs and computing the *depth first canonical form* for CNs. However, even a medium sized database schema graph and a medium value of $CN_{max}$ can result in a large number of CNs [11]. Therefore, in the implementation of our system, to achieve high efficiency, we generate a set of CNs in a pre-processing step by assuming that all the relations have non-empty query tuple sets like in [11].

*Example 2* In Example 1, there are two non-empty query tuple sets $P^Q$ and $A^Q$. Using them and the database schema graph, if $CN_{max} = 5$, the generated CNs are: $CN_1 = P^Q$, $CN_2 = A^Q$, $CN_3 = P^Q \leftarrow W \rightarrow A^Q$, $CN_4 = P^Q \leftarrow W \rightarrow A^Q \leftarrow W \rightarrow P^Q$, $CN_5 = P^Q \leftarrow W \rightarrow A^F \leftarrow W \rightarrow P^Q$, $CN_6 = A^Q \leftarrow W \rightarrow P^Q \leftarrow W \rightarrow A^Q$, $CN_7 = A^Q \leftarrow W \rightarrow P^F \leftarrow W \rightarrow A^Q$.

### 2.4 Scoring method

The problem of top-$k$ keyword search that we study in this paper is to compute the top-$k$ JTTs based on a certain scoring function, which will be described below. In the literature, several methods have been proposed for measuring the relevance of keyword search results in relational databases [2,3,12–15]. We adopt the scoring method employed in [2], 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:

$$score(T, Q) = \frac{\sum_{t \in T} tscore(t, Q)}{size(T)}, \tag{1}$$

where is a tuple contained in $T$. $tscore(t, Q)$ is the *tuple score* of $t$ with regard to $Q$ defined as follows:

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

where $tf_{t,w}$ is the *term frequency* of keyword $w$ in tuple $t$, $df_w$ is the number of tuples in relation $r(t)$ (the relation corresponds to tuple $t$) that contain $w$. $df_w$ is interpreted as the *document frequency* of $w$. $dl_t$ represents the size of tuple $t$, that is, the number of letters in $t$, and is interpreted as the *document length* of $t$. $N$ is the *total number* of tuples in $r(t)$, $avdl$ is the *average tuple size* (*average document length*) in $r(t)$, and $s$ ($0 < s < 1$) is a constant which is usually set to 0.2.

Table 1 shows the tuple scores of the six matched tuples in Example 1 we suppose all the matched tuples are shown in Fig. 1, and the numbers of tuples of the two relations are 150 and 170, respectively. Therefore, the top-3 results are

**Table 1** Statistics and tuple scores of tuples in $P^Q$ and $A^Q$

| Tuple sets | $P_Q$ | | | $A_Q$ | | |
|---|---|---|---|---|---|---|
| Statistics | $N$ | $df_{\text{P2P}}$ | $avdl$ | $N$ | $df_{\text{James}}$ | $avdl$ |
| | 150 | 3 | 57.8 | 170 | 3 | 14.6 |
| Tuple | $P_1$ | $P_2$ | $P_5$ | $a_1$ | $a_3$ | $a_5$ |
| $dl$ | 88 | 28 | 83 | 10 | 22 | 23 |
| $tf$ | 1 | 3 | 1 | 1 | 1 | 1 |
| $tscore$ | 3.28 | 7.04 | 3.33 | 4.03 | 3.40 | 3.36 |

$T_1 = p_2$ ($score = 7.04$), $T_2 = a_1$ ($score = 4.00$) and $T_3 = a_1 \leftarrow w_1 \rightarrow p_2$ ($score = (7.04 + 4.00)/3 = 3.68$).

The score function in Eq. (1) has the property of *tuple monotonicity*, defined as follows. For any two JTTs $T = t_1 \bowtie t_2 \bowtie \ldots \bowtie t_l$ and $T' = t'_1 \bowtie t'_2 \bowtie \ldots \bowtie t'_l$ generated from the same CN $C$, if for any $1 < i < l$, $tscore(t_i, Q) < tscore(t'_i, Q)$, then we have $score(T, Q) < score(T', Q)$. As shown in the following discussion, this property is critical to the existing top-$k$ query evaluation algorithms.

## 3 Related work

Keyword search in relational databases has attracted substantial research effort in recent years, which can be categorized into two approaches. The *graph-based* methods [16–29] 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 [16], distinct rooted trees [18], $r$-radius Steiner graphs [19], multi-center subgraphs [21] or $r$-clique [24]) from the database graph. For the details, please refer to the survey papers [1,30]. The *schema-based* approaches [2–5,8,10,31–41] in this area utilize the database schema to generate SQL queries. After receiving a keyword query, they first utilize the database schema to generate a set of CNs, which can be interpreted as select-project-join views and all have explicit meanings. Then, these CNs are evaluated by sending the corresponding SQL statements to the RDBMS to find JTTs. A data graph cannot exploit the semantics of the underlying database schema directly. Another drawback of the data graph model is that a graph of the tuples must be materialized and maintained; therefore, it may not be scalable when maintaining a large size database [30]. This paper adopts the schema-based framework for query processing, but materializes small fractions of the entire database graph in the process of query processing.

There would be a huge number of valid results for a keyword query in a large database. However, only the top 10 or 20 most relevant matches for the keyword query—according to some definition of "Relevance"—are generally of inter-

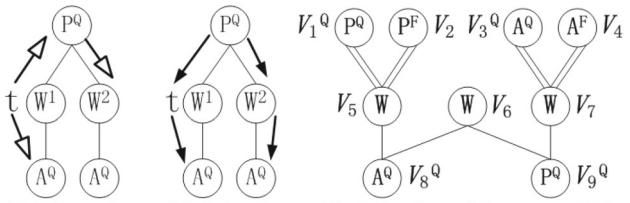



**Fig. 3** Query processing in KDynamic

est [2]. DISCOVERII [2], SPARK [3,42] and SPARKII [4,11] efficiently execute top-$k$ queries by avoiding the creation of all the query results. DISCOVERII proposed the *global-pipelined* (*GP*) algorithm. For a keyword query $Q$, given a CN $C$, let the set of query tuple sets of $C$ be $\{R_1^Q, R_2^Q, \ldots, R_m^Q\}$. Tuples in each $R_i^Q$ are sorted in non-increasing order of their scores computed by Eq. (2). For each tuple $R_i^Q \cdot t_j$, the upper bound score for all the JTTs of $C$ that contain $R_i^Q \cdot t_j$, denoted as $\overline{score}$, is computed. Algorithm GP initially mark all tuples in each tuple set as un-processed except for the top-most one. Then in each iteration (one round), the un-processed tuple, assume it be $C_0 \cdot R_s^Q \cdot t_p$, maximizes the $\overline{score}$ is selected for processing, which is done by testing all the combinations as $(t_1, t_2, \ldots, t_{s-1}, R_s^Q \cdot t_p, t_{s+1}, \ldots, t_m)$, where $t_i$ is a processed tuple of $C_0 \cdot R_i^Q$ $(1 < i < m, i \neq s)$. If the $k$th relevance score of the found results is larger than $\overline{score}$ values of all the un-processed tuples in all the CNs, GP stops and outputs the $k$ found results with the largest relevance scores.

One drawback of the GP algorithm is that when a tuple $C \cdot R_s^Q \cdot t_p$ is processed, it has to test all the combinations as $(t_1, t_2, \ldots, t_{s-1}, R_s^Q \cdot t_p, t_{s+1}, \ldots, t_m)$. This operation is costly due to extremely large number of combinations when the number of processed tuples becomes large [9]. SPARK proposed the *skyline-sweeping* and *block-pipeline* (BP) algorithms, which highly reduce the number of tested combinations. SPARKII proposed the *tree-pipeline* algorithm, which can share the computational cost among CNs in some extent, using the *binary decompositions* of them. However, SPARK and SPARKII still cannot avid testing a huge number of combinations which cannot produce results.

*KDynamic* [7,32] formalizes each CN as a rooted tree, whose root is defined to be the node $r$ such that the maximum path from $r$ to all leaf nodes is minimized.[1] Figure 3a shows the rooted tree of $CN_6$. Each node $V_i$ in the rooted trees is associated with an output buffer, denoted by $V_i \cdot \mathbb{OP}$, which contains the tuples of $V_i$ that can join at least one tuple in the output buffer of its each child. Tuples in the output buffer are called the output tuples of the node. Thus, each output tuple of the root can form JTTs with the output tuples of its descendants.

Tuples of CNs are processed in a two-phase approach in the rooted tree. In the filter phase, as illustrated in Fig. 3a, when a tuple $t$ is processed at the node $W^1$, KDynamic uses selections to check if (1) $t$ can join at least an output tuple of each child of $W^1$; and (2) $t$ can join at least an output tuple of the ancestors of $W^1$. The tuples that cannot pass the checks are pruned; otherwise, in the join phase (shown in Fig. 3b), a joining process is initiated from each output tuple of the root node that can join $t$, in a top-down manner, to find the JTTs involving $t$. KDynamic achieves full tuple reduction by pruning the tuples that cannot form JTTs, and thus the join operations can always produce results.

In order to share the computation cost among CNs, all the rooted trees are compressed into a $\mathcal{L}$-lattice by collapsing their common subtrees. Thus, the output tuples of a node are shared by more than one nodes, among different CNs. Figure 3c shows the lattice of the seven CNs. We use $V_i^Q$ to denote a node of query tuple set particularly. The dual edges between two nodes, for instance, $V_1^Q$ and $V_5$, indicate that $V_5$ is a dual child of $V_1^Q$.

