The VLDB Journal (2011) 20:35–57
DOI 10.1007/s00778-010-0190-x

REGULAR PAPER

Scalable keyword search on large data streams

Lu Qin · Jeffrey Xu Yu · Lijun Chang

Received: 2 June 2009 / Revised: 10 April 2010 / Accepted: 12 April 2010 / Published online: 19 May 2010
© Springer-Verlag 2010

Abstract It is widely recognized that the integration of information retrieval (*IR*) and database (*DB*) techniques provides users with a broad range of high quality services. Along this direction, *IR*-styled m-keyword query processing over a relational database in an RDBMS framework has been well studied. It finds all hidden interconnected tuple structures, for example connected trees that contain keywords and are interconnected by sequences of primary/foreign key relationships among tuples. A new challenging issue is how to monitor events that are implicitly interrelated over an open-ended relational data stream for a user-given m-keyword query. Such a relational data stream is a sequence of tuple insertion/deletion operations. The difficulty of the problem is related to the number of costly joins to be processed over time when tuples are inserted and/or deleted. Such cost is mainly affected by three parameters, namely, the number of keywords, the maximum size of interconnected tuple structures, and the complexity of the database schema when it is viewed as a schema graph. In this paper, we propose new approaches. First, we propose a novel algorithm to efficiently determine all the joins that need to be processed for answering an m-keyword query. Second, we propose a new demand-driven approach to process such a query over a high speed relational data stream. We show that we can achieve high efficiency by significantly reducing the number of intermediate results when processing joins over a relational data stream. The proposed new techniques allow us to achieve high scalability in terms of both query plan generation and query plan execution. We conducted extensive experimental studies using synthetic data and real data to simulate a relational data stream. Our approach significantly outperforms existing algorithms.

Keywords Keyword search · Relational databases · Data streams

L. Qin · J. X. Yu (✉) · L. Chang
The Chinese University of Hong Kong, Hong Kong, China
e-mail: yu@se.cuhk.edu.hk

L. Qin
e-mail: lqin@se.cuhk.edu.hk

L. Chang
e-mail: ljchang@se.cuhk.edu.hk

1 Introduction

The *IR*-styled m-keyword query processing in RDBMS has been recently studied [1,6,9,12,14,16,18,20,25,26]. Such m-keyword queries provide users with the insights on the interconnected tuple structures in an RDB that the users cannot find easily. Given m-keywords, $K = \{k_1, k_2, \ldots, k_m\}$, one of such interconnected tuple structures is connected trees. A connected tree shows how the tuples, which contain keywords, are interconnected by sequences of primary/foreign key relationships among tuples. Such interconnected structures are hidden among relations in an RDB. Consider an E-Commerce application; it is highly desirable to find out what products are interested by customers in a certain area and what other products are related. For example, given two keywords, "Dress" and "Texas". One of many connected trees may show that a customer in Texas ordered a product and the product appears in a package that contains the keyword of Dress. The tuples may appear in different relations and are connected by primary/foreign key relationships. And all connected trees found possibly have different structures (or different inter-connectivity).

In the literature, the focus is to find such interconnected tuple structures in a static RDB. However, decision-makers may care how their customer order pattern changes over time. As an example, customers in a certain area may change their

patterns of ordering fashion design products which come from different vendors over time. Such needs stem from real applications where users need to monitor important or interesting events that are not explicitly related over a large amount of data that come as a relational data stream. Here, a relational data stream is a sequence of tuple insertion/deletion operations. Such a relational data stream can be generated by either sensors or RFID-powered equipments or appears as a stream that merges different streams from different sites in a distributed RDBMS. It is infeasible to process an m-keyword query whenever a tuple is inserted or deleted due to the high computational cost.

Markowetz, Yang and Papadias studied m-keyword query processing over a relational data stream [26]. Given a database schema as a graph $G_S(V, E)$ where there are primary/foreign key relationships among relations, an m-keyword query finds all connected trees via primary/foreign key references among tuples that co-exist in a sliding window over a relational data stream. The hardness is that it needs to generate a query plan that needs a large number of joins to process an m-keyword query on an RDBMS. Consider a database schema of $|V|$ relations with reasonable primary/foreign key references. It needs to generate a plan that includes up to $|V| \cdot 2^m$ temporal/base relations for handling all the possible subsets of m-keywords that may appear in any possible way and in any relation over a data stream. The cost of generating such a join plan is high. As reported in [26], it may take hours to generate a plan. It is worth noting that the number of joins is also heavily affected by the maximum size of connected trees a user prefers to observe. The larger the size, the harder to process. In addition, the join processing itself is expensive and the low efficiency is caused by computing a large number of intermediate results that are not eventually used to form any connected trees. Consider an example, $(R \bowtie S) \bowtie T$. Suppose that there are many incoming tuples for R and S, the intermediate results can be very large. However, the intermediate results of $R \bowtie S$ may not be able to join any T tuple. The costs to join and to maintain all the intermediate results are very high.

The main contributions in this paper are summarized below. First, we propose a novel template-based approach to generate all the join plans efficiently. Our approach is not affected by the complexity of the database schema and can handle a larger number of keywords and a larger maximum size of connected trees. Second, we propose a new join processing approach. We attempt to avoid using joins directly, and instead process m-keyword queries using selection/semi-join [5] to fully reduce the number of intermediate results first, followed by joins. In other words, we only maintain those tuples that can be possibly joined to output some connected trees. We conducted extensive performance studies and confirmed that our approach significantly outperforms the existing approaches.

The remainder of the paper is organized as follows. Sect. 2 gives problem statement and discusses the main factors that make this problem challenging. We discuss an up-to-date existing work and address the computational costs that need to be improved in handling scalability in Sect. 3. We give an overview on our new approach in Sect. 4 and discuss the details on *CN* generation and *CN* evaluation in Sect. 5 and Sect. 6, respectively. Experimental studies are given in Sect. 7 followed by discussions on related work in Sect. 8. Finally, Sect. 9 concludes the paper.

2 Problem definition

We consider a database schema in a relational database as a directed graph $G_S(V, E)$, called a schema graph, where V represents the set of relation schemas $\{R_1, R_2, \ldots\}$ and E represents foreign key references between two relation schemas. Given two relation schemas, R_i and R_j, there exists an edge in the schema graph, from R_i to R_j, denoted $R_i \rightarrow R_j$, if the primary key defined on R_i is referenced by the foreign key defined on R_j. Parallel edges may exist in G_S if there are several foreign keys defined on R_j referencing to the primary key defined on R_i. To distinguish one foreign key reference among many, we use $R_i \stackrel{X}{\rightarrow} R_j$, where X is the foreign key attribute names. A relation on relation schema R_i is an instance of the relation schema (a set of tuples) conforming to the relation schema, denoted $r(R_i)$. A tuple can be inserted into a relation and deleted from a relation. Below, we use R_i to denote $r(R_i)$ if the context is obvious. And we use $V(G_S)$ and $E(G_S)$ to denote the set of nodes and the set of edges of G_S, respectively.

Example 2.1 A simple E-Commerce database schema, G_S, is shown in Fig. 1. It consists of four relation schemas: `Customer`, `Order`, `Product` and `Group`. `Customer` has four attributes and the primary key is defined on `CID`. `Product` has four attributes and the primary key is defined on `PID`. `Order` has two foreign keys, `CID` (refer to the primary key defined on `Customer`) and `PID` (refer to the primary key defined on `Product`). `Group` specifies a package (part/subpart relationship) using two foreign keys, namely, `PID` and `SID`. `PID` is the product ID for a package and `SID` is the product ID for a subpart of the package. Both `PID` and `SID` refer to the primary key defined on the `Product` relation schema. The two foreign keys defined on `Group`

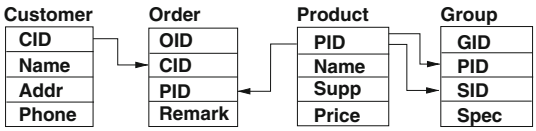


Fig. 1 A simple E-Commerce database schema

Customer

CID	Name	Addr	Phone
c1	Andy	Iowa	312425
c2	Miki	Texas	534345
c3	Mike	Iowa	246762
c4	Jim	Iowa	562354
c5	Kate	Texas	482724

Product

PID	Name	Supp	Price
p1	Trousers	AT Co.	$131
p2	Shirt	DBC	$35
p3	Suit	DS Ltd.	$138
p4	Dress T1	Null	$150
p5	Dress T2	Null	$320

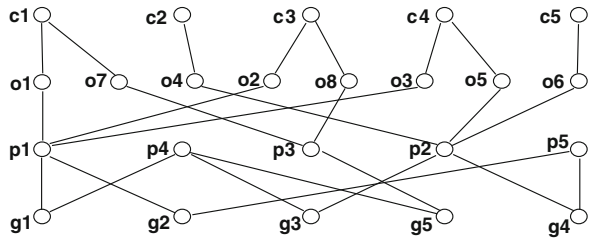
Order

OID	CID	PID	Remark
o1	c1	p1	XL
o2	c3	p1	Cheap
o3	c4	p1	Tight
o4	c2	p2	Null
o5	c4	p2	Expensive
o6	c5	p2	Null
o7	c1	p3	Null
o8	c3	p3	Too small

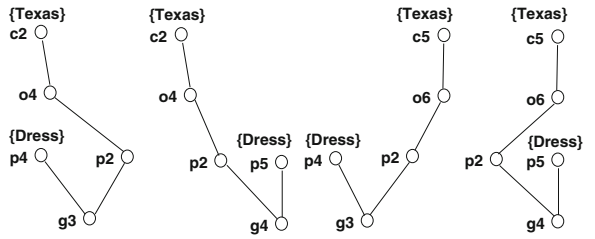
Group

GID	PID	SID	Spec
g1	p4	p1	White
g2	p5	p1	Null
g3	p4	p2	Large
g4	p5	p2	Null
g5	p4	p3	Blue

(a) Database



(b) Tuple Connections



(c) *MTJNT*s

Fig. 2 A simple E-Commerce database

referencing to the primary key on `Product` are parallel edges. A simple E-Commerce database is shown in Fig. 2. Figure 2a shows the four relations, and Fig. 2b illustrates the tuple connections based on primary/foreign key references. Below, we also use the initials of relation name to represent relation names.

An m-keyword query is a set of keywords of size m, $\{k_1, k_2, \ldots, k_m\}$. A result of an m-keyword query is a minimal total joining network of tuples, denoted *MTJNT* [16,26]. First, a joining network of tuples (*JNT*) is a connected tree of tuples where every two adjacent tuples, $t_i \in r(R_i)$ and $t_j \in r(R_j)$, can be joined based on the foreign key reference defined on relational schemas R_i and R_j in G_S (either $R_i \rightarrow R_j$ or $R_j \rightarrow R_i$). Second, by total, it means that a joining network of tuples must contain all the m keywords. Third, by minimal, it means that a joining network of tuples is not total if any tuple is removed. The minimal condition implies that every leaf tuple in the tree must contain at least one keyword. The size of a *MTJNT* is the number of nodes in the tree, and a user-given parameter Tmax is used to specify the maximum number of nodes in *MTJNT*s, in order to avoid a *MTJNT* to be too large in size, because it is not meaningful if two tuples are connected by a long chain of tuples.

Example 2.2 Consider the E-Commerce database shown in Fig. 2a. A simple 2-keyword query is {`Dress`, `Texas`}. Several *MTJNT*s of size 5 are shown in Fig. 2c.[1]

2.1 Keyword query processing over a data stream:

The problem of m-keyword query processing we study in this paper is to find **all** *MTJNT*s of size $\leq$ Tmax, for a given continuous m-keyword query, $\{k_1, k_2, \ldots, k_m\}$, on a schema graph G_S, over a high-speed large data stream, in the framework of RDBMS. It reports new *MTJNT*s when new tuples are inserted, and in addition, reports the existing *MTJNT*s that become invalid when tuples are deleted. A sliding window (time interval), W, is specified. A tuple, t, has lifespan since it is inserted from time $t.start$ to $W + t.start - 1$, if t is not deleted before then. Two tuples can be joined if their lifespans overlap.

In the framework of RDBMS, the two main steps of processing an m-keyword query over a graph schema G_S are candidate network generation and candidate network evaluation.

- In the first candidate network generation step, it generates a set of candidate networks over G_S, denoted $\mathcal{C} = \{C_1, C_2, \ldots\}$, to be evaluated in the second step. In brief, a *candidate network* (*CN*), C_i, corresponds to a relational algebra that joins a sequence of relations with selections of tuples for keywords over the relations involved. The set of *CN*s shall be sound/complete and duplicate free. The former ensures all *MTJNT*s must be found, and the latter is mainly for efficiency consideration.
- In the second candidate network evaluation step, all $C_i \in \mathcal{C}$ generated will be evaluated over a high-speed data stream dynamically.

The main factors that make the problem challenging are the graph complexity (the number of nodes/edges in the database schema G_S, $|G_S|$), the number of keywords (m), the maximum size of *MTJNT* allowed (Tmax), and the time interval (W). The first three factors contribute to the number

[1] In this example, the *MTJNT*s are simply paths. In general, a *MTJNT* can be any shape of trees.

Algorithm 1 CNGen (Expanded Schema Node v_i)

```
1: Q ← ∅;
2: Tree C_first ← a tree of a single node v_i;
3: Q.enqueue(C_first);
4: while Q ≠ ∅ do
5:   Tree C' ← Q.dequeue();
6:   for all u ∈ V(G_X) do
7:     for all u' ∈ V(C') do
8:       if u can be legally added to u' then
9:         Tree C ← a tree by adding u as a child of u';
10:        if C is a CN then
11:          output C; break;
12:        if C has the potential of becoming a CN then
13:          Q.enqueue(C);
```

of CNs, $|\mathcal{C}|$, to be generated and therefore to be evaluated. Given an m-keyword and a large Tmax over a large database schema G_S, the number of CNs to be generated can be very large. The number of CNs increases exponentially while any $|G_S|$, m, or Tmax increases. We will discuss this issue in detail later. The last factor is mainly related to the number of tuples that need to be joined in sliding windows. We emphasize the fact that a new query processing mechanism is needed to significantly reduce the join processing cost for such a large number of CNs while tuples can be inserted/deleted over a high-speed data stream in a large sliding window.

3 The existing work

The m-keyword query processing over a relational data stream was first studied by Markowetz et al. in [26]. We highlight their contributions followed by the discussions on the further research issues regarding scalability.

