Efficient Keyword Search over Relational Data Streams

Savong BOU†, Toshiyuki AMAGASA††, and Hiroyuki KITAGAWA††

† Graduate School of Systems and Information Engineering, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
†† Center for Computational Sciences, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
E-mail: †savong.bou@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.ac.jp

Abstract Keyword search over relational streams plays an important role when enabling users to issue queries without knowing the details about information sources as well as query languages. The state-of-the-art approaches exploit Candidate Networks (CNs), which are high-level (or schema-level) representation of joining networks of tuples containing all query keywords, to filter relational streams. However, in fact, the performance of these approaches seriously degrades in particular when the maximum size of CNs (T_{max}) and/or the number of query keywords are large, due to the explosive increase in the number of CNs. To cope with this problem, this paper proposes a scalable approach that consolidates all CNs in such a way that all common edges in the CNs can share processing, thereby avoiding unnecessary evaluation of different CNs as much as possible. The experimental results reveal that the query execution time of the proposed approach greatly outperforms the comparative methods.

Key words Keyword Search, Relational Data Streams

1. Introduction

With the recent trends of Cyber Physical Systems [6, 11], Internet of Things [5, 15], etc., the number of real-time information sources has been explosively increasing. Besides, it has become common to extract information from various social medias, such as Twitter, and Facebook, in real-time for making analysis of diverse social activities. Typically, each data item generated at stream data sources can be modeled as structured records. For this reason, such data sources are often called *relational streams* [3], where structured records that conform to some predefined relational schemas are continuously generated. Therefore, the importance of query processing over relational streams has been increasing.

Processing *keyword search* over relational streams is considered to be a practical approach due to its user-friendliness, which does not requires users to learn neither (potentially) complicated query language, like CQL [3], nor the schemas of streams being queried, which is also very complicated in many real applications. So far, keyword search over permanently-stored relational data [2, 7, 8, 10, 12] has been extensively studied, but only a few works have addressed keyword search over relational streams [9, 13].

In the works [9, 13], they employ candidate network-based approach for better performance. Specifically, all candidate networks (CNs) that represent all possible combinations of keyword occurrences on join paths are enumerated. Then, they are merged to generate a query plan by which actual relational streams are processed. More precisely, in S-KWS [9], a set of CNs are merged only if they share at least one leaf node called *root*, and possible subtrees are merged to remove redundant processing as much as possible. In SS-KWS [13], common partial networks are merged more aggressively from every leaf node of CNs, thereby generating more compact query plans.

However, it should be noted that the performance of S-KWS and SS-KWS considerably degrades when the number of query keywords and/or network size (T_{max}) are increased. The increase of these two parameters causes rapid increase in the number of CNs, which results in a lot of common partial networks remain unintegrated. To exemplify the problem, let us take TPC-H dataset [1] as an example. When the number of keywords and T_{max} are increased from four to five, the number of CNs increases from 3,600 to 85,803 [13]. Likewise, the total number of edges in the query execution plans exponentially increases from 4,276 to 73,596 in S-KWS and from 7,486 to 222,040 in SS-KWS. (More detailed discussion can be found in Section 5. 1.) Thus the performance of S-KWS and SS-KWS would deteriorate in particular when dealing with a lot of query keywords and/or large relational streams consisting of many relations.

How can we cope with such exponential blow up of CNs and the complication of query plans? If we consider the edges in CNs, each of them actually represents primary/foreign-key

relationships between two tables. For this reason, in the existing approaches, even though the number of edges grows at an exponential rate, most of them, which are created from smaller number of primary/foreign-key relationships in the original schema, are actually redundant. With the same example above when the number of keywords and T_{max} are increased from four to five, the total number of unique edges in all CNs grows linearly from 1,088 to 3,536. Under this observation, to cope with the problem of CN's exponential blow up, it is possible to consolidate the edges sharing the same primary/foreign-key relationship into one edge when generating a query plan, which leads to great performance improvement.

This paper proposes a novel approach to processing keyword search over relational streams by taking into account the above idea. Specifically, an *MX-structure* is proposed to consolidate common edges in different CNs as much as possible. The experimental results prove that the proposed approach can process relational streams about 40 to 70 times faster than the state-of-the-art approaches when the number of keywords and T_{max} are increased to five.

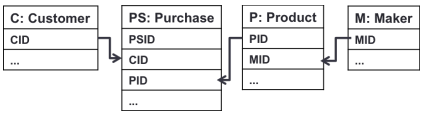
This paper is organized as follow. We introduce related works in Section 2.. Section 3. introduces existing works of keyword search over relational streams, and the proposed approach is in Section 4.. Experiments and conclusion are in Section 5. and 6. respectively.

2. Related Works

So far, many proposals have been done to enable keyword search on permanently-stored-relational data [2,4,7,8,12] and few proposals on relational streams [9, 13].

