Scalable Keyword Search over Relational Data Streams by Aggressive Candidate Network Consolidation☆

Savong Bou[a], Toshiyuki Amagasa[a], Hiroyuki Kitagawa[a]

[a]*Center for Computational Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577 Japan*

Abstract

Keyword search over relational streams is useful when allowing users to query on streams without understanding the details about the streams and query language as well. There have been several research works on this direction, and the state-of-the-art approaches exploit Candidate Networks (CNs), which are schema-level descriptions of possible joining networks of tuples, and generate query plans based on CNs. 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, we propose a novel query plan called *MX-structure* to consolidate CNs as much as possible. We suppress explosive blowup of nodes in query plans by consolidating all common edges among CNs. The experimental results prove that the proposed algorithm performs much better than the state-of-the-art approaches.

Keywords: Keyword search, relational streams, candidate network

1. Introduction

With the recent trends of Cyber Physical Systems [11, 22], Internet of Things [10, 27], etc., the number of real-time information sources has been explosively increasing. Besides, it has become common to extract information from various social media, such as Twitter and Facebook, in real-time for making analysis of diverse social activities. Such stream data sources can typically be modeled as *relational streams*, where structured records (relational tuples) are transmitted. Therefore, the importance of query processing over relational streams has been increasing.

When querying relational streams, *keyword search* is considered to be an attractive and practical approach due to several reasons. One of the major reasons is that users do not need to learn neither (potentially) complicated query language, like CQL [5], nor the schemas of streams being queried, which are also very complicated in many real applications. Instead, what they only need to do is to give several query keywords. So far, keyword search over permanently-stored relational data [23, 21, 26, 15, 4, 14] has been extensively studied, but only a few works have addressed keyword search over relational streams [19, 24].

In the works [19, 24], they employ candidate network-based approach for improving query performance. Specifically, for a given set of query keywords and a parameter that defines the maximum size of resulting networks of tuples (T_{max}), they first enumerate all candidate networks (CNs) that represent all possible combinations of keyword occurrences on join paths, and the generated CNs are merged to generate a query plan. Then, the actual streams are processed according to the query plan. More precisely, in S-KWS [19], a set of CNs are merged only if they share at least one leaf node called *root*, and possible sub-trees are merged to remove redundant processing as much as possible. In SS-KWS [24], 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

☆This work is an extend version of a conference paper "An Improved Method of Keyword Search over Relational Data Streams by Aggressive Candidate Network Consolidation. DEXA (1) 2016: 336-351".

dataset [2] 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 [24]. 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 5.2.) Thus the performance of S-KWS and SS-KWS would deteriorate in particular when dealing with a lot of query keywords and/or large-scale relational streams consisting of many relations. As reported in [13], the average query length to the search engines has been increasing. For example, the ratio of queries containing more than five words has increased by 10% every year, while that of single keyword queries has decreased by 3%.

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 can be associated to one of the primary/foreign-key relationships between two tables, whose number is in general small. In other words, CNs include intensive duplicates of edges representing the same primary/foreign-key relationships in the schema. In the above example, 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. Therefore, we can cope with the problem of CN's exponential blow up, if we consolidate the edges representing the same primary/foreign-key relationship into one edge when generating a query plan. It will lead 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 reveal that the proposed approach greatly outperforms comparative methods in both CPU running time and memory usage.

2. Problem Statement

In this section, we first introduce keyword search on relational databases. As a common basis, graph representation of a relational database is used to define the semantics of keyword search [25]. 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. Keyword search on the database is to find all minimal total joining networks of tuples (MTJNT) [15], which is defined in Definition 1.

Definition 1. *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) [15] that meet the following conditions:*

- Total*: All keywords are contained in each joining network of tuples.*
- Minimal*: Removing any tuple from a joining network leads to loss of eligibility for query results.*

Figure 1(c) shows an example of MTJNT. Note that the maximum size of joining networks is bounded by parameter T_{max}.

In contrast to conventional relational data, *relational streams* [5] can be modeled as possibly unbounded sequences of relational tuples that conform to relational schemas. In other words, each tuple in a stream can be represented by a pair of 1) a relational tuple and 2) a timestamp of a discrete and ordered time domain, e.g., integer. Thus tuples are regarded that they arrive according to their timestamps. Figures 1(a) and 1(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. In such window semantics, two tuples can be joined only if both tuples are in the sliding window.

Having defined relational streams and sliding windows, keyword search over relational streams is defined in Definition 2.

Definition 2. *Given a set of query keywords $\{k_1, k_2, \ldots, k_n\}$, a maximum network size T_{max}, and a window specification W, keyword search over relational streams continuously:*

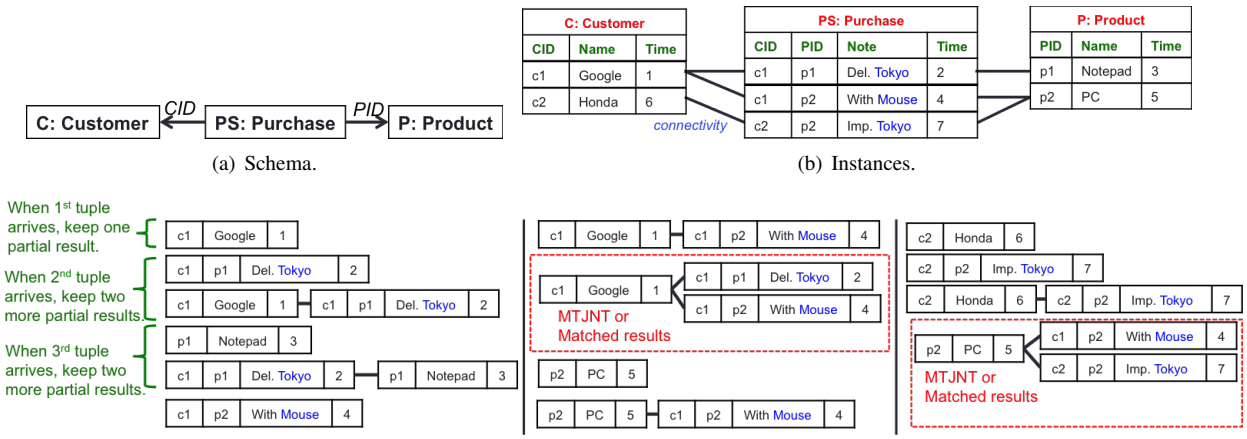
- *Reports new MTJNTs when new tuples are delivered.*
- *Invalidates the affected MTJNTs due to aging of tuples.*

3. Existing Works

S-KWS [19] and SS-KWS [24] are the predecessors of this work. In this section, we briefly overview these works.