3.1 The approach

First, in [26], a novel duplicate-free algorithm is proposed to generate all CNs, $\mathcal{C}$, in the candidate network generation step, based on an *expanded schema graph*, denoted G_X.G_X is conceptually constructed in [16,26]. Given a database schema, G_S, and a set of keywords of size m, $K = \{k_1, k_2, \ldots, k_m\}$. A node, R_i, in G_S, will be duplicated 2^m times, denoted $R_i\{K'\}$, in G_X, for any subset of keywords, $K' \subseteq K$. Here, $R_i\{K'\}$ represents a sub-relation of relation $r(R_i)$ that contains those tuples in $r(R_i)$ that only contain all keywords in K' and no other keywords, as defined in Eq. (1) [16].

$$R_i\{K'\} = \{t | t \in r(R_i) \wedge \forall k \in K', t \text{ contains } k \wedge \forall k \in (K - K'), t \text{ does not contain } k\} \quad (1)$$

An edge, $R_i\{K'\} \rightarrow R_j\{K''\}$, exists in G_X, if $R_i \rightarrow R_j$ exists in G_S.

Over G_X, a CN, C_i, is generated as a connected tree, where the number of nodes is less than or equal to Tmax, and the union of the keywords represented in every $R_j\{K'\}$ in C_i is K. The pruning rules given in *DISCOVER* [16] are used to prune CNs for m-keyword query processing over data stream: (Rule-1) duplicate CNs are pruned (based on tree isomorphism), (Rule-2) a CN can be pruned if there is a leaf node, $R_j\{K'\}$, where $K' = \emptyset$, because it will generate results that do not satisfy the minimal condition in *MTJNT*, and (Rule-3) when there only exists a single foreign key reference between two relation schemas, for example, $R_i \rightarrow R_j$, CNs including $R_i\{K_1\} \rightarrow R_j\{K_2\} \leftarrow R_i\{K_3\}$ will be pruned, where K_1, K_2, and K_3 are three subsets of K. The Rule-3 implies the fact that the primary key is defined on R_i and a tuple in the relation of $R_j\{K_2\}$ must reference to the same tuple appearing in both relations of $R_i\{K_1\}$ and $R_i\{K_3\}$. Based on Eq. (1), the same tuple cannot appear in two sub-relations, and the join results for $R_i\{K_1\} \bowtie R_j\{K_2\} \bowtie R_i\{K_3\}$ will be empty.

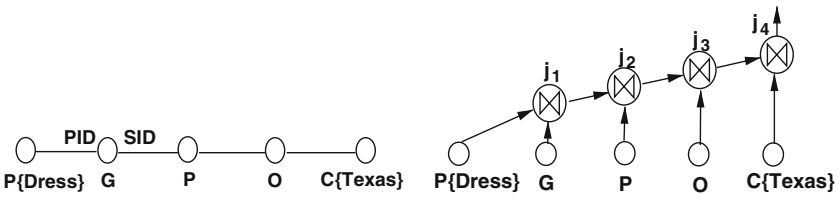
The algorithm in [26], called InitCNGen, assigns a unique identifier to every node in G_X and generates all CNs by iteratively adding more nodes to a temporary result in a pre-order fashion. It does not need to check duplications using tree isomorphism for those CNs where no node, $R_i\{K'\}$, appears more than once and can stop enumeration of CNs from a CN, C_i, if C_i can be pruned, because any CN C_j ($\supset C_i$) must also be pruned. The general algorithm InitCNGen is shown below.

1. for all nodes, n_i, in G_X containing k_l, ordered by node-id
2. CNGen (n_i);
3. remove n_i from G_X;

And the procedure CNGen is shown in Algorithm 1.

CNGen first initializes a queue $\mathcal{Q}$ and inserts a simple tree with only one node v_i into $\mathcal{Q}$ (line 1–3). It then iteratively expands a partial tree in $\mathcal{Q}$ by adding one node every time until $\mathcal{Q}$ becomes empty (line 4–13). In every iteration, a partial tree C' is removed from $\mathcal{Q}$ to be expanded (line 5). For every node u in G_X and u' in the partial tree C', it tests whether u can be legally added as a child of u'. Here, by "legally", it means (1) u' must be in the rightmost root-to-leaf path in the partial tree C'; (2) u must be no less than any of its siblings lexicographically and (3) if after adding u, the partial tree contains all the keywords, all the immediate subtrees for each node must be lexicographically ordered. The condition (3) can be checked by comparing the canonical code for each subtree in the partial tree.[2] If u can be legally added, then it adds u to a child of u' and forms a new tree C (line 8–9).

[2] A canonical code of a subtree is a string that can be generated by using the pre-order traversal of the subtree.



(a) *CN* **(b)** Left-deepTree

Fig. 3 A *CN* and its left-deep operator tree

If C itself is a *CN*, it outputs the tree. Otherwise, C will be added into $\mathcal{Q}$ for further expansion, if C has the potential of becoming a *CN*. Note that a partial tree C has the potential to become a *CN* if it satisfies two conditions: (1) the size of $\mathcal{Q}$ must be smaller than the size control parameter Tmax, and (2) every leaf node contains a unique keyword if it is not on the rightmost root-to-leaf path in C.

InitCNGen algorithm completely avoids three types of duplicates of *CN*s to be generated, comparing to the algorithm in *DISCOVER* [16]. The first is isomorphic duplicates between *CN*s generated from different roots. This is done by removing the root node from the expanded schema graph each time after calling CNGen. The second is duplicates that are generated from the same root following different insertion order for the remaining nodes. This is done by the second condition in the legal node testing (line 8). The third type of duplicates occurs when the same node appears more than once in a *CN*, this type of duplicates can be avoided by checking the third condition of the legal node testing (line 8). Avoiding the last two types of duplicates ensures that no isomorphic duplicates occurs for *CN*s generated from the same root. All together ensures that InitCNGen generates a complete and duplicate-free set of *CN*s.

Second, in [26], it processes a *CN* using a left-deep operator tree. The leaf nodes are $R_i\{K'\}$ which are sub-relations of $r(R_i)$ by selecting the tuples that only contain every keyword in K' and no others (Eq. 1). The non-leaf nodes are joins based on primary/foreign key references.

Example 3.3 Given a 2-keyword query, {Dress,Texas} over the E-Commerce database (Fig. 2). Let C, O, P, and G denote Customer, Order, Product, and Group. There exists a *CN*: $P\{Dress\} \bowtie G\{\} \bowtie P\{\} \bowtie O\{\} \bowtie C\{Texas\}$. The two joins, $P\{\} \bowtie O\{\}$ and $O\{\} \bowtie C\{Texas\}$ are specified using the only foreign key reference between two relation schemas, respectively. There are two foreign key references defined on P. The join between $P\{Dress\} \bowtie G\{\}$ is specified using the foreign key SID defined on G, and the join $G\{\} \bowtie P\{\}$ is specified using the foreign key PID defined on G. Both reference to the primary key on P. The *CN* is shown in Fig. 3a, where an edge with a label (foreign key attribute) indicates the foreign key that is used to join between two relations if there are multiple foreign key references, and an edge without a label implies that there is only one foreign key reference used in the join. Its left-deep operator tree is shown in Fig. 3b.

In order to reduce the query processing cost, a mesh, which is a forest of left-deep operator trees for all *CN*s, is constructed to share the processing cost of common join, selection, and projection operators. The entire mesh has $|V(G_S)| \cdot 2^m$ leaf nodes. Here, $|V(G_S)|$ is the number of nodes (relation schemas) in G_S, and 2^m is the number of sub-relations for each relation schema R_i, for an m-keyword query. Let the leaf nodes be at level 0, the level 1 is non-leaf nodes representing joins between two leaf nodes. The max number of level is Tmax $-$ 1. The mesh can be constructed in parallel with candidate network generation. The number of leaf nodes, $|V(G_S)| \cdot 2^m$, in the entire mesh **cannot** be reduced, simply because a relation that does not contain it may contain it in the future.

The authors in [26] observed that in a data stream environment, some joins need to be processed when there are incoming new tuples from its inputs but not all joins need to be processed all the time, and therefore proposed a demand-driven operator execution. A join operator has two inputs and is associated with an output buffer. The output buffer of a join operator becomes input to many other join operators that share the join operator. A tuple that is newly output by a join operator in its output buffer will be a new arrival input to those joins that share the join operator. A join operator will be in a running state, if it has new arrival tuples from both inputs. A join operator will be in a sleeping state if either it has no new arrival tuples from the right input or all the join operators that share it are currently sleeping. The demand-driven operator execution noticeably reduces the query processing cost.

3.2 Discussions: the scalability issue

The approach presented in [26] shows its effectiveness when processing an m-keyword query with both small m and Tmax over a slow data stream. The main problem of the approach is the scalability. The efficiency becomes a problem when processing an m-keyword query over a large schema graph $|G_S|$ for a given large Tmax. The reason is that the number

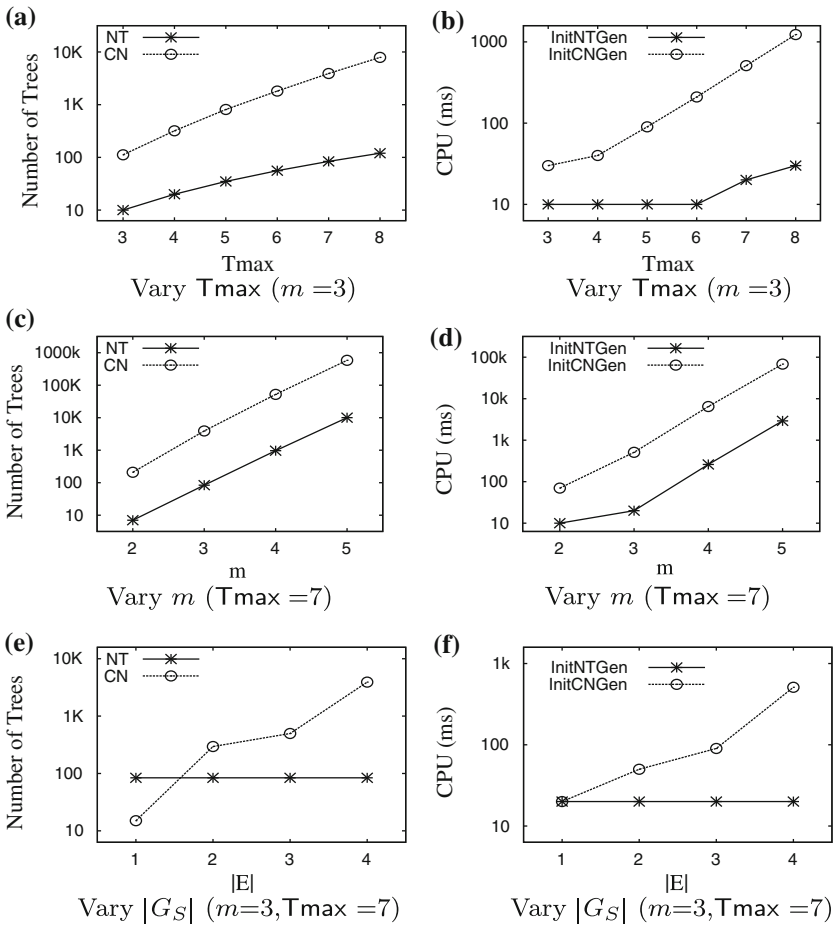


Fig. 4 *CN*/*NT* numbers on the E-Commerce database

of *CN*s becomes large, and the size of mesh becomes huge, when $|G_S|$, Tmax, or m becomes large. We discuss two main problems below.

The first is the scalability in generating all *CN*s. As also indicated in [26], it may take hours to generate all *CN*s when $|G_S|$, Tmax, or m becomes large. Note: in a real application, a schema graph can be large in size with a large number of relation schemas and complex foreign key references. There is also a need to be able to handle larger Tmax values. Consider a case where three customers in the same district ordered the same product in Example 2.1. The smallest number of tuples needed to include a *MTJNT* for such a case is Tmax = 7 (3 `Customer` tuples, 3 `Order` tuples, and 1 `Product` tuple).

Figure 4 shows the number of *CN*s, denoted `CN`, for the E-Commerce database schema (Fig. 1). Given the entire database schema, Fig. 4a shows the number of *CN*s by varying Tmax when the number of keywords is $m = 3$, and Fig. 4c shows the number of *CN*s by varying the number of keywords, m when Tmax $= 7$. Figure 4e shows the number of *CN*s by varying the complexity of the schema graph (Fig. 1). Here, the 4 points on x-axis represent four cases: Case-1 (`Customer` and `Order` with foreign key reference between the two relation schemas), Case-2 (Case-1 plus `Product` with foreign key reference between `Order` and `Product`), Case-3 (Case-2 plus `Group` with one of the two foreign key references between `Product` and `Group`), and Case-4 (Case-2 with both foreign key references between `Product` and `Group`). For the simple database schema with four relation schemas and four foreign key references, the number of *CN*s increases exponentially. For example, when $m = 5$ and Tmax $= 7$, the number of *CN*s is about 500,000. The number of joins in the mesh becomes the same order even after sharing common joins. The CPU time (ms) for computing all *CN*s by InitCNGen becomes high as shown in Fig. 4b, d, and f, respectively. When $m = 5$ and Tmax $= 7$, for computing 500,000 *CN*s, InitCNGen takes 70 s. CPU time increases exponentially when either $|G_S|$, Tmax, or m increases.

The second is the scalability in evaluating all *CN*s. The query processing cost becomes high when the speed of incoming data stream is high, for example, in an RFID stream context. Consider the 2-keyword query {`Dress`, `Texas`} over the E-Commerce database in Fig. 2. The join operator

(j_3) in Fig. 3b for the *CN* in Fig. 3a will output 14 tuples, but only 4 will be output as *MTJNT*s. This fact implies that a large number of intermediate tuples, which are computed by many join operators in the mesh with high processing cost, will not be eventually output in the end. The existing approach cannot avoid computing such a large number of unnecessary intermediate tuples, because it is unknown whether an intermediate tuple will appear in a *MTJNT* beforehand. The probability of generating a large number of unnecessary intermediate results becomes higher when either the size of sliding window, W, is large, or new data arrive in high speed. It becomes challenging how to reduce the processing cost by reducing the number of intermediate results.

4 A new approach

In this paper, we propose a novel scalable approach to process m-keyword queries. We also take the same two steps in our RDBMS framework: candidate network generation and candidate network evaluation.