DISCOVER [8] and DBXPLORER [2] are the CN-based keyword search on (static) relational data. In DISCOVER [8], first all CNs are generated from the given keyword search and relational data's schema. Then, a plan is built for efficient evaluation of all CNs by reusing some common edges of CNs. DBXPLORER-II [7] adopts IR-style document-relevance ranking technique to keyword search over relational data. Since the total number of CNs can be very big, and evaluation of all CNs is costly, [12] proposes an algorithm to rank all CNs, and only top-k CNs are chosen to evaluate against relational data.

S-KWS [9] is the first work to enable keyword search over relational streams. It is also a candidate network based approach, which makes use of common edge attached to root nodes to elevate query searching. And the most recent work to enable keyword search on relational streams is SS-KWS [13], which is proved to outperform S-KWS [9] when most tuples from the relational data streams mostly match CNs that have common edges at leaf-nodes.



(a) Schema.

C			
CID	Name	Addr	Time
c1	Kitagawa	Tokyo	1
c2	Amagasa	Ibaraki	3

PS				
PSID	CID	PID	Note	Time
ps1	c1	p1	Urgent del.	6
ps2	c3	p1	Kyoto bran.	8

P				
PID	MID	Name	Price	Time
p1	m1	TV	52,154¥	2
p2	m2	*Laptop*	95,487¥	4

M			
MID	Name	Addr	Time
m1	Hitachi	Ibaraki	5
m2	*Panasonic*	Iwate	7

(b) Instances.

Figure 1 A sample e-commerce application.

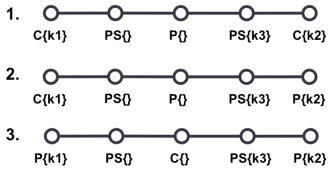
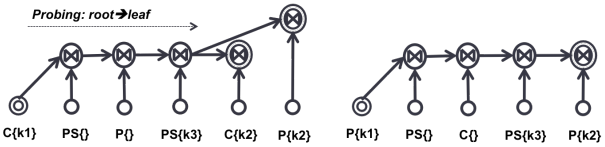


Figure 2 Some CNs created from schema in Figure 1(a) for query $\{k_1, k_2, k_3\}$.



(a) Cluster #1. (Combine the first two CNs.)

(b) Cluster #2. (Only CN 3 is included.)

Figure 3 Operator mesh for CNs in Figure 2.

3. Keyword Search over Relational Streams

3.1 Problem Definition

In this section we shall introduce keyword search on relational streams. First, we start our discussion from keyword search on relational databases.

As a common basis, graph representation of relational database is used to define the semantics of keyword search [14]. In a data graph, each node represents a tuple and an edge represents a primary/foreign-key reference between two tuples. Now, let us assume a relational schema and a database that conforms to the schema. Given a set of user-specified query keywords, $\{k_1, k_2, \ldots, k_n\}$, keyword search on the database is to find all minimal total joining networks of tuples (MTJNT) [8], each of which contains all query keywords. More precisely, *total* means that all keywords are contained in each network, and *minimal* means that removing any tuple from a network of tuples leads to loss of eligibility for query results. Notice that the maximum size of data graphs is bounded by parameter T_{max}.

In contrast to conventional relational data, *relational streams* [3] can be modeled as possibly unbounded sequences

of relational tuples. In other words, each tuple in a stream can be represented by a pair of 1) a relational tuple and 2) a time instant of a discrete and ordered time domain, e.g., integer. Thus tuples are regarded that they are arrived according to their timestamps. Figure 1 (a) and (b) illustrate a sample schema and its instances.

When dealing with (relational) streams, we often use *sliding windows* to convert an infinite stream of tuples to a relation of finite tuples. There are two types of sliding windows, namely, time- and tuple-based sliding windows. Given a window size either in term of time interval (e.g., 1 hour) or number of tuples (e.g., 100 tuples), it generates relations by capturing the latest tuples within the window while sliding the window over the stream. In such window semantics, two tuples can be joined only if both tuples are valid.

Having defined relational streams and sliding windows, keyword search over relational streams can be defined as follows. Given a set of query keywords $\{k_1, k_2, \ldots, k_n\}$, a maximum network size T_{max}, and a window specification W, it continuously reports 1) new MTJNTs when new tuples are delivered and 2) invalidation of existing MTJNTs due to deletion or aging of tuples.

3.2 Existing Works

As mentioned in Section 1., S-KWS [9] and SS-KWS [13] are the predecessors of this work. In this section, we briefly overview these works.

3.3 Overview

In S-KWS and SS-KWS the process of keyword search on relational streams comprises two main steps: *preprocessing* and *evaluating* steps.