3.1. Overview

In S-KWS and SS-KWS, the process of keyword search on relational streams comprises two main steps: *preprocessing* and *filtering* steps as shown in Figure 3.



(a) Schema. (b) Instances.

(c) All partial results that need to be kept for future evaluating as streams of tuples arrive.

Figure 1: An example of keyword search "Tokyo, Mouse" on relational streams in Figure 1(b). Notice that T_{max} is set to three. The joining networks of tuples (JNTs) in red box are MTJNTs because they contain all query's keywords, while the rests are not MTJNTs as shown in Figure 1(c).

Preprocessing step Given a schema, a set of query keywords, and T_{max}, all *Candidate Networks* (CNs) [19, 24] are generated. Candidate Network (CN) is defined in Definition 3.

Definition 3. *Given a schema, a set of query keywords, and T_{max}, a* Candidate Network (CN) *is a tree, where:*

- *Each node represents a relation.*
- *Each edge represents a relational* join *operation.*
- *All CNs must conform to the concept of MTJNT [15].*

Figure 2 shows all CNs that are generated from schema in Figure 1(a) for relational keyword search "*k1*, *k2*". In each CN, each node represents a tuple set, and line linked between two nodes represents their relationship. Only one single node (tuple set) is also a CN as long as it contains all keywords of the given query. For example, in CN 1, node (tuple set) "*C*{*k1k2*}" refers to all tuples from table "*Customer*" that contain both keywords "*k1*" and "*k2*". In CN 5, node "*PS*{}" is referred to all tuples from table "*Purchase*" that does not contain any keyword in relational keyword search "*k1*, *k2*". Then a query plan is created by combining all CNs.

Filtering step In this step, the query plan is evaluated over relational streams. 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 [19].

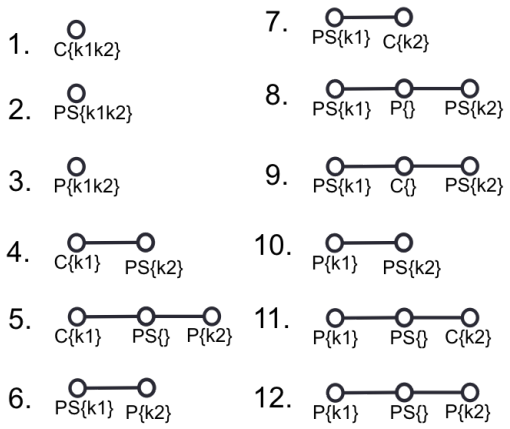


Figure 2: All CNs created from schema in Figure 1(a) for query "k_1, k_2". Notice that the label under each node is a tuple set, and "*C*" is referred to table "*Customer*", "*PS*" is referred to table "*Purchase*", and "*P*" is referred to table "*Product*". The keyword inside the curly bracket is referred to the keyword of the given query that each node contains. Notice that, for this example, T_{max} is set to 3.

3.2. S-KWS

S-KWS [19] is one of the pioneering works for this search framework. In this work, for each CN, the *root* node is defined as the node containing one chosen query

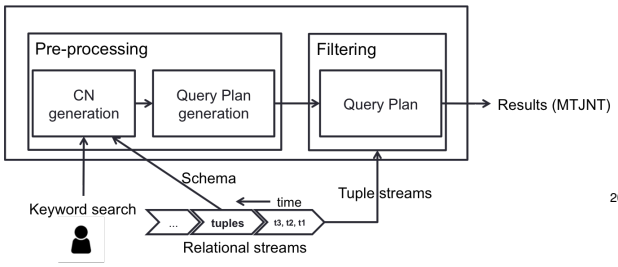


Figure 3: General framework.

keyword. Then, left-deep operator tree is created for each CN.

To improve performance, they propose a query plan, called an operator mesh, by grouping all left-deep operator trees that share the same root into a cluster so that all common join operators can be consolidated, resulting in improved performance by sharing common operations on the same data. For example, suppose we have keyword search, $\{k_1, k_2\}$, over relational streams whose schema is shown in Figure 1(a), and T_{max} is set to three, all CNs are shown in Figure 2. Suppose that the root node is chosen as a node that contains query keyword k_1. Then, an operator mesh is created by combining all CNs into different clusters as shown in Figure 4.

When processing relational streams, all partial results are cached in each operator's buffer for efficient retrieval of matched results. However, caching all partial results is the main performance bottleneck due to its high memory cost.

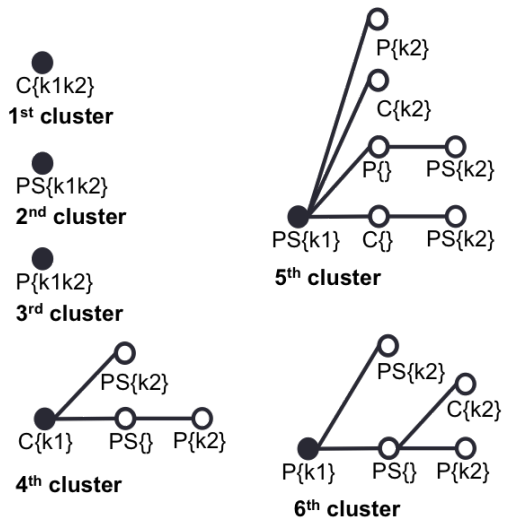


Figure 4: Operator mesh that has several clusters created from all CNs in Figure 2. Notice that black-filtered circles are root nodes.