First, for generating all *CN*s, we propose a new fast template-based approach. In brief, we first generate all *CN* templates (candidate network templates or simply network templates), denoted *NT*, and then generate all *CN*s based on all *NT*s generated. In other words, we do not generate all *CN*s directly like InitCNGen in [26]. The cost saving is high. We discuss our cost saving in comparison with InitCNGen in [26]. Recall: given an m-keyword query against a database schema G_S, there are $2^m \cdot |V(G_S)|$ nodes (relations), and accordingly there are $2^{2m} \cdot |E(G_S)|$ edges in total in the extended graph G_X. There are two main high overheads in InitCNGen.

–(**Cost-1**) The number of nodes in G_X that contain a certain selected keyword k_l is $|V(G_S)| \cdot 2^{m-1}$ (line 1). InitCNGen treats each of the $|V(G_S)| \cdot 2^{m-1}$ nodes, n_i, as the root of a *CN* cluster, and calls CNGen to find all valid *CN*s starting from the root n_i.

–(**Cost-2**) The CNGen algorithm, being called, expands a partial *CN* edge-by-edge, based on G_X, in every iteration, and searches all *CN*s whose size is $\leq$ Tmax. Note: in the expanded graph G_X, a node is connected to/from a large number of nodes. CNGen needs to expand **all** possible edges that are connected to/from every node (refer to line 8 in CNGen (Algorithm 1)).

In order to significantly reduce the two costs, we propose a new template-based approach. Here, a template, *NT*, is a special *CN* where every node, $R\{K'\}$, in *NT* is a variable that represents a sub-relation, $R_i\{K'\}$. Note: a variable represents $|V(G_S)|$ sub-relations. A *NT* represents a set of *CN*s. An example is shown in Fig. 5. The leftmost is a *NT*, $R\{k_1\} \bowtie R\{\} \bowtie R\{k_2\}$, shown as a tree rooted at $R\{\}$. There are many *CN*s that match the *NT* as shown in Fig. 5. For example, $C\{Texas\} \bowtie O\{\} \bowtie P\{Dress\}$ and $P\{Texas\} \bowtie G\{\} \bowtie P\{Dress\}$ match the *NT*. The number of *NT*s is much smaller than the number of *CN*s, as indicated by NT in Fig. 4a, c and e. When $m = 5$ and Tmax $= 7$, there are 500,000 *CN*s, but only less than 10,000 *NT*s.

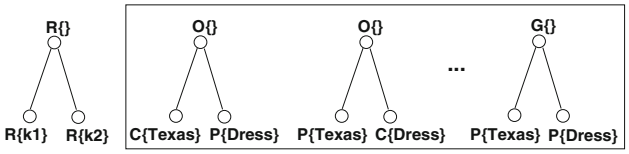


Fig. 5 A *NT* that represents many *CN*s

4.1 NT generation

We generate all *NT*s using a slightly modified InitCNGen algorithm against a special schema graph $G_1(V, E)$, where V only consists of a single node, R, and there is only one edge between R and itself. The relation schema R is with an attribute which is both primary key and foreign key referencing to its primary key.[3] With the special graph schema G_1, both costs (Cost-1 and Cost-2) can be significantly reduced in generating all *NT*s. In brief, InitCNGen [26] needs to call CNGen $|V(G_S)| \cdot 2^{m-1}$ times. Our approach only calls 2^{m-1} times. For Cost-2, since there is only one edge in the special schema graph G_1, we only need to check one possible edge when expanding,

4.2 CN generation

For every generated *NT*, we further generate all *CN*s that match it using a dynamic programming algorithm, which can be done fast. We will discuss it below in detail. For computing the same 500,000 *CN*s when $m = 5$ and Tmax $= 7$, our approach only takes 3 s. It is important to note that the complexity of schema graph, $|G_S|$, does not have significant impacts on the computational cost to generate *CN*s using our approach, as shown in Fig. 4f.

Second, for evaluating all *CN*s generated, we propose a novel demand-driven evaluation approach that fully reduces the intermediate join results. Our evaluation is a two-phase approach. In the first phase, we use low-cost selection and semijoin [5] to filter the tuples that cannot be joined. In the second phase, we only join tuples that can be possibly joined. We explain it using the same *CN*: $P\{Dress\} \bowtie G\{\} \bowtie$

[3] Note: in [26] InitCNGen can handle a database schema with self-loops. By labeling an edge with a foreign key, InitCNGen can handle parallel edges (multiple foreign key references) between two nodes in G_S.

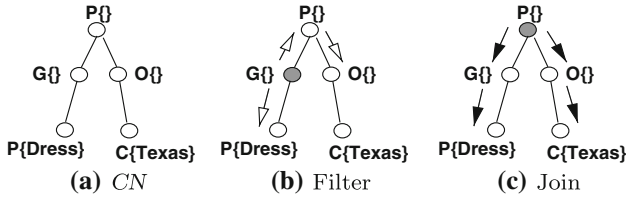


Fig. 6 A two-Step *CN* evaluation

Algorithm 2 FastKWS(G_S, q)

Input: a schema graph G_S, and a m-keyword query, q, with a given Tmax and sliding window W

1: $\mathcal{C} \leftarrow$ InitNTGen(G_S);
2: construct a lattice $\mathcal{L}$ based on $\mathcal{C}$;
3: index all m-keywords, $\{k_1, k_2, \ldots, k_m\}$;
4: $\Sigma \leftarrow \emptyset$;
5: **if** there exists a non-empty database on G_S to start **then**
6: $\quad \Sigma \leftarrow$ CNEvalStatic($\mathcal{L}$);
7: CNEvalDynamic($\mathcal{L}, q, \Sigma$);

$P\{\} \bowtie O\{\} \bowtie C\{Texas\}$ given in Example 3.3 (Fig. 3). The preliminary work was reported in [28]. We will discuss the details later in this paper.

- We construct a rooted tree for a given *CN* (Fig. 6a). Here, a node represents a sub-relation and an edge represents a join operator. Suppose there are already some tuples in the sub-relations, and consider query processing when a new tuple arrives.
- In the filter phase, when a new tuple arrives, for example g_i, in $G\{\}$, as indicated by the solid node in Fig. 6b, we first check if its child node, $P\{Dress\}$, has a tuple that can join g_i using a selection against $P\{Dress\}$. If there is no tuple in $P\{Dress\}$ that can join g_i, then the processing of the newly arrived g_i will stop. If there is at least one tuple in $P\{Dress\}$ that can join, we then use a semijoin to inform its parent node, $P\{\}$, of the new arrival tuple g_i. Assume we find that there is a tuple, p_j in $P\{\}$ that can join g_i, then we further check if p_j can join a tuple in the other child node of $P\{\}$, $O\{\}$, using a selection. If there is no tuple in $O\{\}$ that can join p_j, the processing will stop. Note: $O\{\}$ does not need to check further its child nodes.
- In the join phase, suppose we find that p_j in $P\{\}$ can be joined by some tuples in both its child nodes, namely, $G\{\}$ and $O\{\}$, it starts joining process in a top-down manner, as indicated in Fig. 6c. When we join, all the tuples must be able to join, and there is no unnecessary intermediate results. By this, we mean that we can achieve full reduction in terms of intermediate results. It is important to know that it is the main reason that we can achieve high efficiency when handling high-speed data streams.

Our main algorithm is outlined in Algorithm 2. In brief, it first generates all *CN*s, by calling InitNTGen at line 1. Then,

Algorithm 3 InitNTGen(G_S)

1: $\mathcal{C} \leftarrow \emptyset$;
2: $\mathcal{T} \leftarrow$ InitCNGen(G_1);
3: **for all** $T \in \mathcal{T}$ **do**
4: $\quad$ **for all** nodes $R_i \in V(G_S)$ **do**
5: $\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cup$ NT2CN(T, R_i, G_S);
6: **return** $\mathcal{C}$;

Procedure NT2CN(T, R_i, G_S)

1: **Procedure** NT2CN(T, R_i, G_S)
2: $r \leftarrow root(T)$;
3: $\mathcal{C}' \leftarrow \{R_i\{K'\}\}$ if r is with a subset of keywords K';
4: **for all** immediate child of r, s, in T **do**
5: $\quad \mathcal{C}'' \leftarrow \emptyset$;
6: $\quad$ **for all** $\{R_j | (R_i, R_j) \in E(G_S) \vee (R_j, R_i) \in E(G_S)\}$ **do**
7: $\quad\quad \mathcal{C}'' \leftarrow \mathcal{C}'' \bigcup$ NT2CN($subtree(s), R_j, G_S$);
8: $\quad \mathcal{C}' \leftarrow \mathcal{C}' \times \mathcal{C}''$;
9: $\quad$ prune $\mathcal{C}'$ using (Rule-3);
10: **return** $\mathcal{C}'$;

it compacts all *CN*s by constructing a *CN* lattice, denoted $\mathcal{L}$ (line 2).We will discuss them in Sect. 5. Let the set of initial *MTJNT*s be Σ. If there already exists a database when the service of m-keyword query processing is to start or to resume after it has suspended for a while, it can quickly compute all *MTJNT*s from the database, if needed (line 5–6). Finally, it calls CNEvalDynamic to start processing m-keyword queries over a data stream. We will discuss them in Sect. 6.

5 Candidate network generation

5.1 The algorithm

Our new template-based algorithm to generate *CN*s is illustrated in Algorithm 3. It first initializes the set of *CN*s, $\mathcal{C}$, to be empty (line 1). Second, it generates all *NT*s, $\mathcal{T}$, by calling the InitCNGen algorithm given in [26] with a special one node and one edge graph, G_1. An example is given in Fig. 7 to show how to generate one *NT* with 3 keywords $K1$, $K2$, and $K3$ (right on the top row) from the special schema graph G_1 (left on the top row). As shown in the bottom row in Fig. 7. Initially, start from the partial *NT* consists of only one node $R\{K1\}$ that contains the first keyword $K1$; in every step, it adds a node into the partial *NT* on its rightmost root-to-leaf path until it contains all the keywords $K1$, $K2$ and $K3$. Then, it outputs this as a valid *NT*. Note that in each step after adding a new node the 3 legality checking conditions introduced in Sect. 3.1 must hold. Then, for every network template $T \in \mathcal{T}$, it attempts to generate all *CN*s that match T by enumerating all the relations, R_i, in the schema graph, G_S. It calls NT2CN for every template T and every relation (line 3–5). The set of *CN*s, $\mathcal{C}$, will be returned in the end. We will prove that $\mathcal{C}$ is sound/complete and duplicate free.

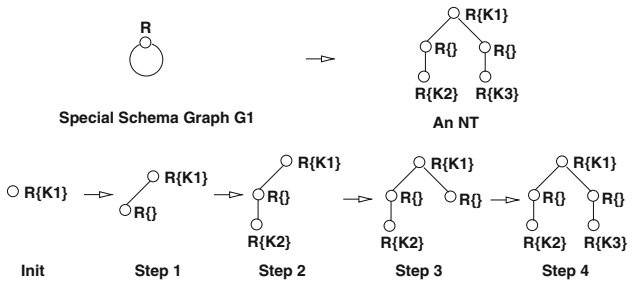


Fig. 7 An example to generate *NT*s

We formalize every template T as a rooted tree and therefore obtain a set of *CN*s as a set of rooted trees that match some template. Here, the root of a *NT*/*CN* tree is the node, r, in the tree such that the maximum path from r to its leaf nodes is minimized. Such a root can be quickly identified as the center of the tree by removing all the leaf nodes in every iteration repeatedly until there is either a single node (one-center) or two nodes with a single edge left (two-centers). If a tree has one-center, the center is the root which is unique. If a tree has two centers, we select the node that has a smaller node identifier as the root. The height of any *NT* is at most $\mathsf{Tmax}/2 + 1$.

The procedure NT2CN (line 7–16, Algorithm 3) assigns real labels such as relation names to a *NT* and therefore generates all *CN*s that match a given *NT*. In NT2CN, for a given *NT*, T, and a real relation R_i, it first identifies the root of T, as r (line 8). Because a node *NT* is associated with a set of keywords K', $R_i\{K'\}$ is a possible match to match the root note r, and $R_i\{K'\}$ is assigned to $\mathcal{C}'$ (line 9). Next, it considers the child nodes of r, indicated as s, in T, and repeats the same attempts to label s with a relation name by recursively calling NT2CN (line 10–13). For any return results, $\mathcal{C}''$, of the union for recursive calls (line 13), it conducts $\mathcal{C}' \times \mathcal{C}''$ (line 14) followed by pruning using the same (Rule-3) to prune any partial invalid *CN*s.

The whole process of InitNTGen is illustrated in Fig. 8. It starts from a special graph G_1 and then generates all *NT*s. For each *NT*, it generates a set of *CN*s that match the *NT*. We prove the correctness of InitNTGen in Theorem 5.1.

Theorem 5.1 *The new* InitNTGen *algorithm generates a sound/complete and duplicate-free set of CNs.*

Proof Sketch: We prove Theorem 5.1 by showing that our InitNTGen generates the same set of *CN*s that is generated by InitCNGen. The algorithm to generate all *NT*s is exactly the same as InitCNGen by applying on the schema graph G_1. Its correctness is proved in [26]. We show the main ideas here in brief. First, every result tree will be generated. It is because every tree can be constructed by adding a node on the rightmost root-to-leaf path iteratively (see legality checking condition (1) of Algorithm 1). Second, the result trees

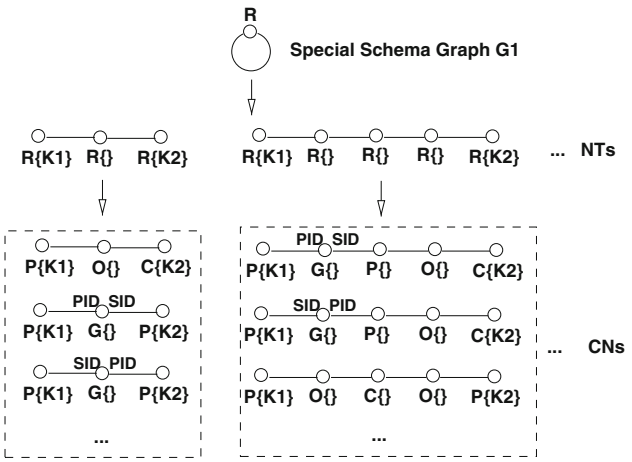


Fig. 8 Generate all *CN*s using *NT*s

generated are not redundant. It is because for each result tree we can always find a unique representation such that all sibling nodes are lexicographically ordered (see legality checking condition (2) and (3) of Algorithm 1).