Evaluating the CNs using the lattice can achieve full reduction because all the output tuples of the root nodes can form JTTs. However, KDynamic cannot evaluate the CNs in a pipelined way to support top-$k$ result computing. The two important ideas of the pipelined query evaluation methods of DISCOVERII, SPARK and SPARKII are: (a) calculate upper-bounds for the relevance score of the un-found results; and (b) prune unnecessary calculations in finding the top-$k$ results. In the following, we incorporate the ranking mechanisms and the pipelined evaluation into the above query processing method of KDynamic, and makes several optimization methods to support efficient top-$k$ keyword search in relational databases.

## 4 Pipelined evaluation of lattice

In this section, we will show the method of evaluating the lattice in a pipelined way to find the top-$k$ results. Firstly, we present the algorithm of evaluating the lattice. Then, we prove the correctness of the algorithm. Lastly, we use the execution process of the lattice of the seven CNs as an example.

### 4.1 The LP algorithm

In order to find the top-$k$ results in a pipelined way, we first sort tuples in each node $V_i^Q$ in non-increasing order of *tscore*. We use $V_i^Q.cur$ to denote the index such that the tuples in $V_i^Q$ before it are all processed; and we use $V_i^Q.cur \leftarrow V_i^Q.cur + x$ to move $V_i^Q.cur$ to the next $x$ position. Initially, for each

[1] Note that the CN defined in KDynamic has some differences with ours.

node $V_i^Q$ in $\mathcal{L}$, $V_i^Q.cur$ is set to the top tuple in $V_i^Q$, i.e., the tuples have the maximum tuple score. Note that, for a node $V_i$ that is of a free tuple set $R_i^F$, we regard all its tuples as processed tuples for all the times.

The key to evaluate queries in a pipelined way in DISCOVERII and SPARK is to compute an upper bound for the relevance score of the un-found results. For a keyword query $Q$, given a CN $C$, let the set of query tuple sets of $C$ be $\{R_1^Q, R_2^Q, \ldots, R_m^Q\}$. For each tuple $R_i^Q \cdot t_j$, DISCOVERII computes the upper bound score for all the JTTs of $C$ that contain $R_i^Q \cdot t_j$ as:

$$\overline{score}\left(C \cdot R_i^Q \cdot t_j,\ Q\right) = \frac{t_j \cdot iscore + \sum_{i' \neq i} C \cdot R_{i'}^Q \cdot t_1 \cdot tscore}{size(C)}, \quad (3)$$

where $C \cdot R_{i'}^Q \cdot t_1$ indicates the top-most tuple of query tuple set $C \cdot R_{i'}^Q$. Using this equation, for each node $V_i^Q$, this paper computes the maximum *score* of the found JTTs by processing the un-processed tuples at $V_i^Q$ as:

$$\overline{score}\left(V_i^Q,\ Q\right) = \left\{ \begin{array}{l} 0, \text{ a child of } V_i^Q \text{ has empty } \mathbb{OP} \\ \max_{C \in V_i^Q \cdot CN} \overline{score}(C \cdot V_i^Q \cdot t_{cur},\ Q) \\ otherwise \end{array} \right\}, \quad (4)$$

If a child of $V_i^Q$ has an empty output buffer, processing any tuple at $V_i^Q$ cannot produce JTTs; hence $\overline{score}(V_i^Q, t_j, Q) = 0$ in such cases, which chokes the tuple processing at $V_i^Q$ until all its child nodes have non-empty output buffers. This property of $\overline{score}(V_i^Q, Q)$ can be seen as our version of the event-driven evaluation in KDynamic, which is firstly proposed in *S-KWS* [6] and can noticeably reduce the query processing cost. For instance, the $\overline{score}$ value of the node $V_8^Q$ shown in Fig. 3c is computed as $max_{C \in \{CN_2, CN_7\}}(\overline{score}(C \cdot A^Q \cdot a_1,\ Q)) = 4.00$.

*LP* algorithm (Algorithm 1) outlines our pipelined algorithm of evaluating lattice $\mathcal{L}$ to find the top-$k$ results. Lines 1–3 are the initialization steps, which sort tuples in each query tuple set and initialize each $V_i^Q.cur$. Then in each while iteration (lines 4–8), *step* un-processed tuples in the node $V^Q$ which maximizes $\overline{score}(V_i^Q, Q)$ are processed. Processing tuples at a node is done by calling the procedure *Insert*. Algorithm 1 stops when $max_{V_i^Q \in \mathcal{L}} \overline{score}(V_i^Q, Q)$ is not larger than the relevance score of the top-$k$th found result because no results with larger relevance scores can be found in the further evaluation. The procedure *Insert*($V_i$, $\mathbb{S}$) is firstly provided in KDynamic, which updates the output buffers for $V_i$ (line 12) and all its ancestors (lines 17–19),

**Algorithm 1:** *LP* (lattice $\mathcal{L}$, the top-$k$ value $k$, integer *step*)

```
1  topk ← ∅: the priority queue for storing found JTTs ordered by score;
2  Sort tuples of each V_i^Q.R^Q in non-increasing order of tscore^u;
3  foreach node V_i^Q in L do let V_i^Q.cur ← 1;
4  while max_{V_i^Q ∈ L} score(V_i^Q, Q) > topk[k].score do
5      Suppose score(V_0^Q, Q) = max_{V_i^Q ∈ L} score(V_i^Q, Q);
6      path ← ∅;    // A stack which records the join sequence
7      Insert(V_0^Q, {V_0^Q.t_cur, ..., V_0^Q.t_{cur+step-1}});
       // Process step tuples at V_0^Q
8      V_0^Q.cur ← V_0^Q.cur + step;
9  Output the first k results in topk;
10 Procedure Insert(lattice node V_i, set of tuples S)
11 Let S' ← {t | t ∈ S, t can join at least one outputted tuple of every child of V_i };
12 Add the tuples in S' into V_i.OP;
13 if S' ≠ ∅ then
14     Push (V_i, S') to path;
15     if V_i is a root node then
16         Add the JTTs in EvalPath(V_i, S', path) into topk ;
17     foreach father node of V_i, V_i' in L do
18         Let S'' be the set of processed tuples of V_i' that can join tuples in S';
19         if S'' ≠ ∅ then Insert(V_i', S'');
20     Pop (V, S') from path;
21 Procedure EvalPath(node V_i, tuple set S, stack path)
22 T ← S;                    // The set of found JTTs
23 foreach V_i's child node V_i' in L do
24     if V_i' ∈ path then
25         S' ← the set of output tuples of V_i' that are stored in path;
26     else
27         S' ← the set of output tuples of V_i' that can join tuples in S;
28     T' ← EvalPath(V_i', S', path);
29     T ← T ⋈ T';    // Join the JTTs in the two sets
30 return T;
```

and finds all the JTTs containing tuples of $\mathbb{S}'$ by calling the procedure *EvalPath* (line 16), which is firstly provided by KDynamic too. In KDynamic, the second parameter of *Insert* and *EvalPath* is one tuple. As shown by the *BP* algorithm of [3], processing tuples in batch can achieve high efficiency due to the reduced numbers of database accesses. Hence, tuples are processed in batch in Algorithm 1: *step* tuples are processed when *Insert* is called in line 7, and *EvalPath* also handles a set of tuples. However, it is not the larger *step* is, the higher efficiency of *LP* algorithm has. Because a larger step can result in un-necessary tuple processing in some lattice nodes. We will experimentally study how to select a proper *step* value. The recursive procedure $EvalPath(V_i, \mathbb{S}, path)$ constructs JTTs using the output tuples of $V_i$'s descendants that can join tuples in $\mathbb{S}$. The stack

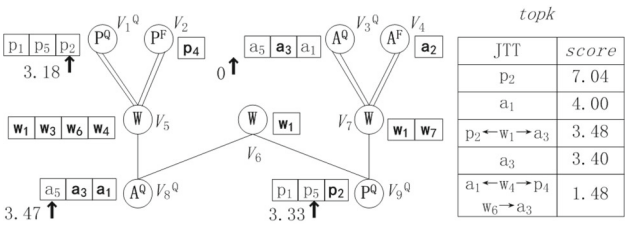



**Fig. 4** After finding the top-3 results (output tuples are shown in bold)

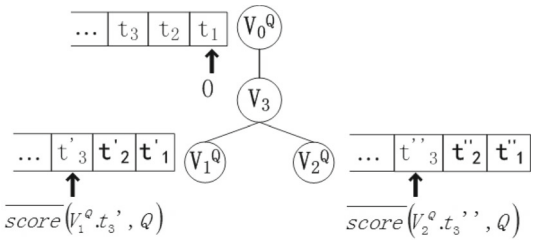


**Fig. 5** A lattice after finding the top-k results

*path* is used to record the join sequence for reducing the join cost (see line 25).