- In preprocessing step, given a schema, a set of query keywords, and T_{max}, all *Candidate Networks* (CNs) [9, 13] (both linear and tree) are generated. Each node in a CN represents a relation, and the edges represent relational *join* operation. Notice that all CNs must conform to the concept of MTJNT [9]. Figure 2 shows three examples of linear CNs from the schema in Figure 1(a).
- In evaluation step, keyword search over relational streams is actually processed by evaluating all CNs. When new MTJNTs are detected due to arrivals of new tuples, they are reported. On the other hand, expired tuples are removed by using either eager or lazy approaches [9].

3.3.1 S-KWS

S-KWS [9] is one of the pioneering works that addressed the problem of keyword search on relational streams using CNs. In their work, for each CN, the *root* node is determined such that 1) it is a leaf containing one chosen query keyword and 2) the keyword has a higher selectivity in the streams than other leaf nodes.

To improve performance, they propose to group all CNs that share the same root into a cluster. In fact, a CN can be converted into a corresponding operator tree, where the leaf nodes represent selection operators, in which their selection conditions may be null, and the internal nodes represent join operators. By combining multiple CNs in a cluster, they construct a data structure called *operator mesh*. Figure 3 shows two clusters of operator mesh created from the three CNs in Figure 2. It is important to notice that, for shared subtrees in the operator mesh, its process should be shared among different CNs, thereby reducing the cost of query processing.

When processing relational streams, all partial results are cached in each operator's buffer. Hence, in the case when some matches are detected in some CNs at the same time, all matched tuples can be efficiently catched from the operator's buffer. However, in fact, the idea of caching all partial results in buffers causes a performance bottleneck due to its high memory cost.

3.3.2 SS-KWS

SS-KWS [13] is a successor of S-KWS, and it can be regarded as the state-of-the-art approach. The novel idea of SS-KWS is to aggressively merge more sub-networks in CNs not only at single leaf, but also at all leaves. Unlike S-KWS, the root is the center node of the CNs. Besides, instead of operator mesh, a lattice is created by combining all CNs so that the query processing cost is reduced by sharing common subtrees except for the root nodes in CNs as much as possible. For example, the lattice structure for the three CNs in Figure 2 is shown in Figure 4. In this example, nodes marked with double lines are root nodes; black colored nodes are leaf nodes; and the rests are other non-leaf nodes.

SS-KWS proposes selection/semi-join approach to fully reduce all partial results. The concept of semi-join approach works as follows. First, each node's buffer is divided into three sub-buffers: N (not joinable), W (waiting), and R (ready). Figure 4 shows the lattice structure and all sub-buffers of node P. Probing sequence is clearly defined:

- In non-leaf nodes, for each incoming tuple, all tuples in sub-buffers W or R of their child nodes are checked. If it is not joinable with any tuple, the incoming tuple is stored in sub-buffer N of its corresponding node, and the probing is finished. Else, the incoming tuple is stored in sub-buffer W of its corresponding node, and the probing continues for the connected parent nodes. In leaf nodes, it immediately probes parent nodes.
- Similarly, in root nodes, for each incoming tuple, all tuples in sub-buffers W and R in the child nodes are checked, and the incoming tuple is stored in the respective sub-buffers as explained above. If all branches are joinable, the detected joining network of tuples (JNT) is emitted as a result, and all tuples in the JNT are moved to sub-buffers R of the re-

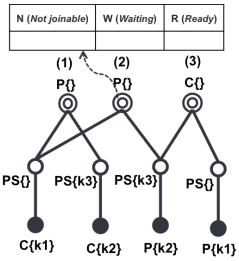
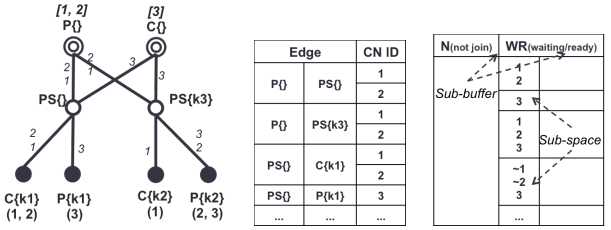


Figure 4 Lattice for CNs in Figure 2.



(a) MX-structure. (b) Edge-mapping table. (c) Sub-space in node PS{}.

Figure 5 MX-structure for CNs in Figure 2.

spective nodes.

3.4 Scalability Issues in Existing Approaches

We discuss the scalability issues of these approaches. As a common problem, the number of CNs grows exponentially as the number of keywords and/or T_{max} increase. This gives a significant impact on both time and space.

In S-KWS, partial results are maintained in the buffers in an operator mesh. Due to the low sharing rate of common subtrees in CNs; i.e., in an operator mesh, we can find a lot of common edges shared by different CNs but are not consolidated, because they are either in different clusters or do not have same root node. Consequently, in query processing, a lot of partial results need to be duplicated in buffers and need also to be processed independently.