Let the set of *CN*s generated by InitCNGen be $\mathcal{C}$, and let the set of *CN*s generated by our new InitNTGen be $\mathcal{C}'$. We show that $\mathcal{C} \subseteq \mathcal{C}'$ and $\mathcal{C} \supseteq \mathcal{C}'$. Note: the set of *CN*s, $\mathcal{C}'$, is generated from a set of *NT*s, $\mathcal{T}$, which is computed by the same InitCNGen in [26] (refer to line 2 in Algorithm 3).

First, we prove $\mathcal{C} \subseteq \mathcal{C}'$ in two steps. (1) For any $C_i \in \mathcal{C}$, it must match a *NT*, $T_j \in \mathcal{T}$. Such a T_j can be determined as follows: we replace all the relation names in C_i by R (the only relation name in G_1), and replace all the edges (joins) in C_i by the only edge in G_1; in addition, we keep the keywords associated with every node in C_i but replace every real user-given keyword by a keyword identifier. For example, `Dress` and `Texas` are replaced by k_1 and k_2, respectively. This *NT*, T_j, is minimal and total, because the corresponding *CN*, C_i, is minimal and total as it is computed by InitCNGen. This T_j must be computed by InitCNGen(G_1) as a minimal and total *NT* (a special *CN*) (refer to line 2 in Algorithm 3). Therefore, C_i matches T_j that is computed by our algorithm. (2) Based on the set of *NT*s, $\mathcal{T}$, we enumerate all *CN*s and obtain $\mathcal{C}'$, therefore $C_i \in \mathcal{C}'$, and we conclude that $\mathcal{C} \subseteq \mathcal{C}'$.

Second, we prove $\mathcal{C} \supseteq \mathcal{C}'$ by showing that $\mathcal{C}'$ is duplicate free. Because $\mathcal{C}$ is duplicate-free and complete, if $\mathcal{C}'$ is duplicate-free then $\mathcal{C} \supseteq \mathcal{C}'$. We do it by considering two cases. (a) Two different *NT*s, T_j and T_k, cannot generate the same *CN*, C_i, which is obvious. (b) All *CN*s generated from the same *NT*, T_j, using our approach are all different. We ensure that in NT2CN which generates all different *CN*s for a given *NT*. For each *CN* C, generated from T_j, there in a unique one-to-one mapping between nodes in C and nodes in T_j. Recall that each leaf node for a *NT* or a *CN* must contain a unique keyword which does not appear in other nodes. This is guaranteed by the minimal condition for a *CN* (a *NT* is also a special *CN*). Given the unique keyword for

every leaf node in C and T_j, the unique one-to-one mapping between all leaf nodes in C and all leaf nodes in T_j can be identified. The unique mapping between the remaining nodes in C and T_j are able to be identified because of the unique mapping between leaf nodes. Based on the unique one-to-one mapping, suppose C and C' are any two *CN*s generated from the same *NT*, T_j, by NT2CN. C and C' have the same structure because they are generated from the same *NT* T_j. We prove that C and C' must have different relation names assigned to some nodes, and they can not be the same. Given the unique one-to-one mapping between nodes in C and nodes in T_j and the unique one-to-one mapping between nodes in C' and nodes in T_j. It is obvious that there is unique one-to-one mapping between nodes in C and nodes in C'. With this condition, we only need to prove that there is at least a node in C, such that the corresponding node in C' has a different type (relation name). Let us assume that two *CN*s, C and C', are the same. As shown in InitNTGen, it is only possible when the two *CN*s, C and C', are generated following the same execution steps. In other words, C and C' must be generated from the same type of root (R_i in line 5 of InitNTGen), and their subtrees must be the same and are generated by the same subtrees recursively (refer to line 11 of InitNTGen). However, it is impossible, since following the exact same execution steps, only one *CN* can be generated. Consider two *CN*s, C and C', that are generated from different execution steps. Also following InitNTGen (line 5 and line 11), there must be one type (relation name) assigned to C and C' different.

Theorem 5.2 *The number of NTs generated is bounded by* $O((\mathsf{Tmax} \cdot 2^m)^{\mathsf{Tmax}})$ *which is independent of the complexity of the schema graph* G_S.

Proof Sketch: When generating each *NT*, before adding a node in each partial *NT*, we have to decide 1) the set of keywords that the node contains and 2) the node in the partial *NT* that the new node is connected to. There are $O(2^m)$ cases regarding 1), and for each of such cases, there are $O(\mathsf{Tmax})$ cases regarding 2). As a result, there are at most $O(2^m \cdot \mathsf{Tmax})$ cases for adding a new node in each partial *NT*. Because the size of an *NT* is at most Tmax, there are at most $O((\mathsf{Tmax} \cdot 2^m)^{\mathsf{Tmax}})$ different *NT*s generated.

Theorem 5.2 shows that the number of valid *NT*s generated can be bounded by a value which is independent of the complexity of the schema graph, even though the upper bound is very loose. Consider $m = 3$ and $\mathsf{Tmax} = 10$. In the worst case, the upper bound following Theorem 5.2 approximately equals 10^{19}. In practice, the bound is much small and can be enumerated in a similar way as discussed above. When generating *NT*s using G_1, which is independent of the schema graph of the database, we can enumerate all possible *NT*s including those that cannot be matched by a *CN*. The number of such all possible *NT*s is an upper bound of the total number of valid *NT*s. Table 1 shows the number of candidate *NT*s generated for different m and Tmax, for all $2 \leq m \leq 5$ and $2 \leq \mathsf{Tmax} \leq 10$. For $m = 3$ and $\mathsf{Tmax} = 10$, the upper bound becomes 220, which is much smaller than 10^{19}. The bounds shown in Table 1 are the results of the enumeration but cannot be easily represented using a formula.

Table 1 The upper bound of the number of *NT*s

Tmax	m			
	2	3	4	5
2	2	4	8	16
3	3	10	33	106
4	4	20	98	456
5	5	35	238	1,506
6	6	56	504	4,152
7	7	84	966	10,031
8	8	120	1,716	21,912
9	9	165	2,871	44,187
10	10	220	4,576	83,512

Remark 5.1 The complexity of schema graph, G_S, does not have significant impacts on the efficiency of our algorithm InitNTGen. The efficiency of InitNTGen is determined by two factors, Tmax and m.

Remark 5.1 is made by the fact that InitNTGen does not generate a large expanded graph G_X to work with. It instead starts from a smaller graph G_1. Remark 5.1 is confirmed by our experimental studies as also indicated in Fig. 4. We discuss the issues related to time complexity below.

The number of *CN*s is exponentially large with respect to three factors, namely, the number of keywords in the query m, the size control parameter Tmax, and the schema complexity $|G_S|$. The time/space complexity for both InitCNGen in [26] and our new InitNTGen are difficult to derive, because the number of *CN*s is not easy to know. Below, we explain why InitNTGen is much better than InitCNGen by showing that our new InitNTGen can avoid checking an exponential number of partial results, which is costly. First, we generate all *NT*s using InitCNGen upon a special schema graph G_1 with only 1 node. The time complexity to generate all *NT*s on G_1 is much smaller than the time complexity to generate all *CN*s on the original graph G_S using InitCNGen, because the former does not need to consider the exponential with respect to the schema complexity $|G_S|$. Our main cost saving is on evaluating all *NT*s using NT2CN. For NT2CN, although we cannot avoid generating all *CN*s that are in the final result, we avoid generating a large number of partial *CN*s that cannot potentially result in a valid *CN* or will result in redundant *CN*s. Recall that in CNGen (Algorithm 1)

proposed in [26], before expanding the partial *CN* by one node u, it needs to check whether u can be legally added to the partial *CN*. The "legally added" means that the node u can be inserted only to the rightmost root-to-leaf path, and it must be no smaller than any of its siblings. Furthermore, if after adding u, the partial tree contains all the keywords, all the immediate subtrees for each node must be lexicographically ordered. Therefore, it needs $O(\mathsf{Tmax}^2)$ time to check whether u can be legally added. There are $O(2^m \cdot |V(G_S)|)$ nodes in total to be checked, and it is highly possible that only few of nodes can be legally added. In the worst case, it needs $O(2^m \cdot |V(G_S)| \cdot \mathsf{Tmax}^2)$ time to check whether it can add one node to a partial *CN*. Reducing such cost can be crucial to the total time complexity to generate all *CN*s. In our algorithm, in generating all *NT*s, for each partial *NT*, there may be an exponential number of partial *CN*s that can match it. We use the same operations to check, but we only need to check it once, because in NT2CN, all *CN*s generated are complete and duplicate free, and no such checking operations are needed. As a result, we can avoid an exponential number of such costly checking operations with respect to m, $|G_S|$ and Tmax.

Theorem 5.3 *Comparing to the* InitCNGen *algorithm proposed in [26], the total time cost saving for our algorithm* InitNTGen *is* $O(M \cdot |V(G_S)|^{\mathsf{Tmax}} \cdot 2^m \cdot \mathsf{Tmax}^2)$*, where M is the total number of NTs generated.*

Proof Sketch: In the worst case, a partial *NT* has at most $\mathsf{Tmax}-1$ nodes, and for each node, there are at most $|V(G_S)|$ relations that can match it. Thus, for each *NT*, InitNTGen can avoid $|V(G_S)|^{\mathsf{Tmax}-1}$ checking operations. As analyzed above, the time cost for each checking operation is $O(2^m \cdot |V(G_S)| \cdot \mathsf{Tmax}^2)$. Consequently, the total time cost saving for the InitNTGen algorithm is $O(|V(G_S)|^{\mathsf{Tmax}-1} \cdot 2^m \cdot |V(G_S)| \cdot \mathsf{Tmax}^2 \cdot M) = O(M \cdot |V(G_S)|^{\mathsf{Tmax}} \cdot 2^m \cdot \mathsf{Tmax}^2)$. □

The time complexities of InitCNGen and InitNTGen algorithms are exponential with respect to m, $|G_S|$ and Tmax. Theorem 5.4 shows that although the InitNTGen algorithm has the same asymptotic cost as the InitCNGen algorithm, the cost saving of InitNTGen is still remarkable. Our experimental results also show such large cost savings.

Below, we drive a lower bound for the cost saving of our InitNTGen algorithm. When expanding a partial *CN* by one node using the InitCNGen algorithm, in each checking operation, we need to check $2^m \cdot |V(G_S)|$ nodes. In the best case, when there is only one node in the partial *CN*, we can use constant time to check whether a node can be legally added to the partial *CN*. In our InitNTGen algorithm, in the best case, when a partial *NT* has only one node, there are totally $|V(G_S)|$ relations that can match it. As a result, InitNTGen can avoid at least $|V(G_S)|$ checking operations. From the above analysis, the lower bound cost saving for our InitNTGen algorithm is $o(2^m \cdot |V(G_S)| \cdot |V(G_S)| \cdot M) = o(M \cdot |V(G_S)|^2 \cdot 2^m)$.

Theorem 5.4 *Compared to the* InitCNGen *algorithm proposed in [26], the lower bound of total time cost saving for our algorithm* InitNTGen *is* $o(M \cdot |V(G_S)|^2 \cdot 2^m)$*, where M is the total number of NTs generated.*

Proof Sketch: See the above discussions.

5.2 $\mathcal{L}$-Lattice

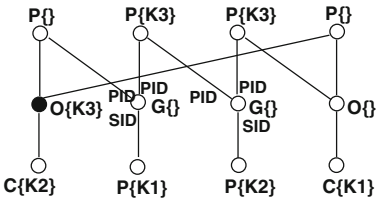
Given a set of *CN*s, $\mathcal{C}$. We construct a lattice, $\mathcal{L}$, in order to share its query processing cost among all *CN*s. The procedure is given below. When a new rooted *CN*, C_i, is inserted to $\mathcal{L}$, we generate the canonical codes for all its rooted subtrees of the rooted *CN* tree, C_i, including C_i itself. A canonical code is a string. Two trees, C_i and C_j, are identical iff their canonical codes are identical. We index all subtrees in $\mathcal{L}$ using their canonical codes over $\mathcal{L}$, while constructing $\mathcal{L}$. For a given rooted *CN* C_i, we attempt to find largest subtrees in $\mathcal{L}$ that C_i can share with using the index and link to the roots of such subtrees. Figure 9a illustrates a partial lattice. The entire lattice, $\mathcal{L}$, is maintained in two relations: `L-Node` relation and `L-Edge` relation (Fig. 9b). Let a bit-string represents a set of keywords, $\{k_1, k_2, \ldots, k_m\}$. In the `L-Node` relation, for any node in $\mathcal{L}$, it maintains a unique `Vid` in $\mathcal{L}$, the corresponding relation name (`Rname`) that appears in the given database schema, G_S, a bit-string (`KSet`) that indicates the keywords the node in $\mathcal{L}$ associated with, and the size of the bit-string (`m`). The `L-Edge` relation maintains the parent/child relations among all the nodes in $\mathcal{L}$ with its parent `Vid` and child `Vid` (`Fid`/`Cid`) plus its join attribute, `Attr`, (either primary key or foreign key). The two relations can be maintained in memory or on disk. Several indexes are build on the relations to fast search given nodes in $\mathcal{L}$.

When processing queries, the lattice $\mathcal{L}$ saves computational cost by sharing subexpressions among different *CN*s. Each node v of the lattice $\mathcal{L}$ represents the subtree rooted at v and can be shared by different nodes, for example u and u', in $\mathcal{L}$ if and only if v is a subtree of the trees rooted at u and u'. When evaluating u and u', the node v only needs to be evaluated once. The uppermost nodes in $\mathcal{L}$ are not shared by any other nodes and will be used to generate the final results. As an example, in the lattice $\mathcal{L}$ shown in Fig. 9a, the node denoted $O\{\}$ represents the subexpression $O\{\} \bowtie C\{K1\}$ and is shared by two nodes denoted $P\{K3\}$ and $P\{\}$. When evaluating the subexpressions rooted at $P\{K3\}$ and $P\{\}$, the subexpression $O\{\} \bowtie C\{K1\}$ only needs to be evaluated once.

6 Candidate network evaluation

In this section, we discuss *CN* evaluation. We discuss two cases. First, given a database schema G_S, in many real

Fig. 9 $\mathcal{L}$-Lattice



(a) Structure

L–Node Table

Vid	Rname	KSet	m
1	O	001	1
...	...	...	...