*Example 3* Figure 4 shows the lattice of the seven CNs after finding the top-3 results, which shows $V_i^Q.cur$ and $\overline{score}(V_i^Q, Q)$ value of the four $V_i^Q$ nodes same as Fig. 5. Suppose $step = 1$, then in the first round, tuple $V_9^Q \cdot p_2$ is processed by calling $Insert(V_9^Q, \{p_2\})$. Since $V_9^Q$ is the root of $CN_1$, $EvalPath(V_9^Q, \{p_2\})$ is called and JTT $T_1 = p_2$ is found. Then, for the two father nodes of $V_9$, $V_6$ and $V_7$, since tuples $\omega_1$ and $\omega_7$ can join $p_2$, $Insert(V_6, \{\omega_1, \omega_7\})$ and $Insert(V_7, \{\omega_1, \omega_7\})$ are called. $V_6 \cdot \mathbb{OP}$ is not updated because $V_8^Q \cdot \mathbb{OP} = \emptyset$; $V_7 \cdot \mathbb{OP}$ is updated to $\{\omega_1, \omega_7\}$. And then, for the two father nodes of $V_7$, $V_3^Q$ and $V_4$, $V_3^Q \cdot \mathbb{OP}$ is not updated since $V_3^Q$ has no processed tuples, and $V_4^Q \cdot \mathbb{OP}$ is set as $\{a_2\}$ because there is only one tuple $a_2$ in $A^F$ that can join $\omega_1$ and $\omega_7$. Since $V_4$ is the root node of $CN_5$, $EvalPath(V_4, \{a_2\}, path)$ is called but the found JTT $p_2 \leftarrow \omega_7 \rightarrow a_2 \leftarrow \omega_7 \rightarrow p_2$ is not a valid result. After processing $V_9^Q \cdot p_2$, $\overline{score}(V_3^Q, Q) = 3.61$ and $\overline{score}(V_9^Q, Q) = \overline{score}(CN_1 \cdot P^Q \cdot p_5, Q) = 3.33$. In the second round, $Insert(V_8^Q, a_1)$ is called, … Lastly, Algorithm 1 stops because *top-k* [3]. *score* is larger than all the $\overline{score}(V_i^Q, Q)$ values.

### 4.2 Proving correctness

**Theorem 1** *After the execution of Algorithm 1, the score values of all the un-found results are not larger than the relevance score of the* (*k*)*th result in the queue top-k.*

*Proof* Relying on the tuple monotonicity property of Eq. (3), in case all the $V_i^Q$ nodes in the lattice having non-zero $\overline{score}(V_i^Q, Q)$ values, it is clear that Theorem 1 is correct. However, if a node $V_0^Q$ has a child node with empty output buffer, its $\overline{score}(V_0^Q, Q)$ would be zero. Then, if we continue inserting tuples into its descendants, all its child nodes can have outputted tuples, and $\overline{score}(V_0^Q, Q)$ is changed to a non-zero value. Since $V_0^Q$ is not evolved in previous evaluation, $\overline{score}(V_0^Q \cdot R^Q \cdot t_1, Q)$ can be larger than $top\text{-}k[k] \cdot score$. Then, tuples of $V_0^Q$ are processed and results that are of $score > top\text{-}k[k] \cdot score$ may be found, which makes Theorem 1 to be wrong. However, following example reveals that even if the above happens, the found results cannot be of $score > top\text{-}k[k] \cdot score$. □

Figure 5 shows a lattice after running the LP algorithm. The arrows in Fig. 5 denote the three $V_i^Q.cur$, and the three $\overline{score}$ values are shown next to the corresponding arrows, respectively. Since the first two tuples of $V_1^Q$ and $V_2^Q$ are processed, $\overline{score}(V_1^Q \cdot t_3', Q) \leq top\text{-}k[k] \cdot score$ and $\overline{score}(V_2^Q \cdot t_3'', Q) \leq top\text{-}k[k] \cdot score$. And since $V_3$ has no output tuples, $\overline{score}(V_0^Q, Q) = 0$. Now suppose inserting tuple $t_3'$ into $V_1^Q$ or inserting $t_3''$ into $V_2^Q$ can result in some output tuples in $V_3$, which can change $\overline{score}(V_0^Q, Q)$ to a non-zero value. Because no tuples have been processed in $V_0^Q$, $\overline{score}(V_0^Q, Q) = \overline{score}(V_0^Q \cdot t_1, Q)$ can be larger than $top\text{-}k[k] \cdot score$. And then $t_1$ is processed and the produced results can be of $score > top\text{-}k[k] \cdot score$. However, the produced results must contain $t_3'$ or $t_3''$, which means that their *score* values are bounded by $\overline{score}(V_1^Q \cdot t_3', Q)$ or $\overline{score}(V_2^Q \cdot t_3'', Q)$. Hence, even if $\overline{score}(V_0^Q, Q) = 0$, the *score* values of all the un-found results are not larger than θ. Therefore, the theorem is proved.

## 5 Optimization methods

In this section, we introduce four optimization methods that can highly improve the efficiency of LP algorithm. Section 5.1 describes the method to avoid computing results of small relevance scores; Sect. 5.2 introduces the method of clustering CNs based on their potentials in producing top-*k* results; Sect. 5.3 presents the approach of optimizing the lattice construction; and Sect. 5.4 shows how to cache the joined tuples for each tuple to reduce the database access operations.

### 5.1 Tuple filtering

Equation (3) assumes that tuple $R_i^Q \cdot t_j$ can form JTT with the first tuple of every query tuple set $R_i^Q (\neq R_i^Q)$ of $C$. This assumption can produce a serious overestimation for the real maximum relevance score of the JTTs that $R_i^Q \cdot t_j$ can

form, due to the small possibility for the assumption being correct. However, Eq. (3) is the best estimate that we can produce efficiently without accessing the database [2]. As a result, Algorithm 1 may find some results that have very low relevance scores, e.g., JTT $a_1 \leftarrow w_4 \rightarrow p_4 \leftarrow w_6 \rightarrow a_3$ in Fig. 4.

Fortunately, when Algorithm 1 processing tuples in the lattice, the practical join relationship of tuples can be found progressively, which can be used to reduce the $\overline{score}(V_i^Q \cdot t,\ Q)$ value computed by Eq. (3) towards the real value gradually. Therefore, we propose to add the following operation after line 12 of Algorithm 1:

**Delete every tuple $t$ that is of $\underline{score}\left(V_i^Q \cdot t,\ Q\right) \leq$ *top-k*$[k] \cdot$ *score* from** $\mathbb{S}'$; where $\underline{score}(V_i^Q \cdot t,\ Q)$ also indicates the maximum relevance score of JTTs formed by $t$, but is computed using the practical join relationship of tuples in the lattice:

$$\underline{score}\,(V_i \cdot t,\ Q) = \max_{C \in V_i \cdot CN} \left( \frac{S_1(t) + s_2}{sizeC} \right),$$
$$S_1(t) = t \cdot tscore + \sum_{V_j} \max_{t' \in V_j \cdot \mathbb{DP} \wedge (t' \bowtie t)} S_1(t'), \tag{5}$$

where $V_j$ is a child node of $V_i$. $S_1(t)$ indicates the maximum relevance score of the tuple trees which are rooted on tuple $t$ and consist of the output tuples that can join $t$ of the descendants of $V_i^Q$. And $S_2(t)$ is computed using Eq. (6):

$$S_2 = \sum_{R_{j'}^Q} C \cdot R_{j'}^Q \cdot t_1 \cdot tscore, \tag{6}$$

where $R_{j'}^Q$ is different from $V_i$ and its descendants.

In the implementation, in order to compute $S_1(t)$ in Eq. (5), for each output tuple $t$ of node $V_i$, we record the maximum tuple score of the joined output tuple in each child node of $V_i$. Then $S_1(t)$ is updated continually at the execution process of Algorithm 1. Not that $S_2$ is the intersection part with Eq. (3) and is computed using the tuple score of the first tuple too. For an output tuple $t$ of a leaf node $V_i$ in the lattice, its $\underline{score}(V_i \cdot t,\ Q)$ equals to $\overline{score}(V_i^Q \cdot t,\ Q)$. But as the procedure *Insert* is called recursively from the bottom to the top at the lattice, $\underline{score}(V_i \cdot t,\ Q)$ will be keep getting close to the real maximum relevance score of the JTTs formed by $t$ since the number of tuples for computing $S_2$ is reducing. Finally, when $V_i$ is a root node, $S_2$ falls to zero, and there will be no more overestimation in $\underline{score}(V_i \cdot t,\ Q)$. Therefore, the JTTs of small relevance scores can be filtered out as much as possible.

For computing $\underline{score}(V_i^Q \cdot t,\ Q)$ for a tuple $t$, we need to find out all the tuples that $t$ can join from the output tuples of $V_i^Q$'s child nodes. By comparison, line 11 of Algorithm 1 only need to check whether $t$ can join at least one output tuple

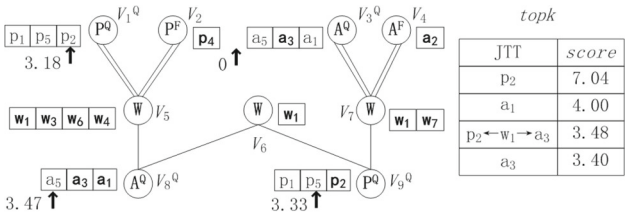



**Fig. 6** After finding the top-3 results, while filtering out results of small relevance scores

of $V_i^Q$'s each child node, which can be done by the RDBMS. However, when the number of output tuples is large, the needed checking in line 11 will be more efficient if it is done by the keyword search system. And in the fourth optimization method described in this section, we also need to find out all the tuples that t can join at the relations of $V_i^Q$'s child nodes. As can be seen from the experimental results, using Eq. (5) to delete tuples that are $\underline{score}(V_i \cdot t,\ Q) \leq top\text{-}k[k] \cdot score$ after line 12 can highly improving the efficiency of Algorithm 1.

*Example 4* Figure 6 shows the lattice after computing the top-3 result, while adopting the optimization method described in this section. Its main difference with Fig. 4 is that the JTT $a_1 \leftarrow \omega_4 \rightarrow p_4 \leftarrow \omega_6 \rightarrow a_3$ is not found. This is because in the round six of Example 3, the $\underline{score}$ of tuple $w_1$, $w_3$ and $w_6$ are all smaller than $top\text{-}k[3] \cdot score = 3.40$; hence, $Insert(V_2, \{p_4\})$ is not called, and then the JTT $a_1 \leftarrow \omega_4 \rightarrow p_4 \leftarrow \omega_6 \rightarrow a_3$ can avoid being computed.