In SS-KWS, the problem of the low sharing rate of common subtrees is mitigated by sharing common subtrees in all possible subtrees. However, there still exists a restriction that it is impossible to consolidate common paths in internal nodes, because 1) sharing is only allowed for common subtrees; and 2) root nodes are not allowed to be shared. Therefore, the number of common paths that are not consolidated in lattice grows rapidly as the number of CNs grows. For the same reason discussed above, such duplicated paths cause high memory consumption in internal buffers and also cause high computational cost for possibly useless processing of (duplicated) intermediate results.

4. Proposed Approach

4.1 Overview

In this section, we propose a novel approach to processing keyword search over relational streams. The proposed scheme also exploits CN-based approach, but tries to improve the performance by the following ideas: 1) given a set of CNs, we generate a maximal sharing structure called *MX-structure* by consolidating all shared edges in CNs regardless of their positions in CNs (i.e., root or leaf); and 2) we try to avoid duplications of tuples in buffers in MX-structure. More precisely, the former is achieved by maintaining information about the edges and their corresponding CNs using tables called *edge-mapping table*. The latter is achieved by introducing a new data organization in each node buffer. To operate with sliding windows, *lazy approach* [9] is used to deal with dead tuples.

4.2 MX-Structure

First, we introduce the proposed *MX-structure*. In each CN, the root (and the output node as well) is determined as the center of the CN. Then, all CNs are merged in such a way that all edges are unique; i.e., edges in MX-structure are created only for different combinations of nodes regardless of the node's position (root or leaf). Such information needs to be maintained as well. (In the sequel discussion, we denote by `()` a leaf node and by `[]` a root node.) We also maintain in the edge-mapping table the information of edge occurrences in different CNs.

Due to the space limitation, we cannot show the algorithm, but it can be constructed in the following way. Basically, all CNs are processed and added to an MX-structure one by one. When adding a new CN, we take each edge, and check its existence; we only add one only if it has not been added yet. Next, we insert the ID of the edge's CN to the edge-mapping table. The information about each CN's root and leaf nodes is also maintained.

Figure 5(a) illustrates an example of MX-structure for the CNs in Figure 2. Nodes marked with double lines show root nodes, and black nodes are leaf nodes. The label on each edge represents the set of corresponding CNs in term of IDs. Table 5(b) is its edge-mapping tables.

4.3 Query Evaluation in MX-Structure

4.3.1 Node Buffers

To enable efficient query evaluation using MX-structure, we propose a novel data organization in each node buffer. Our idea is based on the sub-buffer proposed in SS-KWS [13].

In an MX-structure, each node is associated with a buffer, which is further divided into two sub-buffers, N and WR. Sub-buffer N is for storing tuples that are *not joinable* with tuples in child nodes, while sub-buffer WR is for storing tuples that are *partially matched* (not part of the completed query results yet) or *fully matched* (as part of the completed query results) in some JNTs. Moreover, sub-buffer WR of each node is further divided into sub-spaces for each CN and for all possible combinations of all CNs containing that node.

We use $\sim n$ to denote fully matched to in CN n. If there is no $\sim$ (i.e., n), it means it is partially matched in CN n. Figure 5(c) shows the buffer of node $PS\{\}$ in the MX-structure. As shown in the figure, node $PS\{\}$ appears in CNs 1, 2, and 3. Therefore, sub-spaces for these CNs are created in sub-buffer WR. Sub-space $\{1, 2\}$ is for storing tuples that are partially matched for CNs 1 and 2. Sub-space $\{\sim 1, \sim 2, 3\}$ is for storing tuples that are fully matched for CNs 1 and 2, and partially matched for CN 3.

4. 3. 2 Probing Sequence

Probing sequence in MX-structure is explained as follows. Suppose a tuple arrives to a non-leaf node in an MX-structure. First, all tuples in sub-buffers N and WR of the child nodes are probed if the child nodes are at the leaf level; otherwise, only tuples in sub-buffers WR are probed. Second, if there exist some tuples that are joinable, all sub-buffers of the parent nodes are probed. If a new tuple arrives at one of the leaf nodes, the parent nodes are immediately probed. Unlike SS-KWS, probing is only performed to parent nodes that belong to set of *active* CNs, which are the CNs that have joinable tuples from their respective leaf nodes up to the node of the probing tuple (being used to probe tuples at parent nodes).

Active CNs are identified by 1) the IDs assigned to a connected edge (cn_{edge}) being traversed, 2) the IDs assigned to a leaf node (cn_{leaf}) if the child node is a leaf, and 3) the IDs of non-empty sub-spaces ($cn_{ecsubspace}$) in the child node, and is calculated by the following formula:

$$cn_{active} = cn_{edge} \cap (cn_{leaf} \cup cn_{ecsubspace}) \quad (1)$$

Having finished probing tuples in child/parent nodes, the probed tuple may be moved to appropriate sub-spaces according to matching status. In fact, in our approach, sub-spaces are dynamically created in an on-demand manner; i.e., if there exists an appropriate sub-space, we just use it; otherwise, we create a new sub-space according to the following formula:

$$cn_{npsubspace} = cn_{epsubspace} \cup cn_{active} \quad (2)$$

where $cn_{npsubspace}$ and $cn_{epsubspace}$ are respectively the new sub-space and the existing sub-space marked by IDs of CNs for the probed tuples at parent nodes.