L–Edge Table

Fid	Cid	Attr
1	3	SID
...	...	...

(b) Storage

applications, it starts the services of supporting m-keyword queries when the database on G_S is non-empty. It may request for a large sliding window $W = \infty$, in order to find *MTJNT*s that have some tuples in an incoming data stream and some tuples in the underneath database. In such a case, as discussed before, we first use CNEvalStatic to process the data that are already in the database, and then use CNEvalDynamic to process the data stream. Second, if there is no such a database, we use CNEvalDynamic to process a data stream. We will discuss CNEvalStatic and CNEvalDynamic in Sects. 6.1 and 6.2, respectively.

We maintain $|V(G_S)|$ relations in total to process an m-keyword query $K = \{k_1, k_2, \ldots, k_m\}$, due to the lattice structure we used. In our approach, a node, v, in the lattice $\mathcal{L}$ is uniquely identified with a node id. The node v represents a sub-relation $R_i\{K'\}$. By utilizing the unique node id, we can easily maintain all the 2^m sub-relations for a relation R_i together. We denote such a relation as $\mathbf{R}_i$. The schema of $\mathbf{R}_i$ is the same as R_i plus an additional attribute (`Vid`) to keep the node id in $\mathcal{L}$. When we need to obtain a sub-relation $R_i\{K'\}$ for $K' \subseteq K$ associated with a node, v, in the lattice, we use the node id to select and project $R_i\{K'\}$ from $\mathbf{R}_i$. Therefore, a relation $R_i\{K'\}$ can be possibly virtually maintained. Below we use $\mathbf{R}_i\{K'\}$ to denote such a sub-relation. It is fast to obtain $\mathbf{R}_i\{K'\}$ if an index is build on the additional attribute `Vid` on relation $\mathbf{R}_i$. Such a sub-relation for `Order` relation is shown in Fig. 10b.

6.1 Static evaluation

CNEvalStatic is outlined in Algorithm 4. There are two main phases to compute m-keyword queries in a database on G_S. The first phase is a filtering phase that identifies the set of tuples that can form *MTJNT*s (line 1–3). The second phase is to join such *MTJNT*s (line 5–7). In the first phase, we mainly use selections and projections. In the second phase, we use join operators to join tuples. We achieve full reduction in the second phase because all the tuples in root nodes must be able to join some tuples to form *MTJNT*s. We explain the two phases below. The filtering phase is done in a bottom-up fashion for all nodes, v, in the lattice $\mathcal{L}$. Let v represents $R_i\{K'\}$. It projects the tuples that contain all the keywords in K' and no others from relation $r(R_i)$ and maintain them in $\mathbf{R}_i\{K'\}$ (line 2). The node id of v will be inserted into the attribute `Vid` in $\mathbf{R}_i\{K'\}$ together. Note: when processing a node, v, in $\mathcal{L}$, all its descendants have been processed already in the same manner. In line 3, it uses a selection based on either primary or foreign key to check whether there is a joined tuple in every relation represented by its child node in $\mathcal{L}$. The `Vid` will be used in such selections. The tuples that cannot find a join match in any its child relation will be pruned. It repeats until it reaches the root nodes in $\mathcal{L}$. All the tuples left in the root nodes in $\mathcal{L}$ can form *MTJNT*s. In the second join phase, it starts from the root nodes in $\mathcal{L}$ and process joins in a top-down manner. It calls Eval which is a recursive procedure to join all the tuples to form *MTJNT*s.

Algorithm 4 CNEvalStatic($\mathcal{L}$)

1: **for all** nodes, v, labeled $R_i\{K'\}$ in lattice $\mathcal{L}$ in a bottom-up fashion **do**
2: $\quad\mathbf{R}_i\{K'\} \leftarrow \pi_{K'}(r(R_i))$ with v's id;
3: $\quad$let $\mathbf{R}_i\{K'\}$ be those tuples in $\mathbf{R}_i$ that join at least a tuple in every its child relation in $\mathcal{L}$;
4: $\mathsf{T} \leftarrow \emptyset$;
5: **for all** root node, v, labeled $R_i\{K'\}$ in $\mathcal{L}$ **do**
6: $\quad$**for all** tuple $t \in \mathbf{R}_i\{K'\}$ **do**
7: $\quad\quad\mathsf{T} \leftarrow \mathsf{T} \cup \mathsf{Eval}(v, t)$;
8: **return** T;

9: **Procedure** Eval(v, t)
10: let the label of node v be $R_i\{K'\}$;
11: $\mathsf{T}' \leftarrow \{t\}$;
12: **for all** child node u of v in lattice $\mathcal{L}$ **do**
13: $\quad\mathsf{T}'' \leftarrow \emptyset$;
14: $\quad$let the label of node u be $R_j\{K''\}$;
15: $\quad$**for all** tuples $t' \in \mathbf{R}_j\{K''\}$ that join t **do**
16: $\quad\quad\mathsf{T}'' \leftarrow \mathsf{T}'' \cup \mathsf{Eval}(u, t')$;
17: $\quad\mathsf{T}' \leftarrow \mathsf{T}' \times \mathsf{T}''$;
18: **return** T′;

6.2 Dynamic evaluation

CNEvalDynamic is outlined in Algorithm 5. For a new arrival update operator, $op(t, R_i)$, it processes it in line 3–9 if the operation is an insertion and processes it in line 11–14 if it is a deletion. We explain Algorithm 5 using examples. Here, we suppose the *CN* to be processed is shown in Fig. 10a as a part of $\mathcal{L}$, where every node in $\mathcal{L}$ is uniquely identified with a node id, v_i, for $1 \leq i \leq 5$. Reconsider E-Commerce database,and let the data currently maintained

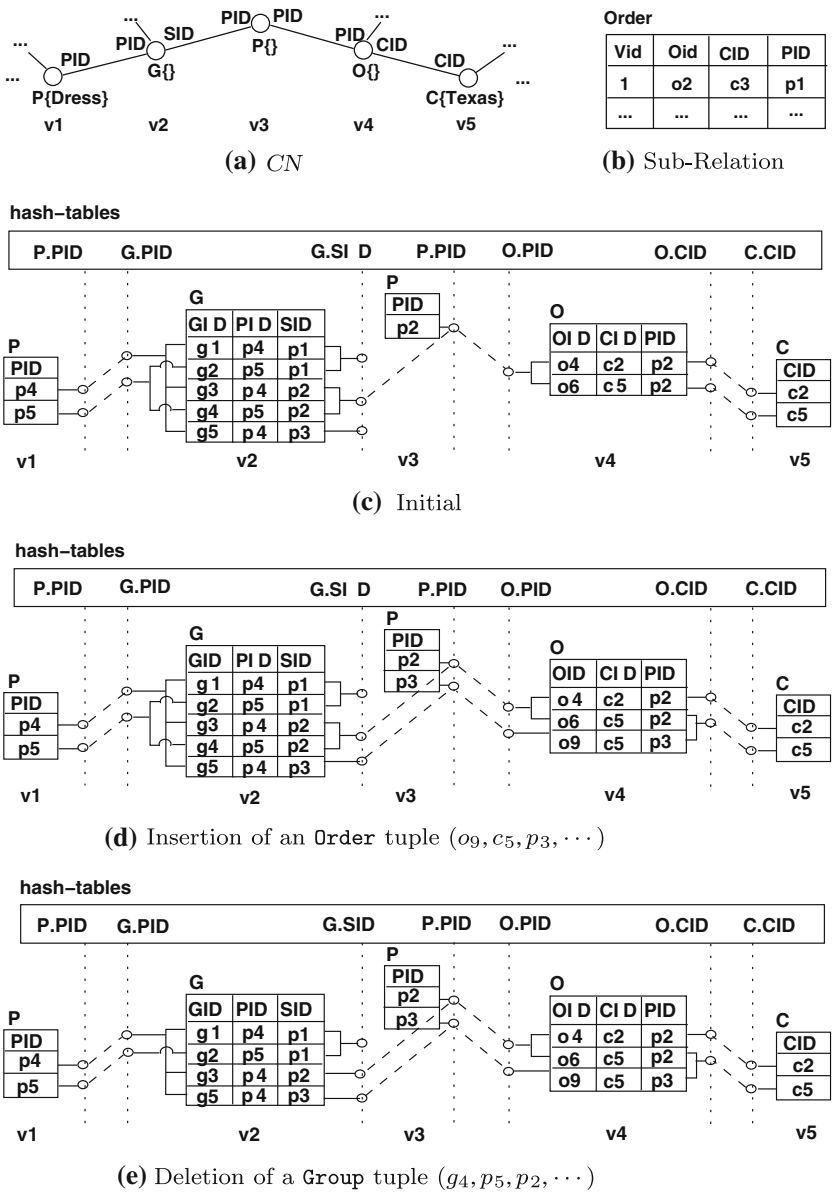


Fig. 10 CNEval examples (insertion and deletion)

(a) *CN* **(b)** Sub-Relation

(c) Initial

(d) Insertion of an `Order` tuple $(o_9, c_5, p_3, \cdots)$

(e) Deletion of a `Group` tuple $(g_4, p_5, p_2, \cdots)$

be Fig. 10c. In Fig. 10c, there is a sub-relation associated with every node v_i, $1 \leq i \leq 5$. For example, v_1 represents $P\{Dress\}$ and maintains all P-relation tuples that contain $Dress$ and no other keywords, $\{p_4, p_5\}$. In a similar fashion, v_2 represents $G\{\}$ and maintains a set of G-relation tuples $\{g_1, g_2, g_3, g_4, g_5\}$. Those G-relation tuples can join at least a tuple maintained in its child relation (v_1). In Fig. 10c, only those tuples are shown if they contain the requested set of keywords and at the same time can join at least a tuple in every its child relation. There are hash tables build to facilitate accessing the required tuples as indicated in Fig. 10.

Suppose a new O-relation tuple (o_9, c_5, p_3) with no keywords is newly inserted. First, we try to insert this tuple into the sub-relation represented by v_4 which is labeled $O\{\}$. As shown in Fig. 10d, we find that (o_9, c_5, p_3) can be inserted into v_4's sub-relation because it is not in the sub-relation yet, and it can join a tuple in the $C\{Texas\}$ relation represented by v_5. It is done using a selection to check whether a tuple in $C\{Texas\}$ contains c_5, which can be done fast. Then, we check whether new tuples can be added into v_4's father node v_3. Suppose we find that the P-relation tuple, p_3, is in the sliding window, and can now, at the first time, join a tuple in v_4, (o_9, c_5, p_3). It can be done using a semijoin. Before we insert p_3 tuple into v_3, we further check whether the p_3 tuple can join at least a tuple in the relation represented by its other child v_2, using a selection. Because it is true, we add p_3 tuple into the sub-relation indicated by v_3. Note: we reach a root node in $\mathcal{L}$. Because p_3 is newly inserted into the root node, there must be new *MTJNT*s to be reported. We then call EvalPath to join all the needed tuples in a top-down fashion.

Algorithm 5 CNEvalDynamic($\mathcal{L}$, q, Σ)

Input: An m-keyword query q, a lattice $\mathcal{L}$, and a set of *MTJNT*s denoted Σ

```
1: while a new update op(t, R_i) arrives do
2:   let K' be the set of all keywords appearing in tuple t;
3:   if op is to insert t into relation R_i then
4:     Δ ← ∅;
5:     for each v in L labeled R_i{K'} do
6:       path ← ∅;
7:       insert(v, t, Δ);
8:       report new MTJNTs in Δ;
9:       Σ ← Σ ∪ Δ;
10:  else if op is to delete t from R_i then
11:    for each v in L labeled R_i{K'} do
12:      if t ∈ R_i{K'} then
13:        delete(v, t);
14:        delete MTJNTs in Σ that contain t, and report such deletions;

15: Procedure insert(v, t, Δ)
16: let the label of node v be R_i{K'};
17: if t ∉ R_i{K'} and t can join at least one tuple in every relation represented by all v's children in L then
18:   insert tuple t into the sub-relation R_i{K'};
19: if t ∈ R_i{K'} then
20:   push (v, t) to path;
21:   if v is a root node in L then
22:     Δ ← Δ ∪ EvalPath(v, t, path);
23:   else
24:     for each father node of v, u, in L do
25:       let the label of node u be R_j{K''};
26:       for each tuple t' in π_{K''}(r(R_j)) that can join t do
27:         insert(u, t', Δ);
28:   pop (v, t) from path;

29: Procedure delete(v, t)
30: let the label of node v be R_i{K'};
31: delete tuple t from the sub-relation R_i{K'};
32: for each father node of v, u in L do
33:   let the label of node u be R_j{K''};
34:   for each tuple t' in R_j{K''} that can join t only do
35:     delete(u, t');
```

The EvalPath is implemented in a similar fashion as Eval in Algorithm 4 using an additional $path$, which records where the join sequence comes from to reduce join cost. Currently, $path$ is $o_9 \rightarrow p_3$. A new *MTJNT* is $(p_4, g_5, p_3, o_9, c_5)$.

Next, suppose we delete a G-relation tuple (g_4, p_5, p_2). As shown in Fig. 10e, we find that tuple (g_4, p_5, p_2) is contained in v_2 which is labeled $G\{\}$. We remove (g_4, p_5, p_2) from the sub-relation and check whether some tuples can be removed from the relation indicated by its father node v_3. We find that only p_2 tuple in v_3 joins (g_4, p_5, p_2). But the p_2 tuple in v_3 cannot be removed because it also joins some other tuples in the relation represented by v_2. We remove the two existing *MTJNT*s, $(p_5, g_4, p_2, o_4, c_2)$ and $(p_5, g_4, p_2, o_6, c_5)$ that contain (g_4, p_5, p_2).

As the number of results itself can be exponential large to the size of the input, we analyze the extra cost for the algorithms to evaluate all *CN*s. The extra cost is defined to be the number of tuples generated by the algorithm minus the number of tuples in the result. We show the efficiency of our algorithm in the following Theorem.