3.3. SS-KWS

SS-KWS [24] is a successor of S-KWS and 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 focusing at a single leaf, but also at all leaves. Unlike S-KWS, the root is the center node (the node with the shortest paths to all leaf nodes) of the CN. Besides, instead of the operator mesh, a query plan, called a lattice, is created. It combines all CNs by sharing common sub-trees except for the root nodes in CNs as much as possible. If T_{max} is at least four, some processing is shared among CNs in the lattice. However, if T_{max} is set to smaller than four, no CNs can share processing. In the same above example where T_{max} is set to three, the lattice for all CNs in Figure 2 is shown in Figure 5. Nodes marked with double lines are root nodes; black colored nodes are leaf nodes; and the rests are other non-leaf nodes. For node $C\{k_1k_2\}$ acts as both root and leaf node. As can bee seen, even though all CNs are consolidated into a lattice, no sub-trees can be shared, which means no process can be shared (e.g., common edge $P\{k_1\} - PS\{\}$ in CNs 11 and 12, and edge $P\{k_2\} - PS\{\}$ in CNs 5 and 12 are not consolidated).

To reduce partial results, SS-KWS proposes selection/semi-join approach by dividing the buffer of each node into three sub-buffers: N (not joinable), W (waiting), and R (ready). It adopts a bottom-up probing sequence. If the tuple is joinable with other tuples, it is stored in sub-buffer W; otherwise, in N. If MTJNT of any CN is detected, all related tuples are moved to sub-buffer R. Thus SS-KWS successfully reduces memory usage compared to S-KWS.

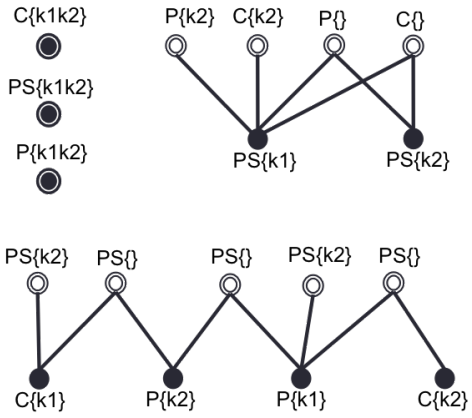


Figure 5: Lattice for all CNs in Figure 2

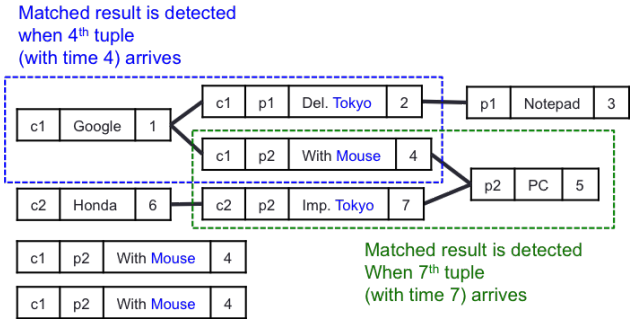


Figure 6: An example of keyword search "Tokyo, Mouse" on relational streams in Figure 1(b) by using SS-KWS approach.

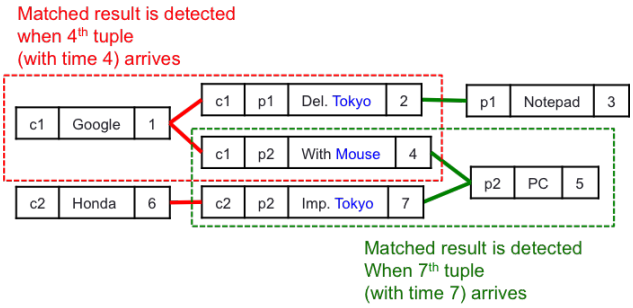


Figure 7: Example of processing keyword search "Tokyo, Mouse" on relational streams in Figure 1(b) by using the proposed approach. Red label represents matchability to CN 9, and blue label to CN 8.

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 insufficient sharing of the common sub-trees in CNs; e.g., in the operator mesh, a lot of edges connecting the same tables are not consolidated, because they are either in different clusters or do not have the same root node. Consequently, in query processing, a lot of partial results have to be duplicated in buffers and need to be processed independently.

The problem of the insufficient low sharing of the common sub-trees also occurs in SS-KWS because it is impossible to consolidate internal common paths because 1) sharing is only allowed for common sub-trees, and 2) root nodes are not allowed to be shared. Therefore, the number of unconsolidated paths grows rapidly as the number of CNs grows. For the same reason as discussed above, such duplicated paths cause high memory consumption in the internal buffers and also cause high computational cost for duplicated intermediate results.

For example, by using S-KWS, to process keyword search "Tokyo, Mouse" on relational streams in Figure 1(b), all partial results shown in Figure 1(c) need to be kept for future incoming tuples. As can be seen, the partial results are quite a lot compared with the incoming stream tuples. Similarly, if SS-KWS is used, the partial results that need to be kept are smaller as shown in Figure 6. Though, the tuple with timestamp four is duplicated three times for different matching evaluation purposes. The total number of such evaluations increases if the number of keywords increases and T_{max} becomes longer.

4. Proposed Approach

4.1. Main Ideas

For the sake of easy detection of matched results, the existing approaches either keep all partial results separately or duplicate tuples multiple times, which leads to very poor performance. We observe that if all partial results are consolidated together without any duplicate tuples, performance will be less impacted by the length of queries because unnecessary duplicate evaluations can be largely reduced . The proposed approach is based on this idea. The proposed approach keep all partial results in a labeled graph with each label representing matchability to each CN. Moreover, matched results are tracked by the labels.

For example, in the proposed approach, to process keyword search "Tokyo, Mouse" on relational streams in Figure 1(b), all partial results are kept together as shown in Figure 7. As can be seen, no tuples are duplicated, and each tuple is evaluated only one time. Moreover, matched results can be easily detected through labels of connected tuples. For example, the network of three connected tuples marked by the red label (marked in the red-broken-line box) is detected as a matched result because it contains all keywords and conforms to CN 9.