Note that, if $cn_{epsubspace}$ is not empty, all tuples in $cn_{epsubspace}$ are already used to probe the parent nodes. In this case, probing parent nodes again can be performed efficiently, because only tuples in $cn_{epsubspace}$ need to be probed.

Let us take a look at Figure 6 as an example. First, when tuple $t5$, which belongs to node $PS\{\}$, arrives (Figure 6(a)), its child nodes (leaf nodes $C\{k1\}$ and $P\{k1\}$) are probed. Since node $P\{k1\}$ is empty, only node $C\{k1\}$ is probed. Since tuple $t1$ can be joined with $t5$, a new sub-space $cn_{epsubspace}$ is computed by Equations (1) and (2). Then, we get $cn_{edge} = \{1, 2\}$, $cn_{leaf} = \{1, 2\}$, and $cn_{ecsubspace} = \{\}$. As a result, we get $cn_{active} = \{1, 2\}$. So, we create new sub-spaces $cn_{epsubspace} = \{1, 2\}$ in sub-buffers WR of nodes $C\{k1\}$ and $PS\{\}$ (Figure 6(b)). Thus, $t1$ and $t5$ are moved to sub-space $\{1, 2\}$ of sub-buffer WR of nodes $C\{k1\}$ and $PS\{\}$, respectively. Since the coming tuple $t5$ is joinable with tuple in child node, it continues to probe parent nodes.

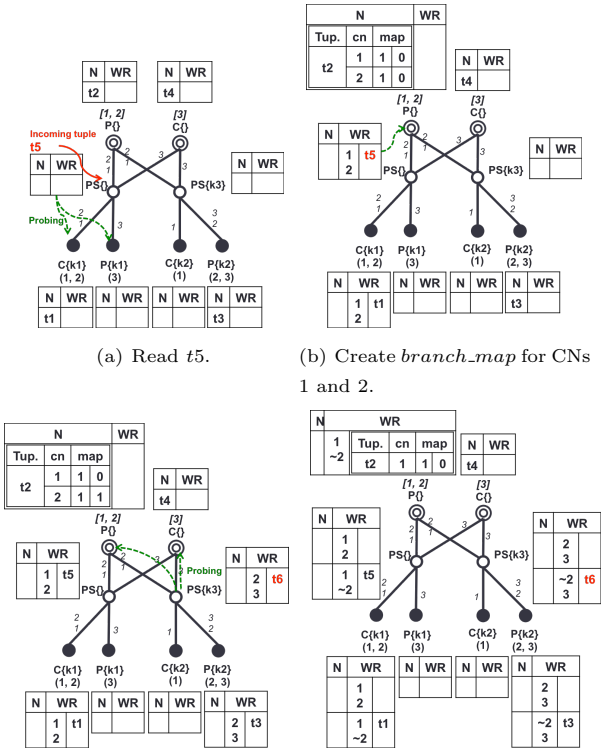
Before probing parent nodes, $P\{\}$ and $C\{\}$, the set of active CNs are computed by Equation (1) as follows. For edge $PS\{\} - P\{\}$, we get $cn_{edge} = \{1, 2\}$, $cn_{leaf} = \{\}$, and $cn_{ecsubspace} = \{1, 2\}$. So, $cn_{active} = \{1, 2\}$. Since $cn_{active} = \{1, 2\}$ is not empty, the parent node $P\{\}$ will be probed, while, for edge $PS\{\} - C\{\}$, we have $cn_{edge} = \{3\}$, $cn_{leaf} = \{\}$, and $cn_{ecsubspace} = \{1, 2\}$. Then, we get $cn_{active} = \{\}$. Because cn_{active} is empty, the parent node $C\{\}$ will not be probed. Since $t2$ can be joined with $t5$, and node $P\{\}$ is the root node for both CNs in cn_{active} ($= \{1, 2\}$), a couple of branch maps (*branch_map*) are created for both CNs, and the first bit is set to 1 in both maps. A branch map is to maintain the status of the CN whether the branches up to the root is connected or not (Figure 6(b)).

Same procedures are performed when $t6$ arrives (Figure Figure 6(c)), which results in the match of another branch in CN 2. Since all bits of CN 2 are set, CN 2 is detected as matched. Therefore, all matched tuples are returned as a query result. Then, they are moved to appropriate sub-spaces as shown in Figure 6(d) for subsequent processing.