### 5.2 Candidate network clustering

According to Eq. (1), relevance scores of JTTs of different CNs have great differences. For example, relevance scores of JTTs of $CN_5$ and $CN_7$ are smaller than that of JTTs of $CN_3$ due to their large sizes. And then the same tuple set can have different numbers of processed tuples in different CNs if they are evaluated separately. If the seven CNs are evaluated separately, $A^Q$ of $CN_7$ would have no processed tuples. However, in the lattice, a node $V_i^Q$ can be shared by multiple CNs. For instance, the node $V_8^Q$ in Fig. 3c is shared by $CN_2$, $CN_3$, $CN_6$ and $CN_7$. We use $V_i \cdot CN$ to denote the set of CNs that node $V_i$ belongs to. Then, when processing a tuple $t$ at node $V_8^Q$, $t$ is processed in all the CNs in $V_8^Q \cdot CN$; hence some results of $CN_7$ can be computed, which would have very small relevance scores and cannot contribute to the top-$k$ results.

The essence of the above problem is that CNs have different potentials in producing the top-$k$ results. Thus, the CNs that have big differences in such potentials should not share tuple sets. The optimal method is merely to share the tuple sets which have the same set of processed tuples if CNs are

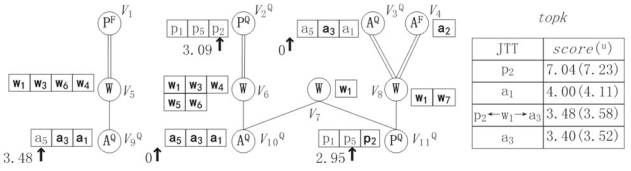



**Fig. 7** After finding the top-3 results if the seven CNs are clustered into two clusters and JTTs of small relevance scores are filtered out

evaluated separately. However, we cannot get these sets without evaluating them. As an alternative, we attempt to estimate these sets according to two heuristic rules:

- If $Max(C) = \frac{\sum_{1 \le i \le m} C \cdot R_i^Q \cdot t_1 \cdot tscore}{size(C)}$ (which indicates the maximum *score* of JTTs that C can produce) is high, the number of processed tuples of tuple sets of $C$ is large.
- If two CNs have the same *Max(C)* values, tuple sets of the CN with larger size have more processed tuples.

Therefore, we can use $Max(C) \cdot \ln(size(C))$ to measure the potential of a CN in producing top-$k$ results, where $\ln(size(C)$ is used to normalize the effect of CN sizes. Then, we can cluster the CNs using their $Max(C) \cdot \ln(size(C))$ values, and only the subtrees of CNs in the same cluster can be collapsed when constructing the lattice. For instance, $Max(C) \cdot \ln(size(C))$ of the seven CNs are: 5.15, 2.93, 5.39, 6.84, 5.32, 5.70 and 3.03; hence they can be clustered into two clusters: $\{CN_2; CN_7\}$ and $\{CN_1; CN_3; CN_4; CN_5; CN_6\}$. Figure 7 shows the lattice after finding the top-3 results, while the CNs are clustered and the optimization method described in Sect. 5.1 is adopted. We can see that clustering the seven CNs further reduced the number of computed JTTs compared to Fig. 6: merely the top-3 results are found.

In the implementation, the CNs are clustered using the $K$-means clustering algorithm [43], which needs an input parameter to indicate the number of expected clusters. And then an independent lattice is constructed for each cluster of CNs. We use *Kmean* to indicate this parameter. The value of *Kmean* represents the tradeoff between sharing the computation cost among CNs and considering their different potentials in producing top-$k$ results. The CNs is not clustered when $Kmean = 1$, then the computation cost is shared at the maximum extent. When $Kmean =$ MAX, all the CNs are evaluated separately.

The time complexity of K-means is approximate to O(#CN), where #CN is the number of CNs. Since the number of CNs cannot be very high (smaller than 1000), CN clustering will not introduce perceptible additional cost to the query evaluation process. As shown in the experimental section, clustering the CNs can highly improve the efficiency in computing the top-$k$ results, and the optimal *Kmean* depends on *CNmax*.

### 5.3 Optimization of lattice construction

For constructing the lattice, when modeling each CN as a rooted tree, *KDynamic* selects the root node as the one whose maximum path to all the leaf nodes is minimized [32]. Although the above policy of selecting the root node can result in the smallest maximum height of each rooted tree (which is $CN_{max}/2+1$), the resulting lattice may not be the optimal, because the popularity and the number of results of subtrees are not considered. Here, the popularity of a subtree is measured by the number of its occurrences in all the CNs. Therefore, the resulting lattice can have the following two problems:

(1) There would be a large number of nodes because the subtrees of some rooted tree cannot be shared by many CNs;
(2) The number of output tuples of some nodes is large; however, none or only a small portion of them can produce JTTs.

DISCOVER [5] has shown that the popularity and the number of results of the common sub-expressions should be considered when sharing them between the CNs. Hence, these two factors must be considered in the construction of the lattice. However, constructing the optimal lattice that has the least computation cost in finding the top-$k$ result for a keyword query is very hard. Let the number of tuple sets in a CN $C$ be $CN_{max}$, then $C$ can have maximal $CN_{max}$ different rooted tree forms (each tuple set can be a tree root and determines a distinct rooted tree). Then the number of all the combinations is $\#CN^{CN_{max}}$, of which all should be considered in constructing the optimal lattice. Recall that $\#CN$ grows exponentially while $CN_{max}$ increases [4,5]; hence, even for a small $CN_{max}$, the time cost for computing the optimal lattice cannot be accepted.[2] Similar to DISCOVER, this paper proposes a greedy algorithm, as shown in Algorithm 2, to compute the near-optimal lattice by choosing the rooted sub-tree that has the maximum *profit* to be shared between CNs in each iteration, until all the CNs are constructed to rooted trees.[3] The method of computing profit for a sub-tree is described later.

Figure 8 shows all the possible rooted sub-trees of $CN_5$ at the beginning of Algorithm 2. The right-most three trees in Fig. 8 are the three possible rooted trees that $CN_5$ can be modeled as. Note that the other two rooted trees of $CN_5$ are identical to two of them. After a sub-tree *sTree* of a CN $C$ has been shared with other CNs, then the rooted trees that are *conflict* with *sTree* will not be considered in the follow-

[2] It has been proven to be a NP-complete problem in DISCOVER.

[3] It worth noting that every CN is started as an un-rooted tree, or a free tree.

**Algorithm 2:** *OPLattice* (a set of CNs *CNSet* )

```
1 while not all the CNs have been constructed to rooted trees do
2     Let sTree be the rooted sub-tree which has the maximum
      profit of the CNs that have not been constructed to rooted
      trees;
3     foreach C ∈ CNSet do
4         Use sTree to replace all the sub-trees of C that are
          identical to it;
```

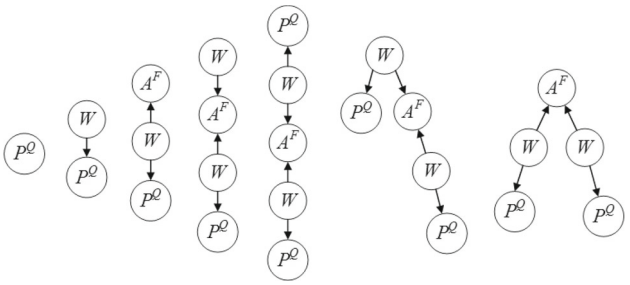


**Fig. 8** All the rooted sub-trees of $CN_5$ at the beginning of Algorithm 2

ing iterations. For example, if $\mathbf{W} \leftarrow P^Q$ (rooted at $W$) has been shared with other CNs, then in the following iteration of Algorithm 2, the second right-most rooted tree in Fig. 8 should not be considered because $\mathbf{W} \leftarrow P^Q$ cannot be a sub-tree of it.

The method of computing the profit of rooted sub-trees is similar with that of DISCOVER:

$$profit(sTree) = \frac{freq^a}{\log^b(size)}, \tag{7}$$

where $freq$ is the time that sub-tree $sTree$ appears in all the CNs, $size$ is the estimated number of results of $sTree$, $a$ and $b$ are two constants that represent a trade-off between the two factors. We have experimented with multiple combinations of values for $a$ and $b$ and found that the optimal solution is closer approximated for $\{a, b\} = \{0, 1\}$ for most of the situations.

In the worst case, all the CNs cannot share any sub-trees, then we have to consider all the rooted sub-trees of each CN in line 2 of Algorithm 2. For a CN of size $CN_{max}$, it can be modeled as $CN_{max}$ different rooted tree, each of which has at most $CN_{max}$ sub-trees. But after a sub-tree *sTree* is selected, in the next iteration of $CN_{max}$, there are at most $(CN_{max} - 1)^2$ need to be considered in line 2. Therefore, the upper bound of time complexity of Algorithm 2 is $\mathrm{O}\left(\#CN \cdot \sum_{i=1}^{CN_{max}} i^2\right) = \mathrm{O}(\#CN \cdot CN_{max}^3)$. Since $\#CN$ grows exponentially while $CN_{max}$ increases, $\mathrm{O}(\#CN \cdot CN_{max}^3) \approx \mathrm{O}(\#CN)$. Therefore, the optimization of lattice construction will not introduce perceptible additional cost to the query evaluation process too. And it can highly improve the efficiency in computing the top-$k$ results in the experimental study.

### 5.4 Caching joined tuples

In Algorithm 1, procedure $Insert$ may be called multiple times upon multiple nodes for the same tuple. And the procedure $EvalPath$ may also be called multiple times for the same tuple in procedure *Insert*. The core of these two procedures are the *select operations*. For example, line 11 selects the tuples that can join tuples of $\mathbb{S}$ from the output buffer of each child node of $V_i$. Although such select operations can be done efficiently by the RDBMS using indexes, the cost is high due to the large number of database accesses. For example, in our experiments, for a tuple $t$, the maximal number of database accesses can be up to several hundred.