4.2. Overview

Our proposed framework for keyword search over relational streams has the same general processing framework as the existing approaches [19, 24] that involve two main steps as explained in Section 3. The first step is to create all CNs from the given keyword search and

schema of relational streams. Then, a query plan is created by combining all CNs together for efficient processing. In the second step, the query plan is directly evaluated against the incoming relational streams to find the query results. Our proposal is to create a better query plan. For this purpose, we propose a novel query plan representation, called *MX-structure* (maximal-sharing structure), that combines all CNs by consolidating all common edges. By MX-structure, we can avoid redundant nodes and edges to be expanded. To enable the processing of MX-structure over relational streams, we introduce fine-grained node buffers and branch maps for managing existing partial/full query results. To deal with expiration of tuples, we adopt *lazy approach* [19] where expired tuples are removed when node buffers are probed.

4.3. MX-Structure

First, we introduce the proposed MX-structure by starting from its construction as follow. First, each CN is marked by one unique ID, which is used to detect its matched MTJNTs. In each CN, the root node (and the output node as well) is determined as the centered node (the node with shortest paths to all leaf nodes). Then, all CNs are merged in such a way that all edges are unique; e.g., edges in MX-structure are created only for different combinations of nodes regardless of their positions (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. Notice that, in MX-structure, each source node and each edge represent selection operation and join operation between two connected nodes, respectively. MX-structure is defined in Definition 4.

Definition 4. *Given a set of CNs, MX-structure is a labeled graph that is generated by combining all CNs, and consists of:*

- *Nodes: Represent relations.*
  - *Root nodes: Represented by* `[]`*. The numbers inside* `[]` *represent IDs of the CNs which include this node as the roots.*
  - *Leaf nodes: Represented by* `()`*. The numbers inside* `()` *represent IDs of the CNs which include this node as leaf nodes.*
  - *Normal nodes: Represented by nodes that do not have both* `[]` *and* `()` *marks.*
- *Edges: Represent relational join operations.*
- *Labels of Edges: Represent the sets of CNs that contain the edges.*

Algorithm 1 MX-structure Construction

Input: *CNs*

```
1: Initialize MX-structure MX
2: for each CN do
3:     for each edge do
4:         if edge not exists in MX then
5:             add that edge into MX
6:         end if
7:         add id of that CN of that edge in MX;
8:         if each node is either root or leaf node then
9:             add id of that CN into MX to mark leaf or root
   node of CN.
10:        end if
11:    end for
12: end for
```

The pseudo code to construct MX-structure is shown in Algorithm 1. Basically, all CNs are added to an MX-structure one by one. When adding a new CN, we take each node/edge and check its existence; we add one only if it has not been added yet. Next, ID of the CN is added to each of its nodes/edges in MX-structure. The information about each CN's root and leaf nodes is also maintained.

Figure 8 illustrates an example of MX-structure for all CNs in Figure 2 (Notice that all CNs in Figure 2 are generated for keyword search "k_1, k_2" on relational streams whose schema is shown in Figure 1(a)). Nodes marked with double lines show root nodes, and black nodes are leaf nodes. The label on each edge represents the set of IDs of the corresponding CNs. The numbers in `()` and `[]` are the IDs of CNs corresponding to the leaf and root nodes, respectively.

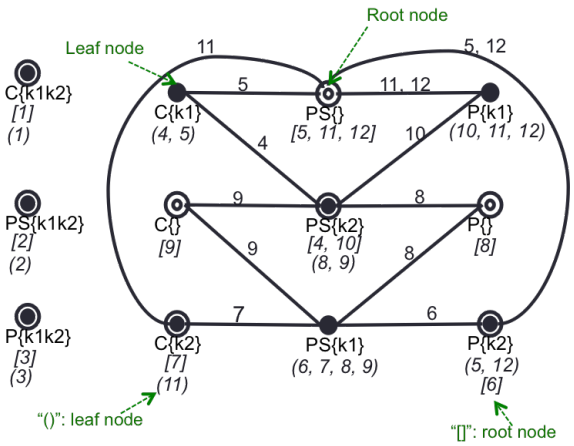


Figure 8: MX-structure for all CNs in Figure 2

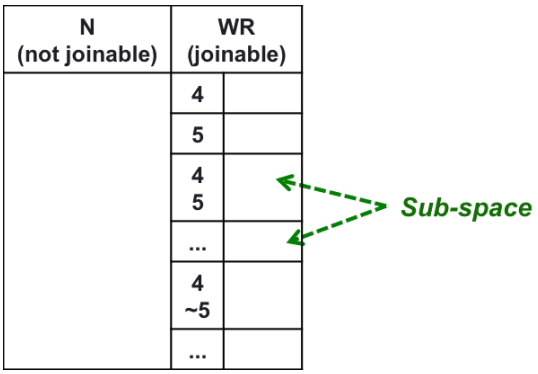


Figure 9: Node buffer of node $C\{k1\}$ of MX-structure in Figure 8

4.4. Query Evaluation in MX-Structure

To evaluate queries over relational streams using MX-structure, we need to track the matching status of each tuple to the respective CNs. For example, let us look at Figure 8. If all joins between all edges ($P\{k1\}$-$PS\{\}$-$P\{k2\}$) of CN 12 are detected, tuples that contribute to form MTJNT of CN 12 need to be output as a query result. This is allowed by the fine-grained status management of (existing) tuples using *node buffers*. More precisely, for each incoming tuple, its join-ability is checked according to the *probing sequence*, and is stored in an appropriate sub-space in a sub-buffer w.r.t. the corresponding CN, which is allocated dynamically when necessary. Thus the proposed scheme achieves better performance while consuming less memory space.

4.4.1. Node Buffers

Several concepts are important to understand node buffers of MX-structure. Statuses "*not joinable*" and "*joinable*", "*partially matched*", and "*fully matched*" of a tuple are defined in Definitions 5, 6, and 7, respectively.