4. 4 Algorithm Details

The proposed algorithm is shown in Algorithm 1. This algorithm works as follows. If the newly-arrived tuple, t_0, belongs to a leaf node, it probes the parent nodes (Line 5) by calling function `Probe_parent_nodes`. For each parent node, cn_{active} is calculated by Equation (1). If cn_{active} is not empty, it continues to check sub-buffers N and WR (Lines 2–3). For each sub-space in sub-buffer WR, $cn_{npsubspace}$ is calculated by Equation (2) (Line 4). If one or more tuples in the sub-space can be joined with t_0, we create a new sub-space $cn_{npsubspace}$ if necessary (Lines 5–6).

Then it checks if any CN in cn_{active} is fully matched by calling function `Match_CN` (Line 7). This function checks each CN in cn_{active} to update the *branch_map*. It is the map, attached to each tuple in the nodes with more than one child nodes for the same CN (Line 2). If the parent node is the root node of the CN, and if all bits in the *branch_map* for the CN are set to 1, the CN is detected to be fully matched (Line 4). Hence, all related fully-matched tuples are output as query results (Line 5). In addition, new sub-space



(a) Read $t5$.

(b) Create *branch_map* for CNs 1 and 2.

(c) Update *branch_map* for CN 2.

(d) Select matched tuples for CN 2.

Figure 6 Query processing using MX-structure. Let tuples $t1$, $t2$, $t3$, and $t4$ are currently maintained in Figure 6(a), and let all tuples joinable together, except $t4$.

for each selected tuples is created if necessary by copying its current sub-space and updating CN to ∼CN (Line 6) to indicate that the new sub-space is for storing the tuples that are fully matched to the CN. Then, it moves the matched tuples into respective new sub-spaces (Line 7). At the same time, $cn_{npsubspace}$ for the parent node needs also to be updated from CN to ∼CN (Line 9). After the call of function `Match_CN`, we go back to function `Probe_parent_nodes`, store t in sub-space $cn_{npsubspace}$, and t_1 is put in the set *sjtp* for subsequent probing to parent nodes.

Back to the main algorithm (Line 5), `Probe_parent_nodes` returns *sjtp*. If *sjtp* is not empty, it will call function `Probe_parent_nodes` again for probing subsequent parent nodes; otherwise, the loop is finished (Lines 7–12).

If the new tuple t_0 does not belong to leaf nodes (Line 14), it will call function `Probe_child_nodes` by following similar procedure above (Lines 14–31).

4. 5 Discussion

In this section we elaborate the reason why the proposed scheme is advantageous to the existing approaches. As we observed, the number of CNs exponentially increases as query keywords and/or the size of CNs grows. Consequently, even though S-KWS and SS-KWS try to merge the CNs by finding common sub-networks, the size of query plans rapidly grows, which leads to poor performance.

In MX-structure, we attempt to integrate different CNs more aggressively by consolidating edges according to the nodes at the both ends. Thus we can avoid the exponential blow up in query plans.

Readers may think that the complication is just migrated to the management of edge-mapping tables and complicated sub-buffers. This is partly true, but the point is that the number of CNs that are actually activated against real streams is far smaller than the possible combinations due to the locality in real data. Therefore, the cost for maintaining edge-mapping table and sub-buffers is expected to be modest. We confirm this in the following experiment.

Algorithm 1 Candidate Networks (CN) Evaluation

Input: Tuple t_0 just from streams, MX-structure MX

```
1: put t0 in set_joint_tuples
2: if t0 from leaf nodes then
3:     while 1 do
4:         while each t in set_joint_tuples do
5:             sjtp = Probe_parent_nodes (t, sjtp, MX)
6:         end while
7:         if sjtp is empty then
8:             break;
9:         else
10:            set_joint_tuples = sjtp
11:            clear sjtp
12:        end if
13:    end while
14: else
15:    while each t in set_joint_tuples do
16:        Probe_child_nodes (t, MX)
17:        if t joinable then
18:            while 1 do
19:                while each t in set_joint_tuples do
20:                    sjtp = Probe_parent_nodes (t, sjtp, MX)
21:                end while
22:                if sjtp is empty then
23:                    break;
24:                else
25:                    set_joint_tuples = sjtp
26:                    clear sjtp
27:                end if
28:            end while
29:        end if
30:    end while
31: end if
```

Function: `Probe_child_nodes` (t, *sjtp*, MX)

```
1: while Each child nodes do
2:     joinable = false
3:     if cn_active not empty then
4:         while Each sub-space in WR do
5:             Calculate cn_npsubspace
6:             if tuple t1 joinable then
7:                 joinable = true
8:                 Matched_CN (CN, cn_active, cn_npsubspace, branch_map,
       node of t, MX)
9:                 Put t in cn_npsubspace
10:                put t1 in sjtp
11:            end if
12:        end while
13:    end if
14:    if joinable == true then
15:        Update branch_map for CNs in cn_active
16:    end if
17: end while
18: Return sjtp
```