Lemma 6.1 *Suppose the number of tuples in every relation is n, given a CN with size t. The extra cost for the algorithm using the left deep tree proposed in [26] to evaluate the CN is $O(n^{t-1})$, and the extra cost for the algorithm using the semijoin-join approach proposed in this paper to evaluate the CN is $O(n \cdot t)$.*

Proof Sketch: Suppose we evaluate a *CN*, $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_t$, using a left deep tree. The ith output of the left deep tree is the result of $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_i$. Such result can contain as large as $O(n^i)$ results, and it is possible that only few of them may participate in the final result. The overall extra cost can be up to $O(n^{t-1})$. If we evaluate the *CN* $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_t$ using our semijoin-join approach. In the first semijoin phase, the ith output is of form $R_1 \rtimes R_2 \rtimes \ldots \rtimes R_i$, and the extra cost for the i-th output is controlled by $O(n)$. So the over extra cost for the first semijoin phase is $O(n \cdot t)$. In the second join phase, all the i-th output can participate in the final result, and no extra cost is needed. □

Theorem 6.5 *The total time cost saving of our algorithm to evaluate all CNs is $O(\frac{N \cdot n^{\mathsf{Tmax}-1}}{|V(G_S)| \cdot 2^{m-1}})$, compared to [26], where N is the total number of CNs and n is the average number of tuples in each relation.*

Proof Sketch: Using the mesh proposed in [26], there are totally N root nodes in each mesh because each root node in the mesh represents a unique *CN*. We denote the root nodes in the mesh the first level and their direct child nodes in the mesh the second level. It is obvious that each father node of any node in the second level contains at most $m-1$ keywords. Thus, each partial *CN* in the second level can generate at most $|V(G_S) \cdot 2^{m-1}|$ *CN*s in the first level, or in other words, there are at most $O(\frac{N}{|V(G_S)| \cdot 2^{m-1}})$ nodes in the second level. As analyzed in Lemma 6.1, the extra cost to evaluate each node in the second level is $O(n^{\mathsf{Tmax}-1})$. Consequently, the total extra cost to evaluate the mesh in [26] is $O(\frac{N}{|V(G_S)| \cdot 2^{m-1}} \cdot n^{\mathsf{Tmax}-1})$. Similarly, the total extra cost for our semijoin-join based approach is $O(\frac{N}{|V(G_S)| \cdot 2^{m-1}} \cdot (\mathsf{Tmax} \cdot n))$. As a result, the extra cost for [26] is $O(\frac{N}{|V(G_S)| \cdot 2^{m-1}} \cdot (n^{\mathsf{Tmax}-1} - (\mathsf{Tmax} \cdot n))) = O(\frac{N \cdot n^{\mathsf{Tmax}-1}}{|V(G_S)| \cdot 2^{m-1}})$. □

Theorem 6.6 *The lower bound of the total time cost saving for our algorithm to evaluate all CNs is $o(\sum_{i=1}^{\mathsf{Tmax}} i \cdot |S_i|)$, where S_i is the set of partial connected trees with size i generated by the algorithm proposed in [26].*

Proof Sketch: For each connected tree with size i ($1 \leq i \leq \mathsf{Tmax}$), the algorithm proposed in [26] needs at least $o(i)$

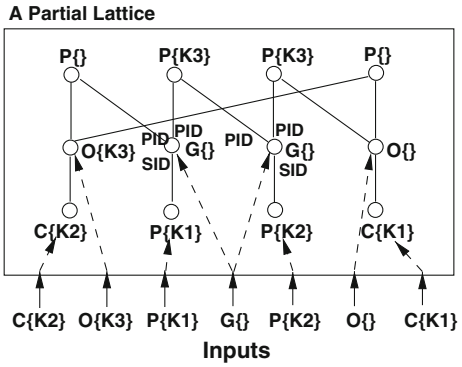


Fig. 11 Lattice and its inputs from a stream

Table 2 Meaning of each parameter

Name	Meaning
W	The size of the sliding window, in minutes
KWF	The probability that a keyword appears in a tuple
$\|V\|$	The number of relations in the schema
$1/sel$	The join selectivity of two relations in the dataset
m	The number of keywords in the query
Tmax	The maximum number of tuples allowed in each result

time to generate it. In our algorithm, we only need to generate the root node with size 1 using the hash table. As a result, we can save at least $o(\sum_{i=1}^{\text{Tmax}} i \cdot |S_i|)$ time in total. Since generating a large number of partial connected trees is the main cost of all algorithms, the lower bound of cost saving of our algorithm is still large.

Finally, we discuss how to implement the event-driven evaluation. As shown in Fig. 11, there are multiple nodes labeled the same $R_i\{K'\}$. For example, $G\{\}$ appears in two different nodes in the lattice. For each $R_i\{K'\}$, we maintain 3 lists named `Rlist` (Ready), `Wlist` (Wait) and `Slist` (Suspend). The three lists contain all the node ids in the lattice. A node in the lattice $\mathcal{L}$ labeled $R_i\{K'\}$ can only appear in one of the three lists for $R_i\{K'\}$. A node v in $\mathcal{L}$ appears in `Wlist`, if the sub-relations represented by all child nodes of v in $\mathcal{L}$ are non-empty, but the sub-relation represented by v is empty. A node v in $\mathcal{L}$ appears in `Rlist`, if the sub-relations represented by all child nodes of v in $\mathcal{L}$ are non-empty, and the sub-relation represented by v itself is non-empty too. Otherwise, v appears in `Slist`. When a new tuple t of relation R_i with keyword set K' is inserted, we only insert it into all relations in the nodes v, in $\mathcal{L}$, on `Rlist` and `Wlist` specified for $R_i\{K'\}$. Each insertion may notify some father nodes of v to move from `Wlist` or `Slist` to `Rlist`. The node v may also be moved from `Wlist` to `Rlist`. When a tuple t of relation R_i with keyword set K' is about to be deleted, we only remove it from all relations associated with node v, in $\mathcal{L}$, on `Rlist` specified for $R_i\{K\}$. Each deletion may notify some father nodes of v to be moved from `Rlist` or `Wlist` to `Slist` and v may also be moved from `Rlist` to `Wlist`.

7 Performance studies

We conducted extensive experiments to test the efficiency of our InitNTGen algorithm for generating *CN*s and our *CN* evaluation algorithm CNEvalDynamic. Note that CNEvalStatic will be used only when there exists a database already at the time it needs to process a data stream. As default, we assume that there is no such a database. We compare our algorithms with the up-to-date algorithms given in [26], namely InitCNGen (for *CN* generation), FM (Full-Mesh) (for *CN* evaluation), and PM (Partial-Mesh) (for *CN* evaluation). The two algorithms Full-Mesh and Partial-Mesh are all proposed in [26] to evaluate the set of *CN*s generated. Partial-Mesh grows and shrinks the operator mesh at runtime and can dramatically reduce the number of temporal tuples generated comparing to the Full-Mesh algorithm. All algorithms are implemented in C++. We conducted all the experiments on a 2.8 GHz CPU and 2 GB memory PC running XP.

We test the algorithms to process m-keyword queries on the condition that the size of a *MTJNT* is up to Tmax, in terms of nodes in a *MTJNT*. We use both synthetic datasets real datasets to conduct our tests. For all tests, we record the total CPU time used for processing the whole data stream in the corresponding dataset.

The synthetic dataset is specified in [21] and is the same synthetic dataset used in [26]. In the synthetic dataset, the schema graph, $G_S(V, E)$, is a tree-structured schema graph. A relation can join with up to 4 relations. In all relations, all attribute values are randomly and independently generated in the range of $[1, sel]$. Then, the join selectivity between two relations that have a primary/foreign relationship is $1/sel$. As indicated in [26], a tuple may contain several different keywords where each keyword is with an independent probability KWF. The sliding window size is W minutes. Over a data stream, a tuple can be inserted/deleted into/from every relation in every second. If there are $|V|$ relations, at a single second, there are $|V|$ insertions/deletions simultaneously. The entire time window for the whole data stream is 5 h. We also list the meaning of each parameter in Table 2. The parameters with their default values (bold) are shown in Table 3 for the synthetic dataset. With the default parameters used in the stream, there are totally 522,000 operations including 270,000 insertions and 252,000 deletions, and the total CPU time cost for all operations to evaluate the default query

Table 3 Parameters for synthetic dataset

Parameter	Range & default
W	5, 10, **20**, 40, 80 min
KWF	0.003, 0.007, **0.01**, 0.013, 0.016
$\|V\|$	5, 10, **15**, 20, 25
sel	500, 750, **1000**, 1250, 1500
m	2, **3**, 4, 5
Tmax	2, 3, **4**, 5, 6

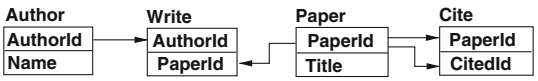


Fig. 12 DBLP database schema

Table 4 Parameters for DBLP dataset

Parameter	Range & default
W	30, 40, **50**, 60, 70
KWF	0.003, 0.007, **0.01**, 0.013
m	2, **3**, 4, 5
Tmax	3, 5, **7**, 9

using our algorithm is about 4 s and the average time for each insertion/deletion operation is 0.0077 ms.

The real dataset we used to simulate a data stream environment is 2009 DBLP (http://dblp.uni-trier.de/xml/). Note: DBLP is not a data stream dataset. The reason we use DBLP to simulate a data stream is because that there is a lack of real data stream datasets, and many reported studies on top-k keyword queries over RDBMS use DBLP, for example [25]. In order to make it more similar to a real stream environment, we first order all papers in DBLP in increasing order of the time they are published, and then order the tuples in the other three tables accordingly. The schema of DBLP is shown in Fig. 12, which shares the same structure as we used in E-Commerce (Fig. 1). In DBLP, in total, there are 5,009,514 nodes and 6,225,324 edges. The sizes of Author, Write, Paper, and Cite relations are, 705,163, 3,000,289, 1,191,689, and 112,373, respectively. The parameters with their default values (bold) are shown in Table 4 for DBLP. In Table 4, the sliding window size W is x. It implies that every relation in DBLP will keep x of its data in the sliding window in a data stream. The reason is that it is difficult to set up a window using time for DBLP, and it is difficult to find results if there are too small numbers of tuples in a window using DBLP. The number of keywords, m, tested is 2, 3, 4, and 5. The m keywords are selected as the first m keywords from a list of keywords (analysis, design, system, networks, model). To test KWF, the keywords selected are listed in Table 5. With the default parameters used in the stream, there

Table 5 KWF and the keywords used

KWF	Keywords
0.003	fuzzy, oriented, evaluation
0.007	time, information, algorithm
0.01	analysis, design, system, networks, model
0.013	systems, data, using

Table 6 Parameters for TPC-H dataset

Parameter	Range & default
W	30, 40, **50**, 60, 70
KWF	0.003, 0.007, **0.01**, 0.013
m	2, **3**, 4, 5
Tmax	2, 3, **4**, 5, 6

are totally 7,514,271 operations including 5,009,514 insertions and 2,504,757 deletions. The total CPU time cost for all operations to evaluate the default query using our algorithm is about 60 s and the average time for each insertion/deletion operation is 0.008 ms.

We also use the benchmark dataset TPC-H (http://www.tpc.org/tpch/) which records the transactions between customers and products. The TPC-H dataset is used for testing the performances of commercial DBMSs. In the TPC-H dataset, all tuples come in and out in increasing order of the time the corresponding transaction was made. In the schema of the TPC-H dataset, there are eight relations and 31 attributes and in all relations, there are totally 8,661,245 tuples. The parameters with their default values (bold) are shown in Table 6 for TPC-H. All parameters have the same meaning as in the *DBLP* dataset. With the default parameters used in the stream, there are totally 12,991,907 operations including 8,661,245 insertions and 4,330,622 deletions. The total CPU time cost for all operations to evaluate the default query using our algorithm is about 130 s and the average time for each insertion/deletion operation is 0.01 ms.

7.1 Exp-1: Test synthetic dataset

First, we test our new template-based InitNTGen algorithm in comparison with InitCNGen algorithm [26] for *CN* generation. The default values are Tmax $= 4$, $m = 3$, $|V| = 15$, which are the same as used in [26]. Note: $|E| = |V| - 1$, because the schema graph is a tree. The results are shown in Fig. 13. In Fig. 13a, we vary Tmax from 2 to 6, in Fig. 13b, we vary m from 2 to 5, and in Fig. 13c, we vary $|V|$ from 5 to 25, where $|E| = |V| - 1$. Also, we further increase $|E|$ from 16 to 34 with $|V| = 15$. As can be seen from all the cases, our InitNTGen significantly outperforms InitCNGen.

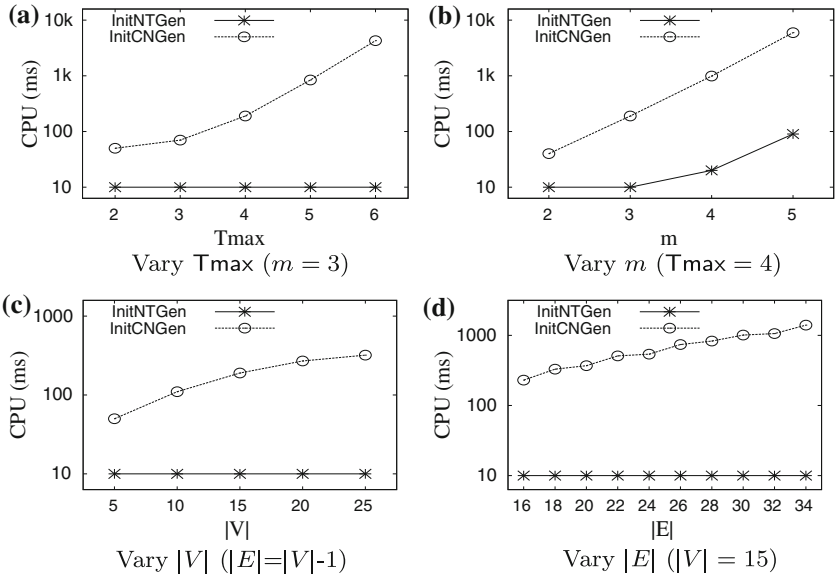


Fig. 13 *CN* Generation (synthetic schema)

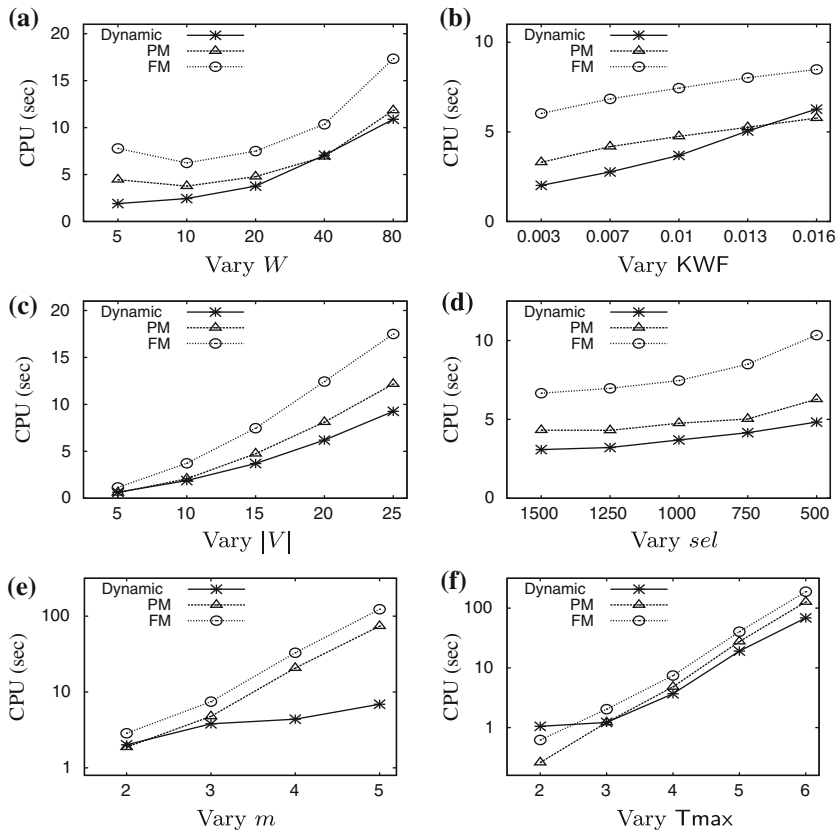


Fig. 14 *CN* Eval (synthetic dataset)

For example, when $m = 3$ and Tmax $= 6$, our InitNTGen took 10 ms, whereas InitCNGen took over 5,000 ms.