In this paper, the selections in $Insert$ and $EvalPath$ are done efficiently by caching the joined tuples for each tuple. Algorithm 3 shows our procedure to find the tuples in $\mathbb{S}$ that can join at least one output tuple of node $V_i$, which is called in line 11 of procedure *Insert*. For each tuple $t$ in $\mathbb{S}$, if the joining tuples of relation $R_i$ are not cached, they are queried from the database and are stored into $t$ in line 3. The procedures of doing the selections in line 18 of *Insert*, and line 27 of $EvalPath$ are also designed in this pattern, which are omitted due to the space limitation. Since the two procedures are called recursively, for each tuple $t$, a tree rooted at $t$ and consist of all the tuples that can join $t$ is created temporarily, which can be seen as the cached localization information of $t$ and is denoted as $\mathcal{T}$. Since $\mathcal{T}$ of different tuples can share the same tuples, fractions of the database graph are created.

Assume procedure $Insert$ is called three times at $V_3^Q$, $V_9^Q$ and $V_{10}^Q$ for a tuple $a_0$, which would incur at most seven selections denoted by arrows in the left part of Fig. 9. For instance, the arrow from $V_3^Q$ to $V_8$ selects the output tuples of $V_8$ that can join $a_0$. There are three selections denoted by dashed arrows because they would not be done if the results of the three selections: from $V_9^Q$ to $V_5$, from $V_{10}^Q$ to $V_6$ and from $V_{10}^Q$ to $V_7$, are empty. If both the two selections, from $V_9^Q$ to $V_5$ and from $V_5$ to $V_1$, have non-empty results, $EvalPath$ is

**Algorithm 3:** *CanJoinOneOutputTuple*(lattice node $V_i$, set of tuples $\mathbb{S}$)

```
1 Let R_i be the relation corresponding to the tuple set of V_i;
2 Let S' ← {t | t ∈ S, t does not store the joining tuples of relation
  R_i };
3 if S' ≠ ∅ then Query the joining tuples of relation R_i for tuples
  in S';
4 foreach Tuple t in S do
5     if the stored joining tuples of relation R_i in t has empty
      intersection with V_i.OP then Remove t from S;
6 return S;
```

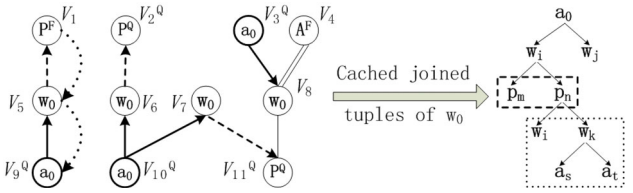



**Fig. 9** Selections done in *Insert* and *EvalPath* and the cached joined tuples for a tuple $a_0$ of $A^Q$

called and would incur the two selections denoted by dotted arrows in Fig. 9. The right part of Fig. 9 shows the created $\mathcal{T}$ for the tuple $w_0$, where tuples in the dashed rectangle are queried in the dashed arrows and tuples in the dotted rectangle are queried in $EvalPath$.

Obviously, there is an obvious tradeoff between caching large amount of data and frequent evaluation of small joins. In the following, we will analyze the upper bound of the cached data theoretically. And then in the experimental section, we will study the practical effect on a real data set. $\mathcal{T}$ is created on-the-fly, i.e., along the execution of procedures $Insert$ and $EvalPath$, and its depth is determined by the recursion depths of them. Therefore, $\mathcal{T}$ is not complete in Fig. 9. The maximum recursion depth of procedure $Insert$ is $\left\lceil \frac{CN\max}{2} \right\rceil$ [32]. And the recursion depth of procedure $EvalPath$ is $\left\lfloor \frac{CN\max}{2} \right\rfloor$. Hence, the height of $\mathcal{T}$ is bounded by $CN_{\max}$. If we use $M_1$ and $M_2$ to indicate the maximum number of adjacent relations that each relation $R_i$ can have and the maximum number of tuples that a tuple of $R_i$ can join in its adjacent relations, respectively. Note that $M_1$ and $M_2$ are often rather small compared to the number of CNs. Then in the worst case, there are tuples of $M_1^l$ relations in level $l$ (the level of the root is 0) of $\mathcal{T}$; and for each relation, there are $M_2^l$ tuples. Hence, the total number of tuples in $\mathcal{T}$ is:

$$O\left(\sum_{l=1}^{CN\max-1} M_1^l \cdot M_2^l\right) = O\left((M_1 \cdot M_2)^{CN\max}\right). \quad (8)$$

In a relational database, can have a large value. For example, in a bibliographic database, a conference or a journal tuple can be referenced by a large number of paper tuples. Hence, Eq. (8) can have a huge result, which makes the efficiency of the method of caching worse than KDynamic's method of frequent evaluation of small joins. Fortunately, $\mathcal{T}$ is quite incomplete for the following two reasons. First, merely finding the top-$k$ results cannot make a large number of JTTs to be found; hence, the recursion depths of *Insert* and *EvalPath* are rather small for most of the tuples, otherwise the joining process would be activated and lots of JTTs could be found. Second, the possibility of the joined tuples of a tuple $t$ that can be found in the processed tuples of $R_i^Q$ can be approximated as $M_2 \cdot IDF \cdot \alpha$, where $IDF$ denotes $\frac{df_w}{N}$ (the ratio of the number of matched tuples to the number of total tuples in $R_i$), and $\alpha$ is the percentage of processed tuples in $R_i^Q$. $IDF$ is small for most keywords in a relational database ($\ll 0.1$), and $\alpha$ is small for most lattice nodes and its average value is about 0.1 in our experiments. Hence, the processing of the filter phase stops at a lattice node $R_i^Q$ in most cases. Therefore, in fact, our method does not need to cache large amounts of data in $\mathcal{T}$.

For a tuple $t$, we use $\pi$ to denote the number of lattice nodes for which the procedure *Insert* and *EvalPath* are called for $t$. In the best case, we only need to query the database at the first time of calling $Insert$ or $EvalPath$, and then the cached tuples in $\mathcal{T}$ can be reused. Since the majority of the database access cost is in executing *Insert* or *EvalPath*, in the best case, the total time cost of handling $t$ can be reduced to $\frac{1}{\pi}$, compared to KDynamic.

In the implementation, the lattice nodes of the same $R_i^{Q\,or\,F}$ shares the same tuple sets, but differs in the *cur* value and the output buffer. For caching the join relationships, each tuple is created one time at the memory and hash indexes are created for quickly locating of tuples. And then for each tuple, it stores the pointers of the joined tuples in other relations. Therefore, $\mathcal{T}$ trees rooted at different tuples can share the joining relationships to the maximum extent. As shown in the experimental results, caching the joined tuples can highly improve the efficiency both in computing the top-$k$ results.

## 6 Experimental study

### 6.1 Datasets

We conducted extensive experiments to test the efficiency of our method. We used the DBLP dataset[4] and the IMDB dataset.[5] DBLP is a continuously growing international bibliography database which mainly focus on the computer science. IMDB is an online database of information related to films, television programs, and video games. These two databases are used in many studies on keyword queries over relational databases, such as [2,3].

The two downloaded XML files are decomposed into relations according to the two schemas shown in Figs. 10 and 11, respectively. The two arrows from *PaperCite* to *Papers* denote the foreign-key-references from *paperID* to *paperID* and *citedPaperID* to *paperID*, respectively. MySQL (v5.6) is used as the RDBMS with the default "Dedicated MySQL Server Machine" configuration. All the relations use the MyISAM storage engine. Indexes are built on all primary

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

[5] http://www.imdb.com.

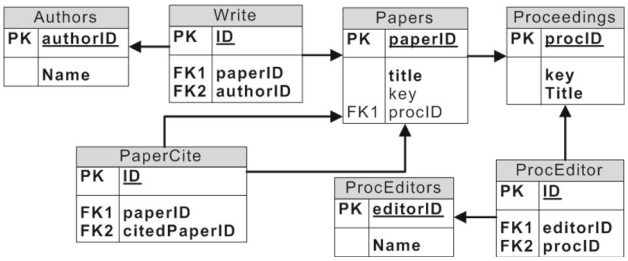



**Fig. 10** The DBLP schema (PK stands for primary key, FK for foreign key)

**Table 2** Tuple numbers in DBLP

| Papers | PaperCite | Write | Authors |
|---|---|---|---|
| 764,403 | 38,675 | 1,678,379 | 641,368 |
| Proceeding | ProcEditers | | ProcEditer |
| 6926 | 12,310 | | 16,351 |

**Table 3** Tuple numbers in IMDB

| Actors | Actresses | Directors | Movie |
|---|---|---|---|
| 983,135 | 993,398 | 189,652 | 189,639 |
| Movielinks | Person | ProdComp | ProdComps |
| 497,913 | 357,379 | 471,275 | 90,478 |

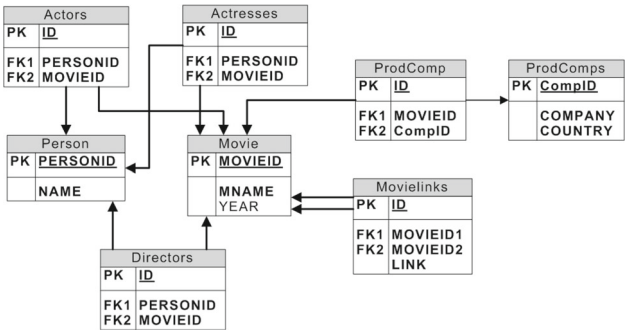



**Fig. 11** The IMDB schema (PK stands for primary key, FK for foreign key)

key and foreign key attributes, and full-text indexes are built for all text attributes. The tuple numbers of the relations in the two databases are listed in Tables 2 and 3. And the total sizes of the two databases, including the indexes, are 368 and 436 MB, respectively. All the algorithms are implemented in C++. We conducted all the experiments on a PC with a 3.0 GHz CPU and 16 GB memory, running Windows 7.