Function: `Probe_parent_nodes` (t, $sjtp$, MX)

```
while Each parent nodes do
    if cn_active not empty then
        while N and each sub-space in WR do
            Calculate cn_npsubspace
            if tuple t_1 joinable then
                Create cn_npsubspace if not exist
                Matched_CN (CN, cn_active, cn_npsubspace, branch_map, node of t, MX)
                Put t in cn_npsubspace
                put t_1 in sjtp
            end if
        end while
    end if
end while
Return sjtp
```

Function: `Matched_CN` (CN, cn_{active}, $cn_{npsubspace}$, $branch\_map$, **node,** MX)

```
while Each CN in cn_active do
    Update branch_map of that CN
    if Parent node is root node then
        if All bits in branch_map set to 1 then
            Select all matched tuples
            Copy sub-space of matched tuple, and update CN to ∼CN.
            Create that new sub-space if not exist, and move matched tuple in it.
            Return all matched tuples as result
            Update CN to ∼CN in cn_npsubspace
        end if
    end if
end while
Return (cn_npsubspace, branch_map, All buffers in MX)
```

Table 1 Parameters used in the experiments.

Parameter	Range and default
Window size (mn)	10, 20, **30**, 40, 50
Keyword frequency (%)	0.003, **0.007**, 0.01, 0.013
# of keywords	2, **3**, 4, 5
T_{max}	2, 3, **4**, 5

5. Experiments

In this section we report the experimental results to demonstrate the performance of the proposed scheme. As comparative methods, we implemented full mesh (FM) and partial mesh (PM) of S-KWS [9], SS-KWS [13], and the proposed approach in C++ language.

To generate relational streams, we read datasets from the disk, and fed them in the implement systems. All experiments were performed using 8-way Intel Core i7 CPU 870 (2.93 GHz) with 31.4 GiB memory running Ubuntu 13.10.

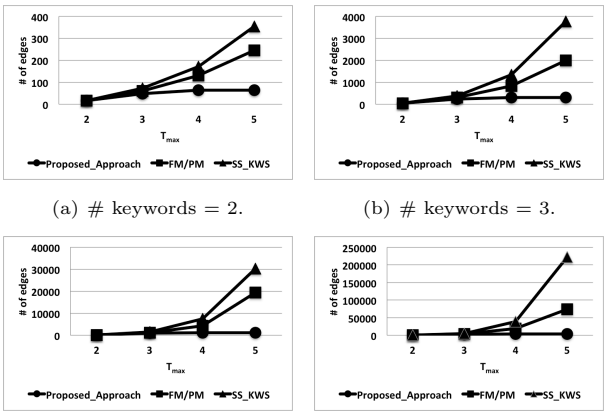
We used both synthetic and real datasets. Due to lack of space, only the result of synthetic dataset is presented. For synthetic dataset, we used TPC-H [1], which deals with ad-hoc decision support system in business environment. In this dataset, there are eight tables with 61 attributes.

Parameters used in the experiments are shown in Table 1. We varied these parameters and compared the performance of the proposed algorithm, S-KWS and SS-KWS. The default parameters are written in bold.

5.1 Edge Count

First, we compared the number of edges in the query plans of all approaches, which significantly affects the performance.

As can be seen in Figure 7, when the number of query keywords and T_{max} increased, the total number of edges in S-KWS and SS-KWS was exponentially increased, which was caused by the explosion of number of CNs whose edges could not be consolidated in their query plans. However, the growth rate of the proposed scheme was linearly increased because it consolidated unique edges into one, and the total number of unique edges, which were the primary/foreign-key relationships between two tables, in all CNs was slightly increased as the number of CNs increased.



(a) # keywords = 2. (b) # keywords = 3.

(c) # keywords = 4. (d) # keywords = 5.

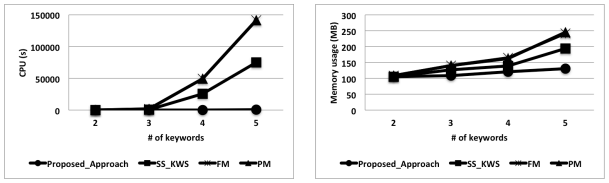
Figure 7 Comparison of number of edges.

5.2 Performance Comparison

We compared CPU running time and memory usage. Notice that this dataset is specially prepared to favor SS-KWS to S-KWS.

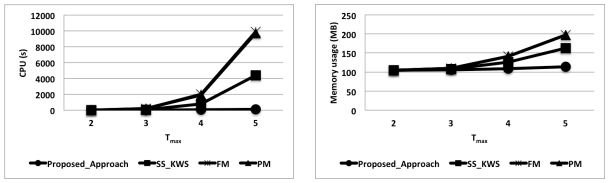
First, we measured the CPU running time and memory usage when varying the number of keywords (Figures 8(a) and 8(b), respectively). As can be seen, CPU running time and memory usage in FM/PM and SS-KWS were increased exponentially, whereas the proposed scheme was not. This is due to the reason as developed above that an explosive increase in number of edges in the query plan of S-KWS and SS-KWS causes an exponensial increase in number of probings in FM/PM and SS-KWS. Similar tendency can be observed when varying T_{max} from 2 to 5 (Figure 9).