Definition 5. *The status of a tuple in a node is marked as "*not joinable*" if it cannot be joined with other tuples in the child nodes of any CN. Otherwise, its status is marked as "*joinable*" w.r.t. one or more CNs.*

Definition 6. *A tuple with "*joinable*" status is consider as "*partially matched*" to one or more CNs if it is not in the MTJNTs of those respective CNs.*

Definition 7. *A tuple with "*joinable*" status is consider as "*fully matched*" to one or more CNs if it is in the MTJNTs of those respective CNs.*

For example, in Figure 7, tuples with timestamp values 1, 2 and 4 have the "*fully matched*" status to the CN of the MTJNT, which is marked by red broken-line box. Similarly, tuples with timestamp values 4, 7 and 5 have the "*fully matched*" status to CN of the MTJNT, which is marked by green broken-line box. However, tuples with timestamp values 6 and 7 have the "*partially matched*" status to the CN, which is marked by red edge. Tuples with timestamp values 2 and 3 have the "*partially matched*" status to the CN, which is marked by green edge.

In MX-structure, each node buffer is divided into two sub-buffers, N and WR. Sub-buffer N is for storing tuples that are *not joinable*, while WR is for storing tuples that are *joinable* with other tuples. Moreover, sub-buffer WR is divided into sub-spaces according to the CNs it belongs to. Each sub-space records joinable tuples to its matched CNs. In the following discussion, we denote $\sim n$ as the sub-space for tuples that are *fully matched* (as part of the complete query results) w.r.t. CN n, whereas n as the sub-space for tuples that are *partially matched* (not part of the complete query results) w.r.t. CN n. The table in Figure 9 shows the buffer of node $C\{k1\}$ of MX-structure in Figure 8. As can be seen, node $C\{k1\}$ appears in CNs 4 and 5. For this reason, some sub-spaces are created in sub-buffer WR; e.g., $\{4, \sim 5\}$ means that tuples partially match CN 4 and fully match CN 5. Note that we dynamically create sub-spaces when necessary to avoid the allocation of unnecessary (unpopulated) sub-spaces.

4.4.2. Probing Sequence

To systematically evaluate queries, for each incoming tuple, we check its joinability with other existing tuples in the node buffers in other child and/or parent nodes, and such probes are performed in the leaf-to-root direction; if a new tuple arrives at a leaf node, then we probe its parent nodes; otherwise, we first probe the child nodes, then probe the parent nodes. More precisely, when probing child nodes, we probe existing tuples in both sub-buffers N and WR if the nodes being probed are at the leaf level, but do so only for WR if the nodes are at non-leaf levels. If it turns out that the incoming tuple is not joinable with any other tuples in the node buffers in the child nodes, then current probing is finished, and the tuple is stored in sub-buffer N (not joinable); otherwise, it is stored in a sub-space in sub-buffer WR that corresponds to the CN(s) to which the incoming tuple contributes to form the resulting MTJNT(s). Then, only parent nodes w.r.t. those corresponding matched CNs will be subsequently probed.

If the incoming tuple contributes to form MTJNT(s) of CN(s), the CNs become *active*. The set of active CNs is defined as follows:

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

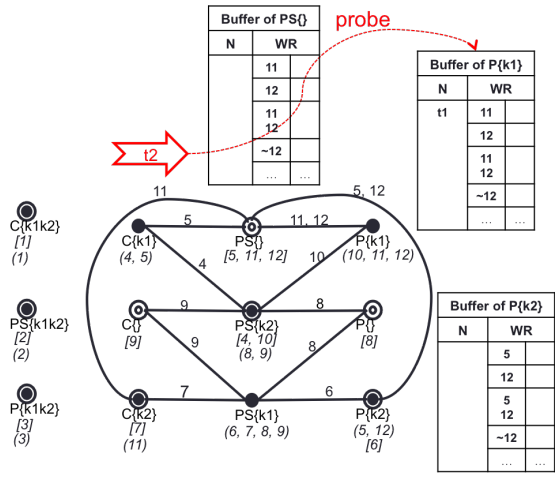
where cn_{edge} is the set of IDs of CNs assigned to the connected edge(s) being traversed, cn_{leaf} is the set of IDs of CNs assigned to the leaf node(s) if the probed child node(s) is a leaf node, and $cn_{ecsubspace}$ is the set of IDs of CNs of non-empty sub-spaces in the child node. Note that, if the probed child node is a non-leaf node, cn_{leaf} is empty. Similarly, in sub-buffer N, $cn_{ecsubspace}$ is also empty. Determining active CNs is beneficial to avoid unnecessary probings due to the fact that inactive CNs in child nodes can never be active in parent nodes. Thus, once active CNs are determined by probing child nodes, only the parent nodes that are connected via edges of active CNs are probed, thereby avoiding unnecessary probings at the upper levels.

Let us look at Figure 10(a) as an example. Notice that only node buffers that store tuples are shown for simplicity. Let us assume that tuples *t1* and *t2*, which are 1) of tables *P* and *PS*, resp., 2) *t1* contains keywords *k1* and *t2* does not contain any query keyword, and 3) they are joinable with each other and arrive in this order. When *t1* arrives, we immediately probe the parent nodes *PS*{} and *PS*{*k2*}, because *P*{*k1*} is at the leaf level. As a result, it turns out that *t1* is not joinable because of empty node buffers in *PS*{*k2*} and *PS*{}, and is stored in the sub-buffer N in *P*{*k1*}. Afterwards, when *t2* arrives, we probe the child nodes, *C*{*k1*}, *P*{*k1*}, *C*{*k2*}, and *P*{*k2*}. Since buffers of nodes *C*{*k1*}, *C*{*k2*}, and *P*{*k2*} are empty, probing is ended. When probing *P*{*k1*}, *t2* turns out to be joinable with *t1* w.r.t. CNs 11 and 12.

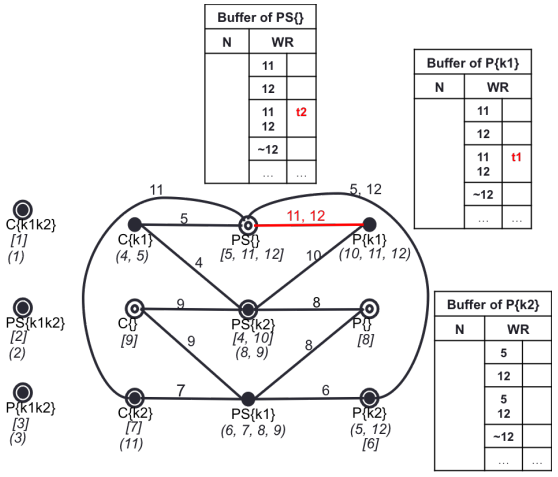
By applying the formula explained above[1], we get $cn_{active} = \{11, 12\}$, so 1) *t1* is moved to the sub-space {*11, 12*} in sub-buffer WR, and 2) *t2* is stored in sub-space {*11, 12*} in sub-buffer WR in the respective nodes as shown in Figure 10(b). For subsequent probings of parent nodes, only active CNs (CNs 11 and 12) are taken into consideration. In this case, *PS*{} has no parent nodes, so probing is finished.