### 6.2 Parameters

We use the following five parameters in the experiments: (1) $k$: the top-$k$ value; (2) $l$: the number of keywords in a query; (3) $IDF$: the $\frac{df_w}{N}$ value of Eq. (3); (4) $CN_{\max}$: the maximum size of the generated CNs; (5) $Kmean$: the number of clusters of CNs; and (6) $step$: the number of tuples being processed one time in Algorithm 1. When $k$ grows, the cost of computing the top-$k$ results increases since we need to find more results. Different values of $CN_{\max}$ can dramatically affect the time of computing top-$k$ results since the number of CNs can grow exponentially when $CN_{\max}$ increases. And the number of related tuples increases when $IDF$ and $l$ getting bigger. Therefore, the above four parameters can highly indicate the scalability of a top-$k$ keyword search system, and are widely adopted at previous studies [6,7].

**Table 4** Parameters (DBLP)

| Names | Values |
|---|---|
| $k$ | 100, **200**, 250, 300 |
| $I$ | 2, **3**, 4, 5 |
| $IDF$ | 0.003, 0.007, **0.013**, 0.025 |
| $CN_{\max}$ | 4, 5, **6**, 7 |
| $Kmean$ | 1, 3, 5, 10, 20, **30**, 40, MAX |
| $step$ | 1, 50, 100, **200**, 300, 400, 800 |

**Table 5** Parameters (IMDB)

| Names | Values |
|---|---|
| $k$ | 200, **300**, 400 |
| $I$ | 2, **3**, 4, 5 |
| $CN_{max}$ | 4, **6**, 7 |
| $Kmean$ | 1, 5, 20, 30, **40**, MAX |
| $step$ | 1, 50, 100, 200, **300**, 400, 800 |

The parameters with their default values (in bold) are shown in Tables 4 and 5. Due to the same sets of CNs when $CN_{\max}$ are 5 and 6, the $CN_{\max}$ values in IMDB experiments do not contain 5. Because the keywords in IMDB are not distributed regularly as in DBLP, we cannot choose a set of $IDF$ values. The numbers of the generated CNs for the different $CN_{\max}$ values are 18, 54, 134 and 336 for the DBLP database; and are 8, 64, 558 for the IMDB database. The keywords selected are listed in Tables 6 and 7 with their $IDF$ values, where the keywords in bold fonts are keywords popular in author or person names.

We run the Algorithm 1 on different values of each parameter while keeping the other five parameters in their default values. Ten top-$k$ queries are selected for each combinations of parameters. To avoid generating a small number of CNs for each query, one author name keyword of each $IDF$ value always be selected for each query. In the experiments, two main metrics are considered: the average time cost ($Time$) and number of computed JTTs ($\#R$).

**Table 6** Keywords and their *IDF* values (DBLP)

| Keywords | *IDF* |
|---|---|
| ATM, collaboration, cluster, Java, navigation, ontology, privacy, QoS, scalable, Spatial **Charles, Eric** | 0.004 |
| Embedded, fuzzy, genetic, Intenet, machine, mining, semantic, sensor, video, XML, **James, Zhang** | 0.007 |
| Adaptive, architecture, database, evaluation, mobile, oriented, optimization, process, security, simulation, wireless, **John, Wang** | 0.013 |
| Algorithm, design, distributed, information, learning, networks, performance, software, time, web, **David, Michael** | 0.025 |

**Table 7** Keywords and their *IDF* values (DBLP)

| Keywords | *IDF* |
|---|---|
| Black, blue, cinema, company, corporation, entertainment, girl, gmbh, group, international, life, little, love, media, pictures, production, story, studio, television, video, world, **David, George, James, John, Michael, Paul, Peter, Richard** | Default |

When $k$ grows, the cost of computing the initial top-$k$ results increases since we need to find more results, and the cost of maintaining the top-$k$ results also increases because the lattice nodes have more outputted tuples since more tuples are processed.

### 6.3 Main results

In this section, we will describe the main experimental results of the two databases, respectively.

#### *6.3.1 DBLP dataset*

First, we want to study the effectiveness of the four optimization methods proposed in Sect. 5. Figure 12 shows the varying of the average times of computing the top-$k$ results when changing parameter *Kmean*, on different $CN_{\max}$ values, where $Kmean = MAX$ means that all the CNs are evaluated separately. Results of four different algorithms are shown in Fig. 12, where *LP* denotes the Algorithm 1, *LP-F* denotes Algorithm 1 with the tuple filtering method, *LP-C* denotes Algorithm 1 with the caching joined tuples method, and *LP-*$FO_{0,1}$ denotes Algorithm 1 with the tuple filtering method and the optimization of lattice construction method, where $O_{0,1}$ indicate $\{a, b\} = \{0, 1\}$ at Eq. (7). Since the results of the $K$-means clustering may be affected by the starting condition [43], for each $Kmean$ value, we run each algorithm five times on different starting condition for each keyword query and report the average result. We have experimented with multiple combinations of values for $a$ and $b$ and found that $\{a, b\} = \{0, 1\}$ can always result in the optimal performance, which means that the popularity of the rooted subtrees should be omitted. It worth noting that we do not report the performance of Algorithm 1 lonely with the optimization of

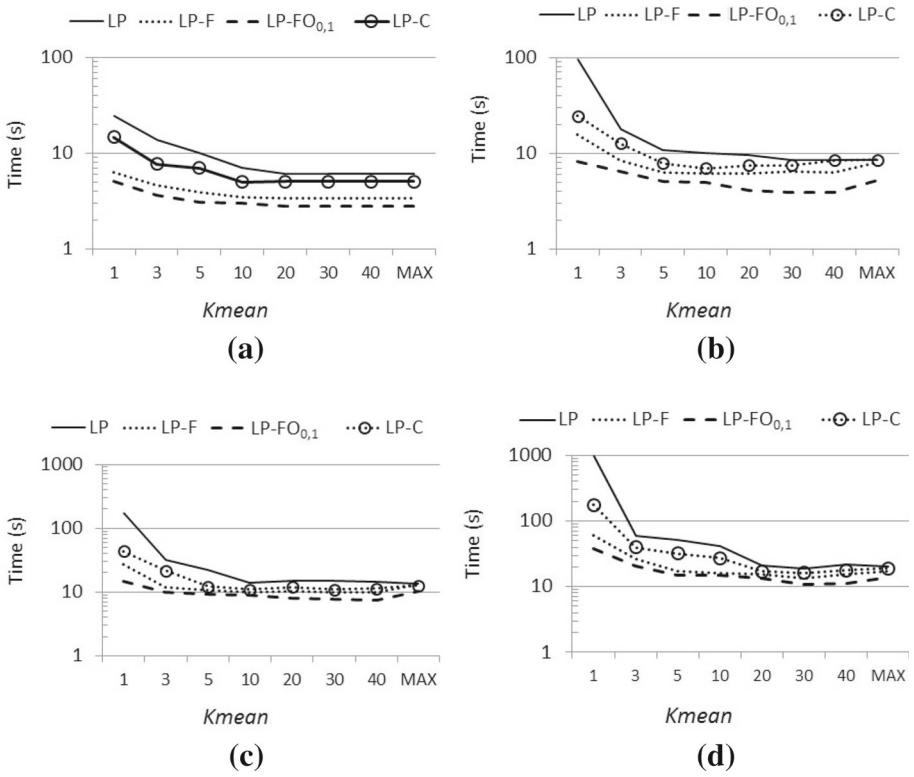



**Fig. 12** The effectiveness of the four optimization methods in DBLP. **a** $CN_{max} = 4$, **b** $CN_{max} = 5$, **c** $CN_{max} = 6$, **d** $CN_{max} = 7$

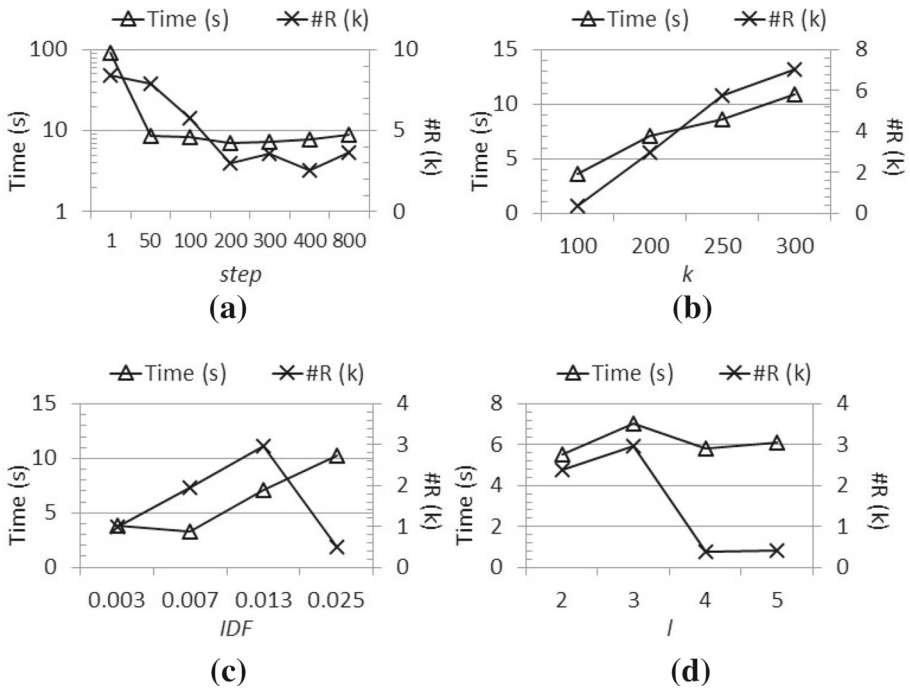



**Fig. 13** The effects of $step$, $k$, $IDF$ and $l$ on the two measures in DBLP. **a** Varying $step$, **b** Varying $k$, **c** Varying $IDF$, **d** Varying $l$

lattice construction method because the curve would be very close to *LP-C* and then makes the figures hard to read.