Next, we increased the size of window from 10 min to 50 min. As expected, when the size of window was increased, the CPU running time and memory usage also increased as shown in Figure 10. This was because fewer tuples in the buffers of all approaches were expired and deleted as a result of the increase in size of window. Nevertheless, the total number of CNs did not increase when increasing window size, which caused little impact on the performance of all



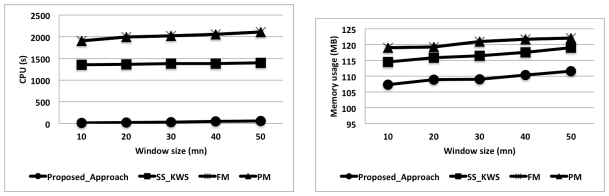
(a) CPU running times. (b) Memory usage.

Figure 8 Varying # of keywords.



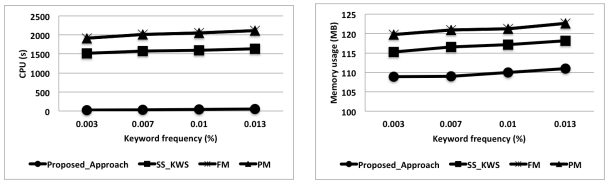
(a) CPU running times. (b) Memory usage.

Figure 9 Varying T_{max}.



(a) CPU running times. (b) Memory usage.

Figure 10 Varying window size.



(a) CPU running times. (b) Memory usage.

Figure 11 Varying keyword frequency.

approaches.

Figure 11 shows the impact on the performance of all approaches when varying keyword frequency in the dataset from 0.003%, 0.007%, 0.01% to 0.013%. When keyword frequency was increased, there were more tuples containing the keywords of the query. As a result, there were more tuples that need to be joint. Moreover, increasing keyword frequency did not cause any increase in total number of CNs. As a result, the increase in keyword frequency did not have much impact on the performance of all approaches.

6. Conclusion

In this paper, we had proposed a novel approach to efficiently process keyword search over relational streams. We had proposed a compact maximal sharing structure, MX-structure, which enables the sharing of any common edges among all CNs. We had also proposed a very efficient algorithm to process MX-structure over relational streams.

We had conducted several experiments by varying types of datasets of both real and synthetic datasets and experimental parameters. The experimental results prove that the proposed algorithm can handle the processing of keyword search over relational streams much better than all state-of-the-art approaches both CPU running times and memory usage. The proposed algorithm can process keyword search over relational streams in average about 40 to 70 times faster than the state-of-the-art approaches when the number of query keywords or T_{max} are increased up to five.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number 26280037.

References

[1] Tpc-h benchmark dataset. *http://www.tpc.org/tpch/*, 2015.

[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, U. Srivastava, and J. Widom. STREAM: The Stanford data stream management system. *Technical Report, Stanford InfoLab, http://ilpubs.stanford.edu:8090/641/*, 2004.

[4] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *Workshop, DBPL 2003*, Potsdam, Germany, 2003.

[5] M. Dyk, A. Najgebauer, and D. Pierzchala. Agent-based ms of smart sensors for knowledge acquisition inside the internet of things and sensor networks. *ACIIDS*, 9012:224–234, 2015.

[6] L. Edward. Cyber physical systems: Design challenges. *University of California, Berkeley Technical Report No. UCB/EECS-2008-8. Retrieved 2008-06-07*, 2008.

[7] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, 2003.

[8] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, Hong Kong, China, 2002.

[9] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIGMOD*, Beijing, China, 2007.

[10] K. Mehdi, A. Aijun, C. Nick, G. Parke, S. Jaroslaw, and Y. Xiaohui. Meaningful keyword search in relational databases with large and complex schema. In *ICDE*, Seoul, Korea, 2015.

[11] O. Niggermann and V. Lohweg. On the diagnosis of cyber-physical production systems. In *AAAI*, Texas, USA, 2015.

[12] O. Pericles, S. Altigran, and M. Edleno. Ranking candidate networks of relations to improve keyword search over relational databases. In *ICDE*, Seoul, Korea, 2015.

[13] L. Qin, J. Xu Yu, and L. Chang. Scalable keyword search on large data streams. In *VLDB Journal*, 2011.

[14] D. Shaul, E. Gadi, G. Shai, and P. Eran. DTL's DataSpot: Database exploration using plain language. In *VLDB*, San Francisco, CA, USA, 1998.

[15] H. Zhang, C. Sanin, and E. Szczerbicki. Experience-oriented enhancement of smartness for internet of things. *ACIIDS*, 9012:506–515, 2015.