4.4.3. Branch Map

In MX-structure, in many cases, root/output nodes are internal (non-leaf) nodes in one or more CNs. In addition, since probing proceeds in the leaf-to-root direction, we need to maintain for each tuple in the root node its matching status so that we can output new MTJNTs as soon as they are detected. To this end, we use a map called *branch map* to track whether there are matched tuples in all nodes from all leaf nodes up to the root/output node of any CN. More precisely, a branch map is attached to each joinable tuple in the root/output nodes. A branch map has several bits corresponding to the branches from the leaf (or leaves). When all bits are set to one, the MTJNT that contains the root tuple is output as a result. For example, node *PS*{} in MX-structure



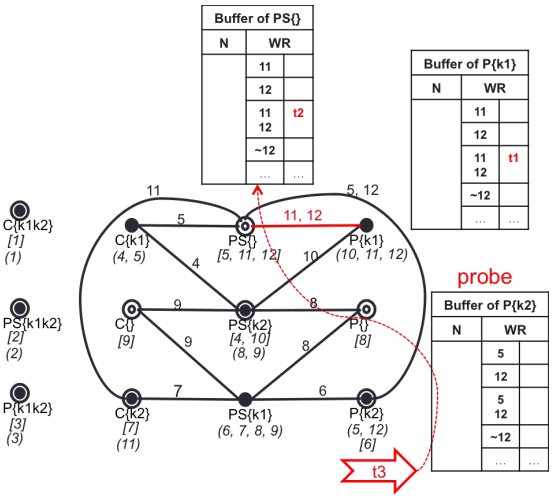
(a) When t1 arrives, it is stored in $P\{k1\}$. When t2 arrives, it probes $P\{k1\}$.



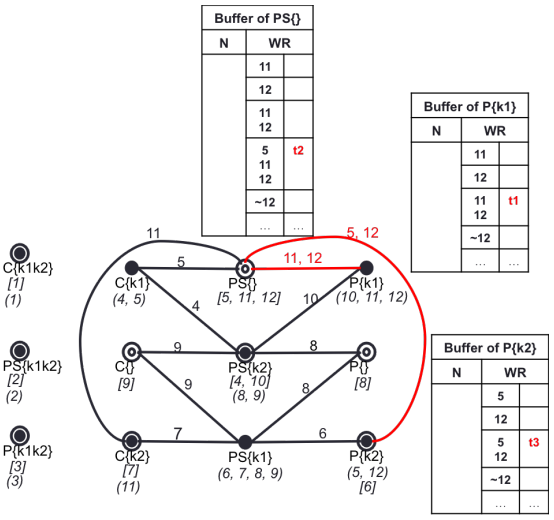
(b) *t2* can be joint with *t1*, so move them to subspace {11, 12} of their respective nodes

Figure 10: Example of probing sequence.

[1]We have $cn_{edge} = \{11, 12\}$ (edge *PS*{}-*P*{*k1*} belongs to CNs 11 and 12), $cn_{leaf} = \{10, 11, 12\}$ (node *P*{*k1*} is a leaf node of CNs 10, 11, and 12), and $cn_{ecsubspace} = \{\}$ (*t1* is currently in sub-buffer N). As a result, we get $cn_{active} = \{11, 12\}$.



(a) When *t3* arrives, it probe *PS*{}



(b) *t3* can be joint with *t2*, so move *t3* and *t2* to subspace {5, 12} and {5, 11, 12} of their respective node

Figure 11: Example of probing sequence

in Figure 8 is the root node of CNs 5, 11, and 12, and it has two branches for each CN. Figure 13 shows the branch maps of this node (*PS*{}). Since each CN has two leaf nodes, each map has two bits which are initialized by zero.

Continued from the example in Figure 10(b). Since tuples t1 and t2 of edge *PS*{}-*P*{*k1*} that belongs to CNs 11 and 12 are joinable, the first bits of CNs 11 and 12 corresponding to edge *PS*{}-*P*{*k1*} are set to one.

Suppose tuple *t3* in *P*{*k2*} has arrived (Figure 11(a)), and is joinable with *t2* w.r.t. active CNs 5 and 12[2]. Then,

[2]We have $cn_{edge} = \{5, 12\}$ (edge *P*{*k2*}-*PS*{} belongs to CNs 5 and 12), $cn_{leaf} = \{5, 12\}$ (node *P*{*k2*} is a leaf node of CNs 5 and 12), and $cn_{ecsubspace}$ is empty (*t3* has just arrived). As a result, we get $cn_{active} = \{5, 12\}$.

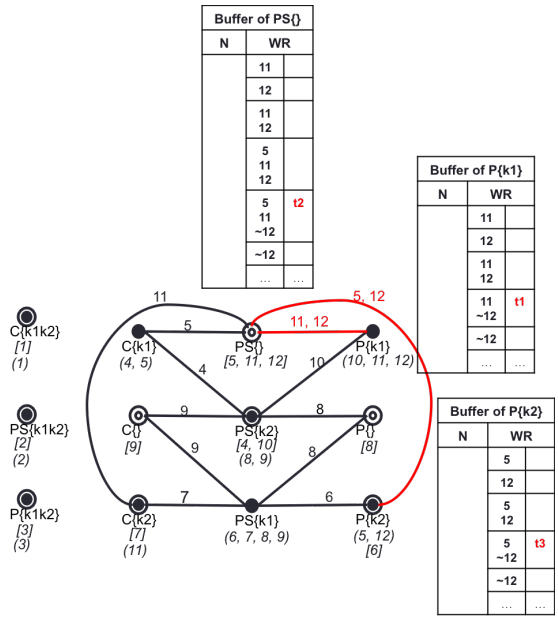


Figure 12: CN 12 is matched, so its MTJTNs is returned as query result. All related matched tuples are moved to the appropriate subspace.

t3 is kept in subspace {5, 12} of node *P*{*k2*} as explained earlier (Figure 11(b)). Moreover, *t2* is now joinable to CNs 5, 11, and 12, so it is moved to subspace {5, 11, 12} of node *PS*{}. Since node *PS*{} is the root node, the second bits corresponding to edge *P*{*k2*}-*PS*{} in the existing branch map for active CNs (CNs 5 and 12) are set to one. Since all bits of CN 12 are set to one, CN 12 is detected as matched, and its matched MTJNT is returned as a query result. Then, all matched tuples are moved to the appropriate sub-spaces of their fully matched CN 12 as shown in Figure 12 for subsequent processing.