We can find the remarkable efficiency improvements of Algorithm 1 caused by methods of tuple filtering and optimization of lattice construction in Fig. 12, especially when $Kmean$ has small values. Figure 12 also shows that CN clustering can significantly improve the efficiency of computing top-$k$ results, which justifies the two heuristic rules proposed in Sect. 5.2. The efficiencies of the four algorithms compared in Fig. 12 are affected differently by the variance of $Kmean$. For the LP algorithm, the time needed to compute top-$k$ results is the least when $Kmean = MAX$. But for the other three algorithms, the times experience a rise after the first decline when $Kmean$ changing from 1 to $MAX$, which can always reach the lowest point near $Kmean = 30$. Therefore, $Kmean$ is set to 30 at the following experiments at the DBLP dataset. Such an optimal $Kmean$ value is depending on the schema of the dataset, which will be 40 at the IMDB database. As a conclusion, the two methods of tuple filtering and CN clustering can mostly increase the efficiency of Algorithm 1 in computing top-$k$ results.

Next, we want to learn the effects of the parameters and the scalability of the proposed methods. Figure 13 shows how the two measures change of LP algorithm plus the four optimization methods while varying $step$, $IDF$, $k$ and $l$, where the values of $\#R$ are all plotted on the right $Y$-axis and the unit is 1000. Figure 13a shows that the two measures all decline rapidly when $step$ is growing from 1 to 200. Decreasing of $Time$ is due to the highly reduced number of database accesses, which proves the importance of processing tuples in batches (or blocks as proposed in

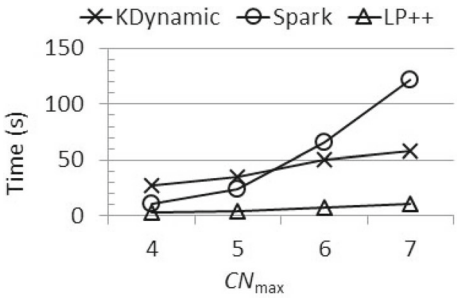



**Fig. 14** LP++ versus *KDynamic* and *SPARK* in DBLP

SPARK [3]). Because $\overline{score}(V_i^Q, Q)$ is computed using the first un-processed tuple, larger values of $step$ can result in more un-necessary tuple processing at node $V_i^Q$. Hence, as can be seen from Fig. 13a, $\#R$ increases while $step$ grows, and the time cost increases while $step$ grows from 200 to 400. In practice, we recommend $k < step < 2k$. The curves of $Time$ and $\#R$ at Fig. 13b–d do not have fast-rising while increasing $k$, $IDF$ and $l$, which imply the good scalability of the proposed method. More importantly, the curves of $\#R$ at Fig. 13c, d changes from rising to falling while $IDF$ increasing from 0.013 to 0.025, and $l$ increasing from 3 to 5. This is because when $IDF$ and $l$ have large values, single tuples would have high probabilities to contain more than one keywords, hence large relevance scores. Therefore, there are more JTTs that have high relevance scores, which results in larger $\theta$ and small values of the two measures.

Figure 14 compares the time cost of computing the top-$k$ results of Algorithm 1 with the four proposed optimization methods, denoted by "LP++", with that of the *BP* algorithm

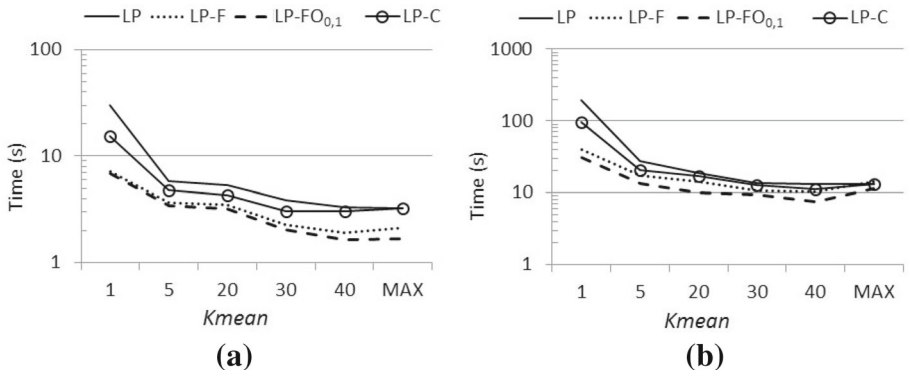



**Fig. 15** The effectiveness of the four optimization methods in IMDB. **a** $CN_{max} = 6$, **b** $CN_{max} = 7$

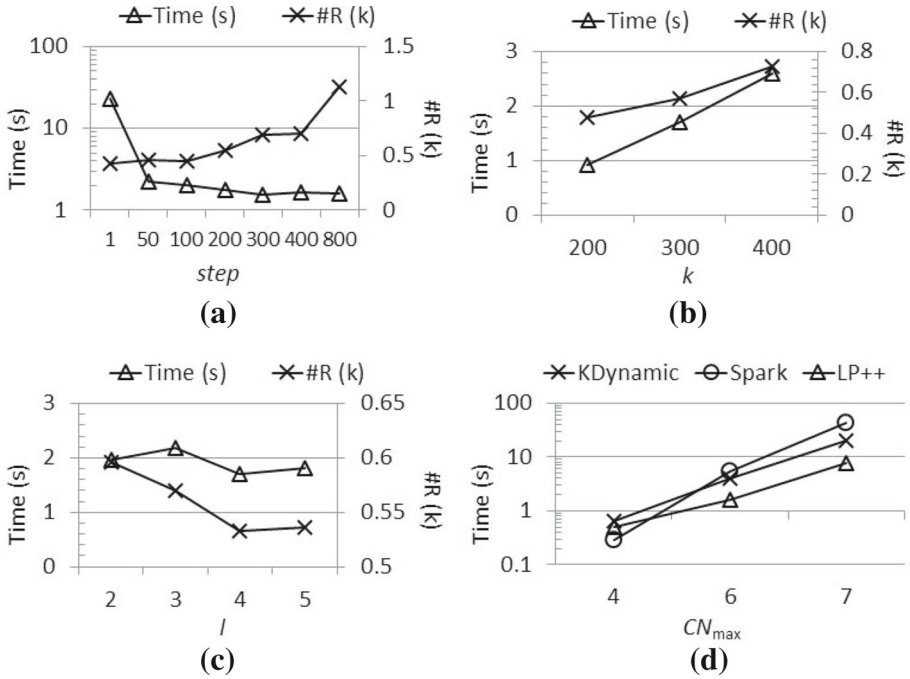



**Fig. 16** The effects of *step, k, l* and $CN_{max}$ on the two measures in IMDB. **a** Varying *step*, **b** varying *k*, **c** varying *l*, **d** varying $CN_{max}$

of SPARK (which is the state-of-art top-$k$ keyword search algorithm [9]) and KDynamic, respectively, while varying $CN_{\max}$. Figure 14 shows that, compared to SPARK, algorithms LP++ and KDynamic are more efficient in finding the top-$k$ results, because evaluating the CNs using the lattice can achieve complete reduction since all the output tuples of the root nodes can form JTTs [32]. The time costs of KDynamic in Fig. 14 are all obtained when $Kmean = 30$. Hence, the difference between our approach and KDynamic reflects the effects of the other three optimization methods. More importantly, the improvement increases as $CN_{\max}$ grows.

### 6.3.2 IMDB dataset

Figure 15 shows the varying of the average times of computing the top-$k$ results when changing parameter $Kmean$, where the four algorithms are some as the experiments at the DBLP dataset. Because the number of CNs generated is very small, Fig. 15 does not report the data when $CN_{\max} = 4$. We can see the same effectiveness of the four optimization methods in Fig. 15 as in Fig. 12, and the two methods of tuple filtering and CN clustering also prove the largest improvements to the efficiency of LP algorithm. The only difference is that the lowest points of the curves are reached near $Kmean = 40$.

Figure 16 shows the effects of parameters of $step$, $k$, $l$ and $CN_{\max}$ to the two measures of the proposed method, and compares with the BP algorithm of SPARK and KDynamic at Fig. 16d. Figure 16a reveals that the optimal value of $step$ in IMDB is 300. The value of $Time$ decreases quickly while $step$ increasing from 1 to 300, and then growing slowing while $step$ increasing from 300. The value of $\#R$ keeps growing bigger while $step$ increasing. However, in the DBLP experiments, as shown in Fig. 13a, $\#R$ shows irregular fluctuations after $step$ is bigger than the optimal value 200. This is because the foreign key reference relationships are more extensively in the DBLP database than in the IMDB database. As a result, the number of JTTs are much bigger in the DBLP

database, which can be shown by the different ranges of $\#R$ values in Figs. 13 and 16. Therefore, when using a bigger $step$ value, it is more probably to find JTTs with large relevance scores in DBLP, which would increase $top\text{-}k[k] \cdot score$ more quickly. In comparison, due to the small number of JTTs in the IMDB database, different values of $step$ cannot affect the changing process of $top\text{-}k[k] \cdot score$. Hence, more tuples of small $\overline{score}(V_i^Q \cdot t,\ Q)$ values will be processed as $step$ growing, which will result in more JTTs.