Second, we test our proposed CNEvalDynamic (denoted `Dynamic` for short) with Full-Mesh (denoted FM) and Partial-Mesh (denoted PM) over a data stream from the first time a tuple arrives for 5 h. We tested CPU time and memory consumption. Due to space limit, we do not report memory consumption for the synthetic dataset and will report memory consumption for the real dataset `DBLP`. The unit for CPU time is second. In Fig. 14, the CPU time when varying W from 5 to 80 min is shown in Fig. 14a, and the CPU time when varying KWF from 0.003 to 0.016 is shown in Fig. 14b. Figure 14c and d show the CPU times when varying $|V|$ (from 5 to 25) and when varying $1/sel$ (from $1/1,500$ to $1/500$),

respectively. The CPU times when varying m (from 2 to 5) and when varying Tmax (from 2 to 6) are shown in Fig. 14e and f, respectively. Our CNEvalDynamic significantly outperforms FM and PM. Among all the testings with various parameters, our CNEvalDynamic is not significantly affected by the changes of W, KWF, $|V|$, sel, and m. The reason that it is not affected by m significantly is as follows. When m increases, on one hand the number of trees increases, and on the other hand the cost to evaluate each tree decreases because there are more keywords (more nodes will contain keywords or nodes will contain more keywords on average). For FM and PM, the former cost is the dominant cost. CNEvalDynamic saves computational cost when evaluating every tree, and the impact of the number of trees is not noticeable. As shown in Fig. 14f, Tmax has great impacts on m-keyword query processing in terms of CN evaluation. It is mainly because it implies the number of joins needed to be conducted.

7.2 Exp-2: Test DBLP dataset

The CPU time for generating CNs based on the DBLP database schema is given in Fig. 4, because the database schema for DBLP is the same as the E-Commerce database schema. Both have the same number of relations and the same primary/foreign key relationships. As shown in Fig. 4, our InitNTGen only takes about 10 ms in most cases, whereas InitCNGen may take long time in the range from 100 to 100,000 ms. We test our new CNEvalDynamic (Dynamic) in comparison with Full-Mesh (FM) and Partial-Mesh (PM). We report CPU time and memory consumption for all the cases in Fig. 15. The unit for CPU time is second, and the unit for memory is the numbers of intermediate tuples. Note: for each intermediate tuple, FM or PM consumes more memory space than CNEvalDynamic. It is because that FM or PM needs to maintain all the join attributes that may be needed to join, whereas CNEvalDynamic only needs to keep tuple ids with a lattice id to record where the tuple is from.

Figure 15a and b show the CPU time and the peak number of intermediate tuples when varying W. As can be seen, CNEvalDynamic performs in constant time, whereas the CPU time of FM or PM increases exponentially. CNEvalDynamic performs less than 100 s in all the testing, whereas FM cannot continue because it runs out of memory when $W > 50$ of the dataset. The reason can be explained by the number of intermediate tuples to be computed. As shown in Fig. 15b, FM needs to maintain a large number of intermediate tuples. When $W = 50$, FM needs to maintain 10M intermediate tuples, whereas CNEvalDynamic only maintains up to 3M intermediate tuples. PM performs better than FM because the number of temporal tuples generated by the PM algorithm is much smaller than that generated by the FM algorithm. Fig. 15c and d show the CPU time and the peak number of intermediate tuples when varying KWF. Comparing Fig. 15c with Fig. 14b, the CPU times of FM or PM seem to behave differently in the two datasets. As shown in Fig. 14b, with the synthetic dataset, the CPU time grows when KWF increases for both algorithms, but the CPU time for FM or PM may decrease when KWF increases, with DBLP. This is because that all the pairs of tuples, (t_i, t_j), have the same probability to be joined in the synthetic dataset. But in DBLP dataset it is not true, because the probability of a pair of tuples, (t_i, t_j), being joined varies. Even when KWF $= 0.003$, it is still likely that there are a large number of tuples that contain some keywords, need to join, which results in a large number of intermediate tuples. Figure 15e and f show the CPU time and the peak number of intermediate tuples when varying m. Our CNEvalDynamic algorithm performs much better than both FM and PM in all cases. When $m = 5$, CNEvalDynamic consumes 150 s and generates 2M temporal tuples, while both FM and PM consume nearly 10,000 s and generate more than 10 M temporal tuples. It is because when m is large, more tuples will be involved in joining. In both FM and PM algorithms, much time is spent on performing such joining operations, while only few of them will contribute to the final results. In our CNEvalDynamic algorithm, we only perform joins when they contribute to the final results, thus the computational cost for such joining operations is largely saved. In Fig. 15g and h, when Tmax increases, the CPU time and peak number of intermediate tuples for all three algorithms increase. CNEvalDynamic performs best especially when Tmax is large. Such results confirm Theorem 6.5, where the computational cost saving of CNEvalDynamic is exponential with respect to Tmax.

7.3 Exp-3: Test TPC-H dataset

We tested the TPC-H dataset for both CN generation and CN evaluation. The results are shown in Fig. 16. The time costs for generating CNs in the TPC-H dataset are shown in Fig. 16a and b when varying Tmax and m, respectively. Our InitNTGen algorithm outperforms InitCNGen algorithm in all cases and is 20 times faster on average. Figure 16c shows the time cost on CN evaluation for all three algorithms when varying the window size W. CNEvalDynamic outperforms FM in all cases. When W is small, CNEvalDynamic and PM have similar performances, because when W is small, the number of tuples for each relation is also small. When evaluating CNs, the sizes of intermediate results for both CNEvalDynamic and PM are all small, thus the CNEvalDynamic algorithm does not outperform PM much. When W is large, CNEvalDynamic is 2–3 times faster than PM because PM generates a large number of intermediate results and CNEvalDynamic is still stable. Figure 16d

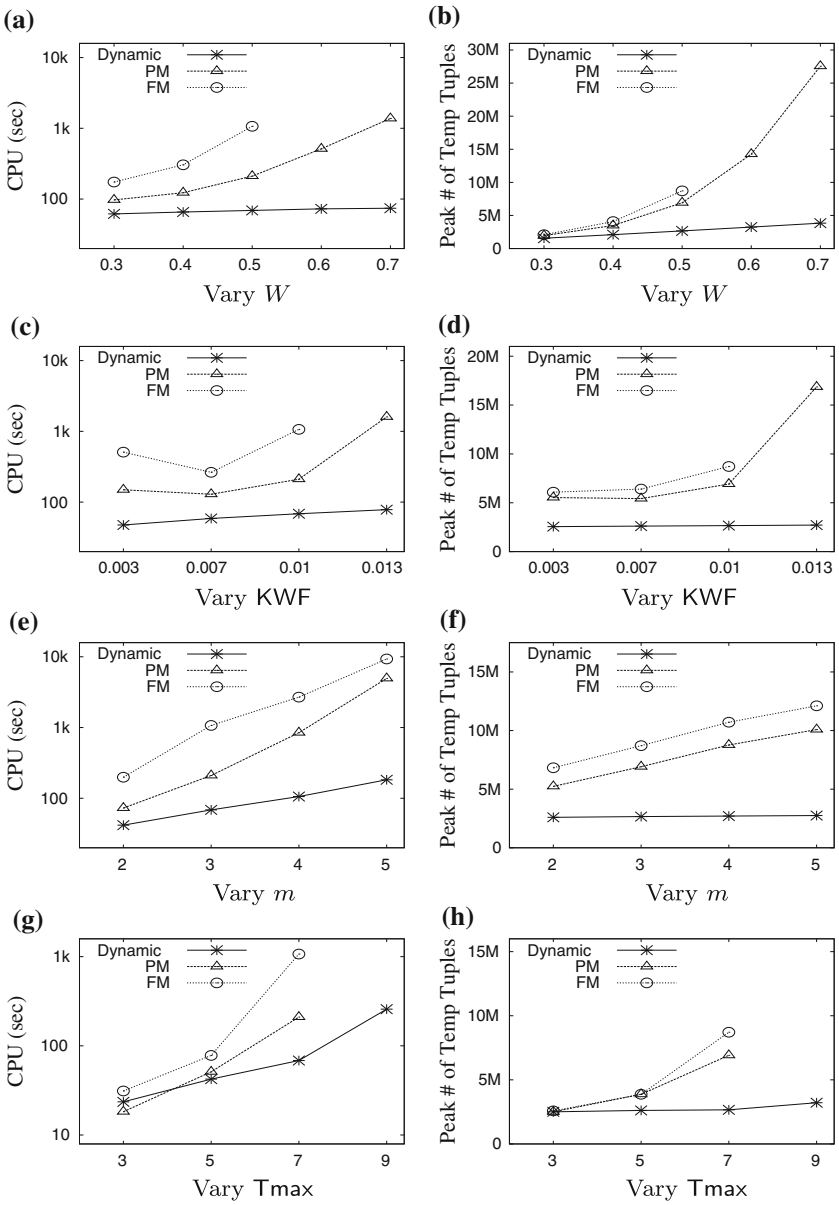


Fig. 15 *CN* Eval (DBLP)

shows the CPU time to evaluate all *CN*s when varying KWF. Our CNEvalDynamic algorithm outperforms FM and PM in most cases, and when KWF is large, CNEvalDynamic is 2–3 times faster than PM and 10 times faster than FM. Figure 16e and f illustrate the results for the *CN* evaluation algorithms when varying m and Tmax, respectively. In all cases, CNEvalDynamic is 2–10 times faster than PM and 5–50 times faster than FM.

Table 7 shows the number of *CN*s generated in TPC-H for all $2 \leq m \leq 5$ and $2 \leq \mathsf{Tmax} \leq 10$. Compared to the result in Table 1, for $m = 5$ and $\mathsf{Tmax} = 10$, when the number of nodes in the schema graph increases from 1 to 8, the number of joining plans increases from 83, 512 to 30, 435, 766. When the complexity of the schema graph further increases, it is even impossible to generate the extremely large number of *CN*s. This is also a drawback of all *CN*-based approaches for keyword search in relational databases.

7.4 Exp-4: Test CNEvalStatic

In this test, we assume that when the system needs to provide services to process m-keyword queries, there already exists a not small amount of data which is needed to be considered in processing m-keyword queries. Further, we assume that the sliding window $W = \infty$. In such a case, there are two ways to process. One is to treat every tuple in the database as a new arrival tuple and use CNEvalDynamic to process. The other is to process data in the database using CNEvalStatic, and then use CNEvalDynamic to process data from a data stream. We conduct testing to test the two approaches using

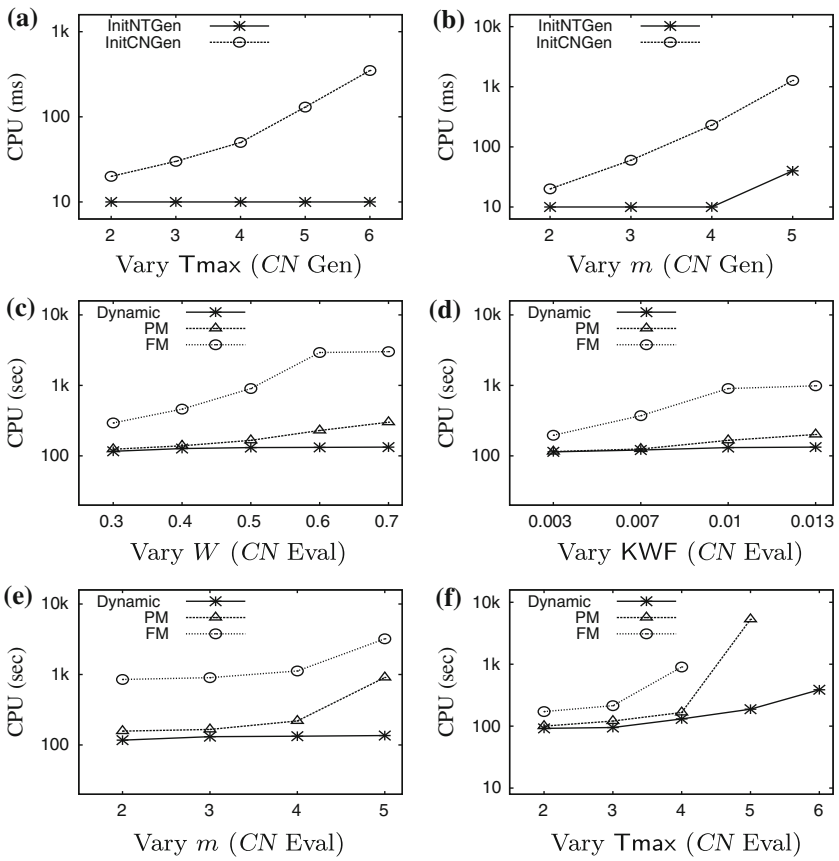


Fig. 16 Scalability of TPC-H

Table 7 The number of *CN*s in TPC-H

Tmax	*m*			
	2	3	4	5
2	24	56	120	248
3	52	224	820	2,768
4	94	649	3600	17,793
5	161	1,633	12,705	85,803
6	261	3,676	38,193	337,061
7	427	7,947	105,532	1,171,151
8	685	16,404	271,386	3,694,081
9	1,120	33,378	669,564	10,919,751
10	1,790	65,719	1,579,082	30,435,766

all synthetic data and real data. We show the results for the synthetic dataset and the DBLP dataset. For the TPC-H dataset, the curves are also similar to the synthetic dataset and DBLP dataset. The results are shown in Fig. 17. The horizontal axis is the percentage of the entire dataset that is supposed to exist already. Let it be x . The CNEvalDynamic approach is to treat the entire dataset as a data stream to process. The CNEvalStatic/CNEvalDynamic approach is to use CNEvalStatic to process the first x of data, and then use CNEvalDynamic to process the remaining $1-x$ of data. As can be seen in Fig. 17, the CNEvalStatic/CNEvalDynamic approach significantly outperforms the CNEvalDynamic approach. The main reason is as follows. When evaluating each node in the lattice $\mathcal{L}$ using the CNEvalStatic algorithm, all the tuples that can be joined with tuples in the child nodes can be generated together. It is possible that a tuple can join with several tuples in the child nodes. In such a situation, once a tuple is generated, it is used only once for joining with tuples in the father nodes. Using the CNEvalDynamic approach, new tuples will come in one by one, and once a tuple t is processed in a certain node of the lattice $\mathcal{L}$, all tuples that can join the tuple t will be processed in the father nodes. In such a way, when multiple tuples (in the child nodes) that can join t comes in, t will be processed for multiple times to make sure that the lattice maintains the correct set of results. Such computational cost can be avoided in the CNEvalStatic algorithm.