ID of CN	map	
5	0	0
11	0	0
12	0	0

Figure 13: Branch maps for node *PS*{} of MX-structure in Figure 8

4.4.4. Dynamic Generation of Sub-spaces

As explained earlier, we dynamically populate sub-spaces when necessary because 1) generating all possible sub-spaces requires huge memory spaces, and 2) only a few sub-spaces are used in query processing. To this end, we populate a new sub-space according to the following formula:

$$cn_{newsubspace} = cn_{oldsubspace} \cup cn_{active} \quad (2)$$

where $cn_{newsubspace}$ and $cn_{oldsubspace}$ are respectively the new sub-space and the existing sub-space marked by IDs of CNs for each joinable tuple. Notice that, if tuple just arrives or is currently in sub-buffer N, its $cn_{oldsubspace}$ is empty.

4.4.5. Dynamic Maintenance of MX-Structure

Purging the expired tuples is important in query processing with MX-structure. We employ a lazy approach as proposed in [19]; i.e., expired tuples are deleted during probing node buffers when their timestamps are known to be expired. The detailed procedure is as follows:

1. Delete the expired tuples from the node buffers.
2. For each expired tuple, we determine the CN(s) where the expired tuple belongs, and, for each CN, we update the statuses of all tuples that are joinable to the expired tuple as follows:
   (a) If there are joinable tuples in the parent node in the CN, we move the tuples that are joinable with the expired tuple to sub-buffer N (not joinable). Also, if they contain branch maps, we modify the bit corresponding to the affected branch from 1 to 0.
   (b) If there are joinable tuples in the child node(s) in the CN, we move the tuples that are joinable with the expired tuple to sub-space of "partially matched" if their status is "fully matched".

4.5. Algorithm Details

The proposed algorithm is shown in Algorithm 2. This algorithm works as follows. If the incoming tuple, t_0, belongs to a non-leaf node, it probes child nodes by calling function `Probe_child_nodes` (Line 3). This function returns *joinable_to_child* = true if there are joinable tuples in child nodes with the incoming tuple. Otherwise, it returns *joinable_to_child* = false, which results in finishing the current probing, and t_0 is stored in sub-buffer N (Line 4).

This function `Probe_child_nodes` works as follow. For each sub-space of sub-buffer WR in each child node (and sub-buffer N if child node is leaf node), cn_{active} is computed by Equation (1). If cn_{active} is not empty, it checks each tuples in that sub-space (Line 2–5). If there are tuples joinable with the incoming tuple, *joinable_to_child* is set to true (Line 6), and function *Match_CN* is called to check if any partially matched CNs in cn_{active} are fully matched (Line 7). This function returns *joinable_to_child* (Line 12).

In function *Match_CN*, each CN in cn_{active} is checked if there are fully matched CNs. First, appropriate sub-space, $cn_{newsubspace}$, is computed by Equation (2) (Line 1). Then, for each partially matched CN, *branch_map* is updated (Line 2). There are fully matched CNs if the parent node is the root node and all bits in the *branch_map* are set (Line 3–4). If any fully matched CN is found, its MTJNT is returned as a result, and sub-space, $cn_{newsubspace}$, are updated according to the fully matched CN (Line 5–6). Finally, all matched tuples are moved to the appropriate sub-space $cn_{newsubspace}$ (Line 8).

Back to the main algorithm, if the incoming tuple is from leaf nodes or *joinable_to_child* is true, subsequent parent nodes are probed until no parent nodes have joinable tuples (Line 8–18) by calling function `Probe_parent_nodes` (Line 10) following similar procedure above.

4.6. Discussion

In this section we elaborate the computational complexity of all algorithms, by investigating their cost in terms of the number of probings. Assume that the total number of relations in the schema of relational streams is S, the average number of tuples within the window of each relation is n, the number of query keywords is m, and the maximum size of CN is T_{max}.

In mesh of S-KWS, the root nodes must be keyword nodes which contain one particular chosen keyword, so the number of possible root nodes is $O(S * 2^{m-1})$. Therefore, the maximum number of nodes in mesh is $O((S * 2^{m-1})^{T_{max}})$. Then, the maximum number of probings between tuples in mesh is $O((S * 2^{m-1})^{T_{max}} * n)$.

Similarly, in lattice of SS-KWS, root nodes can be both keyword and non-keyword node, so the maximum number of root nodes are $O(S * 2^m)$. Therefore, the maximum number of nodes in lattice is $O((S * 2^m)^{T_{max}})$. Then, the maximum number of probings in mesh is $O((S * 2^m)^{T_{max}} * n)$.

For the proposed MX-structure, the total number of nodes is $O(S * 2^m)$ because all nodes are unique.