From Fig. 16b, c we can observe the similar changes of the two measures when varying $k$ and $l$, respectively. And from the comparisons of the algorithms in Fig. 16d, we can see the well effectiveness of the proposed optimization methods and the good scalability of the proposed algorithm. In summary, compared to the existing proposed algorithms, the efficiency of computing the top-$k$ keyword search results has been improved by an order of magnitude in this paper. And the average time needed for a top-$k$ keyword search are smaller than 10 s in our experiments. Therefore, the proposed methods in this paper make the top-$k$ search in relational databases to be more practicable.

## 7 Conclusion

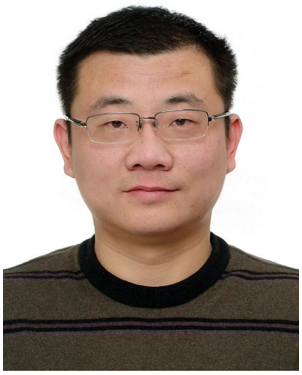
In this paper, we studied the problem of answer continuous top-$k$ keyword query in relational databases. We proposed to store the state of the CN evaluation process, which can be used to restart the query evaluation after the insertion of new tuples. An algorithm was presented to maintain the top-$k$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

**Acknowledgements** This research was supported by the Natural Science Foundation of Shanghai under Grant No. ~14ZR1427700 and the Shanghai Engineering Research Center for Broadband Technologies and Applications (14DZ2280100).

## References

1. Yu, J.X., Qin, L., Chang, L.: Keyword Search in Relational Databases: A Survey. In: Bulletin of the IEEE Technical Committee on Data Engineering, vol. 33, no. 10 (2010)
2. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-style keyword search over relational databases. In: VLDB, pp. 850–861 (2003)
3. Luo, Y., Lin, X., Wang, W., Zhou, X.: SPARK: top-k keyword query in relational databases. In: ACM SIGMOD, pp. 115–126 (2007)
4. Luo, Y., Wang, W., Lin, X., Zhou, X., Wang, J., Li, K.: SPARK2: top-k keyword query in relational databases. IEEE Trans. Knowl. Data Eng. **23**(12), 1763–1780 (2011)
5. Hristidis, V., Papakonstantinou, Y.: DISCOVER: keyword search in relational databases. In: VLDB, pp. 670–681 (2002)
6. Markowetz, A., Yang, Y., Papadias, D.: Keyword search on relational data streams. In: ACM SIGMOD, pp. 605–616 (2007)
7. Qin, L., Yu, J.X., Chang, L., Tao, Y.: Scalable keyword search on large data streams. In: ICDE, pp. 1199–1202 (2009)
8. Xu, Y., Guan, J., Ishikawa, Y.: Scalable top-k keyword search in relational databases. In: Database Systems for Advanced Applications—17th International Conference, DASFAA 2012, Proceedings, Part II, Busan, South Korea, 15–19 April 2012, pp. 65–80 (2012)
9. Yu, J.X., Qin, L., Chang, L.: Keyword Search in Databases, Synthesis Lectures on Data Management. Morgan and Claypool Publishers, San Rafael (2010)
10. Xu, Y., Ishikawa, Y., Guan, J.: Efficient continuous top-k keyword search in relational databases. In: WAIM, pp. 755–767 (2010)
11. Luo, Y.: SPARK: a keyword search system on relational databases. PhD Thesis, The University of New South Wales (2009)
12. Cheng, S., Termehchy, A., Hristidis, V.: Predicting the effectiveness of keyword queries on databases. In: CIKM, pp. 1213–1222 (2012)
13. Liu, F., Yu, C., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. In: ACM SIGMOD, pp. 563–574 (2006)
14. Bergamaschi, S., Ferro, N., Guerra, F., Silvello, G.: Keyword-based search over databases: a roadmap for a reference architecture paired with an evaluation framework. Trans. Comput. Collect. Intell. **21**, 1–20 (2016)
15. Zhang, J., Peng, Z., Wang, S., Nie, H.: CLASCN: candidate network selection for efficient top-$k$ keyword queries over databases. J. Comput. Sci. Technol. **22**(2), 197–207 (2007)
16. Aditya, B., Bhalotia, G., Chakrabarti, S., Hulgeri, A., Nakhe, C., Parag, P.: BANKS: browsing and keyword searching in relational databases. In: VLDB, pp. 1083–1086 (2002)
17. He, H., Wang, H., Yang, J., Yu, P.S.: BLINKS: ranked keyword searches on graphs. In: ACM SIGMOD, New York, NY, USA, pp. 305–316. ACM (2007)
18. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In: VLDB, pp. 505–516 (2005)
19. 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, pp. 903–914 (2008)
20. Li, G., Zhou, X., Feng, J., Wang, J.: Progressive keyword search in relational databases. In: ICDE, pp. 1183–1186 (2009)
21. Qin, L., Yu, J.X., Chang, L., Tao, Y.: Querying communities in relational databases. In: ICDE, pp. 724–735 (2009)
22. Li, G., Feng, J., Zhou, X., Wang, J.: Providing built-in keyword search capabilities in RDBMS. VLDB J. **20**(1), 1–19 (2011)
23. Lopez-Veyna, J.I., Sosa, V.J.S., Lopez-Arevalo, I.: KESOSD: keyword search over structured data. In: KEYS, pp. 23–31 (2012)
24. Kargar, M., An, A.: Efficient top-k keyword search in graphs with polynomial delay. In: ICDE, pp. 1269–1272 (2012)
25. Ling, T.W., Le, T.N., Zeng, Z.: Towards an intelligent keyword search over XML and relational databases. In: International Conference on Big Data and Smart Computing, BIGCOMP 2014, Bangkok, Thailand, 15–17 January 2014, pp. 1–6 (2014)
26. Torlone, R.: Towards a new foundation for keyword search in relational databases. In: Proceedings of the 8th Alberto Mendelzon Workshop on Foundations of Data Management, Cartagena de Indias, Colombia, 4–6 June 2014 (2014)
27. Lin, Z., Li, Y., Lai, Y.: Improve the effectiveness of keyword search over relational database by node-temperature-based ant colony optimization. In: 12th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2015, Zhangjiajie, China, 15–17 August 2015, pp. 1209–1214 (2015)
28. Ling, T.W., Zeng, Z., Le, T.N., Lee, M.: ORA-semantics based keyword search in XML and relational databases. In: 32nd IEEE

International Conference on Data Engineering Workshops, ICDE Workshops 2016, Helsinki, Finland, 16–20 May 2016, pp. 157–160 (2016)
29. Yu, Z., Yu, X., Chen, Y., Ma, K.: Distributed top-k keyword search over very large databases with MapReduce. In: 2016 IEEE International Congress on Big Data, San Francisco, CA, USA, 27 June–2 July 2016, pp. 349–352 (2016)
30. Park, J., Lee, S.: Keyword search in relational databases. Knowl. Inf. Syst. **26**(2), 175–193 (2011)
31. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: a system for keyword-based search over relational databases. In: ICDE, pp. 5–16 (2002)
32. Qin, L., Yu, J.X., Chang, L.: Scalable keyword search on large data streams. VLDB J. **20**(1), 35–57 (2011)
33. Baid, A., Rae, I., Li, J., Doan, A., Naughton, J.F.: Toward scalable keyword search over relational data. PVLDB **3**(1), 140–149 (2010)
34. Fakas, G.J.: A novel keyword search paradigm in relational databases: object summaries. Data Knowl. Eng. **70**(2), 208–229 (2011)
35. Xu, Y., Ishikawa, Y., Guan, J.: Efficient continual top-k keyword search in relational databases. J. Inf. Process. **20**(1), 1–14 (2012)
36. Zeng, Z., Bao, Z., Ling, T.W., Lee, M.L.: iSearch: an interpretation based framework for keyword search in relational databases. In: KEYS, pp. 3–10 (2012)
37. Xu, Y., Guan, J., Li, F., Zhou, S.: Scalable continual top-k keyword search in relational databases. Data Knowl. Eng. **86**, 206–223 (2013)
38. de Oliveira, P., da Silva, A.S., de Moura, E.S.: Ranking candidate networks of relations to improve keyword search over relational databases. In: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, 13–17 April 2015, pp. 399–410 (2015)
39. Zeng, Z., Bao, Z., Lee, M., Ling, T.W.: Towards an interactive keyword search over relational databases. In: Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Companion Volume, Florence, Italy, 18–22 May 2015, pp. 259–262 (2015)
40. Kargar, M., An, A., Cercone, N., Godfrey, P., Szlichta, J., Yu, X.: Meaningful keyword search in relational databases with large and complex schema. In: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, 13–17 April 2015, pp. 411–422 (2015)
41. Zhou, J., Liu, Y., Yu, Z.: Improving the effectiveness of keyword search in databases using query logs. In: Web-Age Information Management—16th International Conference, WAIM 2015, Proceedings, Qingdao, China, 8–10 June 2015, pp. 193–206 (2015)
42. Luo, Y., Wang, W., Lin, X.: SPARK: a keyword search engine on relational databases. In: ICDE, pp. 1552–1555 (2008)
43. Lloyd, S.P.: Least squares quantization in PCM. IEEE Trans. Inf. Theory **28**, 129–136 (1982)



**Yanwei Xu** received his Bachelor Degree of Mathematics and Master Degree of Computer Science from Shandong Normal University in 2004 and 2007. He received his Ph.D. Degree in Computer Science from Tongji University of China in 2012. His main research interests lie in keyword search in relational databases, information retrieval and algorithm complexity analysis.