Figure 17 shows that if a partial database already exists when the new query is registered, the performance of the CNEvalDynamic algorithm can be further improved using the CNEvalStatic/CNEvalDynamic algorithm. Note that both the FM algorithm and the PM algorithm assume that the query is registered before the data comes. The CNEvalStatic/CNEvalDynamic algorithm can improve FM and PM using the same way when part of the database already exists before registering the query. For reference, we also list the time consumption for the same

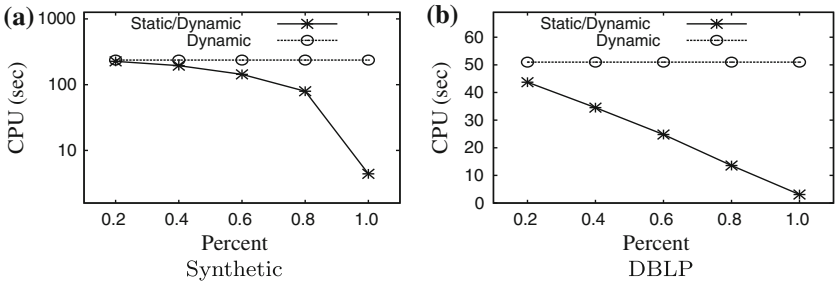


Fig. 17 CNEvalStatic/CNEvalDynamic vs CNEvalDynamic: **a** Synthetic ($m = 4$, Tmax $= 5$, $|V| = 15$, and $|E| = 14$) **b** DBLP ($m = 3$, Tmax $= 7$)

testing case using FM and PM below. For the synthetic dataset, FM consumes 1,033 s and PM consumes 1,560 s. For the *DBLP* dataset, FM consumes 230 s and PM consumes 732 s. All of them are worse than CNEvalDynamic and CNEvalStatic/CNEvalDynamic.

8 Related work

Keyword search in relational databases has been studied extensively recently. The techniques to answer static keyword queries on RDB are mainly in two categories: *CN*-based (schema-based) and graph-based (schema-free) approaches.

In the *CN*-based approaches [1,16,14,25,26], it processes an m-keyword query in two steps, candidate network (*CN*) generation and *CN* evaluation. *CN* evaluation is done using SQL on RDBMS. *DBXplorer* [1] and *DISCOVER* [16] focused on retrieving connected trees using SQL on RDBMS. In [16], Hristidis and Papakonstantinou proved how to generate a complete set of *CN*s to find all *MTJNT*s when the size of *MTJNT*s is at most allowed by a user-given Tmax (the number of nodes in *MTJNT*s) and discussed several query processing strategies with possible query optimization. All the above work [1,16,26] focused on finding all *MTJNT*s, whose sizes are $\leq$ Tmax, which contain all m keywords (AND-semantics), and there is no ranking involved.

Among *CN*-based approaches, in *DISCOVER-II* [14], Hristidis et al. incorporated IR-style ranking techniques to rank the connected trees. Two algorithms, sparse and global pipeline, were proposed in [14] to stop the query execution as soon as the top-k results are returned. *DISCOVER-II* is built in a middleware on top of RDBMS and issues SQL queries to access data. In *SPARK* [25], Luo et al. proposed a new ranking function by treating each connected tree as a virtual document, and unified the AND/OR semantics in the score function using a parameter. A monotonic upper bound score function was proposed to handle a non-monotonic score function. Two sweeping schemes, skyline sweep and block pipeline, proposed in *SPARK* were demonstrated to outperform *DISCOVER-II* [14]. Both work [14,25] focused on finding top-k *MTJNT*s, whose sizes are controlled by Tmax, which contain all or some of the m keywords (AND/OR-semantics). The ranking issues were also discussed in [4,15,24].

Finding top-k interconnected structures have been extensively studied in graph-based approaches in which an RDB is materialized as a weighted database graph $G_D(V, E)$. Here, V is the set of tuples in RDB, and E is the set of edges, where an edge appears in E, if there is a foreign key reference between the two nodes involved in the edge.

The representative work on finding top-k connected trees is [6,18,20,9,12]. In brief, finding the exact top-k connected-trees is an instance of the group Steiner tree problem [10], which is NP-hard. To find top-k connected trees, Bhalotia et al. proposed backward search in *BANKS-I* [6], and Kacholia et al. proposed bidirectional search in *BANKS-II* [18]. Kimelfeld et al. [20] proposed a general framework to retrieval top-k connected trees with polynomial delay under data complexity, which is independent of the underline minimum Steiner tree algorithm. Ding et al. in [9] also introduced a dynamic programming approach to find the minimum connected tree and approximate top-k connected trees. Golenberg et al. in [12] tended to find an approximate result in polynomial time under the query and data complexity.

Top-k connected trees are hard to compute. He et al. proposed the distinct root semantics in *BLINKS* [13]. In *BLINKS*, new search strategies were proposed with a bi-level index built to fast compute the shortest distances. *BLINKS* is a memory-based algorithm, which performs best when the bi-level index is in memory. In order to deal with large scale graphs, when the entire index cannot resident in memory, Dalvi et al. [7] conducted keyword search on external memory graphs under the distinct root semantics.

Li et al. in *EASE* [22] defined a r-radius Steiner graph, where each r-radius Steiner graph is a subpart of a maximal r-radius subgraph. All the maximal r-radius subgraphs are precomputed and stored on disk, using an extended inverted index with keywords as the entries. To answer a keyword query, *EASE* [22] retrieves the relevant maximal r-radius subgraphs from disk and returns the retrieved maximal r-radius subgraphs by removing the irrelevant nodes from the retrieved maximal r-radius subgraphs. Qin et al. studied multi-center communities under the distinct core semantics

in [27] and proposed new polynomial delay algorithms to compute all or top-k communities.

All the above works deal with static data, except for the work of [26] which conducts keyword search on relational data streams. There are also reported studies on continuous keyword search in other data stream environments. [30] and [11] focused themselves on a single textual document stream, where different documents do not need to be joined when processing streams. Hristidis et al. in [17] studied continuous keyword search on multiple text streams, a result is a tree of streams that collectively contain all the query keywords, where two streams are connected if they are correlated, i.e., they have similar topics. Yang and Shi [31] studied keyword search on an XML stream which consists of a single stream of an XML document. Li et al. [23] studied how to identify interesting results on XML streams. All these works of keyword search on text streams or XML streams do not need to conduct complex joins.

Besides the works of keyword search on data streams, there are works of sliding window join processing on data streams. Kang et al. [19] investigated the problem of how to efficiently evaluate sliding window join operators over pairs of unbounded streams. More works are conducted to evaluate the sliding window join operators with limited resources, specifically, the memory limited case was addressed in [8] and [29], and the CPU limited case was addressed in [3]. Note that these works are different from ours in that they only consider one query by joining two streams, while ours involve multiple queries with multiple join operators. Ayad and Naughton [2] studied a more general conjunctive query with sliding window over data streams, but they only consider one query while our work involves multi-queries.

9 Conclusion

In this paper, we studied m-keyword query processing on large relational data streams. We process such queries in the framework of RDBMS in two steps: (1) *CN* generation and (2) *CN* evaluation. In the *CN* generation step, we introduced a novel algorithm to efficiently generate all the *CN*s using a template-based approach. All the *CN*s are compacted in an lattice, $\mathcal{L}$, which is used for *CN* evaluation. In the *CN* evaluation step, we proposed CNEvalStatic and CNEvalDynamic algorithms to significantly reduce the large number of intermediate results to be computed. Our algorithms significantly outperform the up-to-date algorithms.

Acknowledgments We thank A Markowetz, Y Ying, and D. Papadias for providing their FM and PM code. The work was supported by grants of the Research Grants Council of the Hong Kong SAR, China, No. CUHK/419008 and CUHK/419109.

References

1. Agrawal, S., Chaudhuri, S., Das, G.: DBXplorer: A system for keyword-based search over relational databases. In Proceedings of ICDE'02 (2002)
2. Ayad, A., Naughton, J.F.: Static optimization of conjunctive queries with sliding windows over infinite streams. In Proceedings of SIGMOD'04, pp. 419–430 (2004)
3. Ayad, A., Naughton, J.F., Wright, S., Srivastava, U.: Approximating streamingwindow joins under cpu limitations. In Proceedings of ICDE'06, pp. 142 (2006)
4. Balmin, A., Hristidis, V., Papakonstantinou, Y.: ObjectRank: authority-based keyword search in databases. In Proceedings of VLDB'04 (2004)
5. Bernstein, P.A., Chiu, D.-M.W.: Using semi-joins to solve relational queries. J. ACM **28**(1), 25–40 (1981)
6. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. In Proceedings of ICDE'02 (2002)
7. Dalvi, B.B., Kshirsagar, M., Sudarshan, S.: Keyword search on external memory data graphs. PVLDB **1**(1), 1189–1204 (2008)
8. Das, A., Gehrke, J., Riedewald, M.: Approximate join processing over data streams. In Proceedings of SIGMOD'03, pp. 40–51 (2003)
9. Ding, B., Yu, J.X., Wang, S., Qin, L., Zhang, X.,Lin, X.: Finding top-k min-cost connected trees in databases. In Proceedings of ICDE'07 (2007)
10. Dreyfus, S.E., Wagner, R.A.: The steiner problem in graphs. Networks **1**, 195–207 (1972)
11. Fabret, F., Jacobsen, H.-A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering algorithms and implementation for very fast publish/subscribe. In Proceedings of SIGMOD'01 (2001)
12. Golenberg, K., Kimelfeld, B., Sagiv, Y.: Keyword proximity search in complex data graphs. In Proceedings of SIGMOD'08 (2008)
13. He, H., Wang, H., Yang, J., Yu, P.S.: BLINKS: ranked keyword searches on graphs. In Proceedings of SIGMOD'07 (2007)
14. Hristidis, V., Gravano, L., Papakonstantinou, Y.: Efficient IR-Style keyword search over relational databases. In Proceedings of VLDB'03 (2003)
15. Hristidis, V., Hwang, H., Papakonstantinou, Y.: Authority-based keyword search in databases. ACM Trans. Database Syst. **33**(1) (2008)
16. Hristidis, V., Papakonstantinou, Y.: DISCOVER: keyword search in relational databases. In Proceedings of VLDB'02 (2002)
17. Hristidis, V., Valdivia, O., Vlachos, M., Yu, P.S.: Continuous keyword search on multiple text streams. In Proceedings of CIKM'06 (2006)
18. Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R., Karambelkar, H.: Bidirectional expansion for keyword search on graph databases. In Proceedings of VLDB'05 (2005)
19. Kang, J., Naughton, J.F., Viglas, S.: Evaluating window joins over unbounded streams. In Proceedings of ICDE'03, pp. 341–352 (2003)
20. Kimelfeld, B., Sagiv, Y.: Finding and approximating top-k answers in keyword proximity search. In Proceedings of PODS'06 (2006)
21. Krämer, J., Seeger, B.: Pipes—a public infrastructure for processing and exploring streams. In Proceedings of SIGMOD'04 (2004)
22. Li, G., Ooi, B.C., Feng, J., Wang, J., Zhou, L.: EASE: efficient and adaptive keyword search on unstructured, semi-structured and structured data. In Proceedings of SIGMOD'08 (2008)

23. Li, L., Wang, H., Li, J., Gao, H.: Efficient algorithms for skyline top-k keyword queries on xml streams. In Proceedings of DASFAA'09, pp. 283–287 (2009)
24. Liu, F., Yu, C.T., Meng, W., Chowdhury, A.: Effective keyword search in relational databases. In Proceedings of SIGMOD'06 (2006)
25. Luo, Y., Lin,X., Wang, W., Zhou, X.: Spark: top-k keyword query in relational databases. In Proceedings of SIGMOD'07 (2007)
26. Markowetz, A., Yang, Y., Papadias, D.: Keyword search on relational data streams. In Proceedings of SIGMOD'07 (2007)
27. Qin, L., Yu, J.X., Chang, L., Tao, Y.: Querying communities in relational databases. In Proceedings of ICDE'09 (2009)
28. Qin, L., Yu, J.X., Chang, L., Tao, Y.: Scalable keyword search on large data streams. In Proceedings of ICDE'09, pp. 1199–1202 (2009)
29. Srivastava, U., Widom, J.: Memory-limited execution of windowed stream joins. In Proceedings of VLDB'04, pp. 324–335 (2004)
30. Yan, T.W., Garcia-Molina, H.: The sift information dissemination system. ACM Trans. Database Syst. **24**(4), 324–335 (1999)
31. Yang, W., Shi,B.: Schema-aware keyword search over xml streams. In Proceedings of CIT'07, pp. 29–34 (2007)