Algorithm 2 MX-structure Evaluation

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

```
joinable_to_child = false
if t0 from non-leaf nodes then
    joinable_to_child = Probe_child_nodes (t0, MX)
    Put t0 in sub-buffer N if joinable_to_child = false
end if
if t0 from leaf nodes or joinable_to_child = true then
    put t0 in set_joint_tuples
    while 1 do
        for each t in set_joint_tuples do
            sjtp = Probe_parent_nodes (t, sjtp, MX)
        end for
        if sjtp is empty then
            break;
        else
            set_joint_tuples = sjtp
            clear sjtp
        end if
    end while
end if
```

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

```
joinable_to_child = false
for Each child nodes do
    for Each sub-space, sp, in WR (and N if child node is leaf node) do
        if cn_active not empty then
            for Each tuple t1 in sp joinable with t do
                joinable_to_child = true
                Matched_CN (cn_active, MX)
            end for
        end if
    end for
end for
Return joinable_to_child
```

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

```
for Each parent node, pn do
    if cn_active not empty then
        for Each tuple t1 in pn joinable with t do
            Matched_CN (cn_active, MX)
            put t1 in sjtp
        end for
    end if
end for
Return sjtp
```

Function: `Matched_CN` (cn_{active}, *MX*)

```
Compute cn_newsubspace
for Update branch_map of each CN in cn_active do
    if All bits in branch_map set to 1 then
        if Parent node is root node then
            Return all matched tuples (MTJNT) as result
            Update cn_newsubspace to fully match to ~CN.
        end if
        Move all matched tuples into appropriate sub-space cn_newsubspace
    end if
end for
```

However, the common middle nodes of different CNs, which distance from any leaf nodes is bigger than or equal to three might be probed multiple times. Therefore, we would like to find the maximum number of such CNs. Notice that, each CN has at least two leaf nodes with at least two different keywords. Therefore, the maximum number of all common middle nodes is $O(S*2^{\text{m-2}})$. Then, the maximum number of those CNs is $O((S*2^{\text{m-2}})^{T_{max}\text{-}3})$. Therefore, the maximum number of probings in MX-structure is $O([S*2^{\text{m}}+(S*2^{\text{m-2}})^{T_{max}\text{-}3}*(S*2^{\text{m-2}})]*n)$ or $O([S*2^{\text{m}}+(S*2^{\text{m-2}})^{T_{max}\text{-}2}]*n)$.

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. Experimental Evaluation

5.1. Setup and Datasets

SS-KWS [24], full mesh (FM) and partial mesh (PM) of S-KWS [19], and our proposed algorithm were implemented by using C++. All data structures and temporary data were entirely kept in the memory. All experiments were performed on Intel Core i7 CPU 870 @ 2.93GHz x 8 computer with 31.4 GiB memory in Ubuntu 13.10 (64 bits).

We used two types of datasets, synthetic and real datasets. For synthetic dataset, we used TPC-H dataset [2], which is about the transactions between customers and products. It is mainly used for testing performance of commercial DBMSs. In this dataset, there are 8 tables and 61 fields. Due to lack of real data stream datasets, we simulated DBLP [1], published in 2015, so that we could work on it as we work on real relational streams. The simulation was done by attaching time stamp to each tuple in DBLP dataset. And the simulator read tuples in the order of their timestamps and sent tuples continuously to the filtering system. DBLP dataset has 4 tables and 11 fields.

As explained earlier, SS-KWS performs better than FM and PM of S-KWS when the tuples coming from relational data streams mostly match CNs that have common edges at leaf nodes, where a lot of processings can be shared among those matched CNs. Therefore, for experimental purposes, we separately prepare 2 datasets, one of which gives advantage to SS-KWS and the other gives advantage to S-KWS. Then, we investigate how the proposed algorithm can handle both kinds of datasets.

Parameters used in the experiment are shown in Table 1. We varied these parameters and compared the performance of the proposed algorithm with compara-

tive algorithms, SS-KWS and PM/FM of S-KWS. The default parameters are written in bold.

5.2. Comparison of Query Plans

We first made a comparison of the number of edges, memory usage, and construction time of query plans of all approaches because they have great impact on the performance. For this experiment, we only used two parameters, number of query keywords and T_{max}, because other parameters do not have any impact on the construction of query plan. We varied the number of query keywords and T_{max} from 2 to 5.

The results are shown in Figures 14, 15, and 16 for DBLP and Figures 17, 18, and 19 for TPCH.

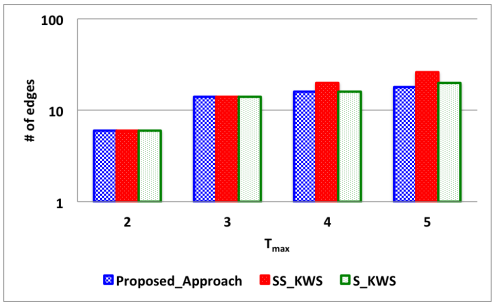
When the number of query keywords and T_{max} are increased, the number of CNs increases. As a result, construction time, memory usage, and the total number of edges in the query plan of all approaches increases for both datasets because more CNs need to be added to the query plan.

We notice that there is an exponential increase of the number of edges in SS-KWS and S-KWS, which is caused by the explosion of number of CNs whose edges cannot be consolidated in their query plans. However, the growth of the proposed scheme is almost linear because it consolidates unique edges into one, and the total number of unique edges, which are the primary/foreign-key relationships between two tables in the schema (which is usually comparatively small), in all CNs slightly increases as the number of CNs increases. As a result, the query plan of the proposed scheme consumes less time and memory than those of the comparative approaches.

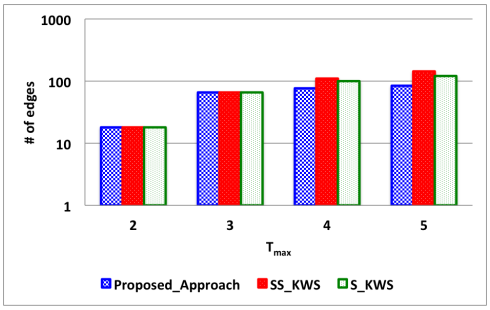
In the proposed approach, all edges need to be maintained by the ID of each corresponding query. Therefore, the maintenance time is still high but less expensive than adding new edges and nodes to the query plan of the comparative approaches. However, IDs of all CNs that have common edge are maintained together as a singe value, so the memory space for storing the labels of edges in the query plan of the proposed approach is far smaller than creating new edges and nodes in the query plans of the comparative approaches.

As a result, even though the query plan of the proposed approach is much more compact than those of the comparative approaches, the improvement of query plan construction time is not so significant than the comparative approaches, but the proposed approach consumes much less memory space.

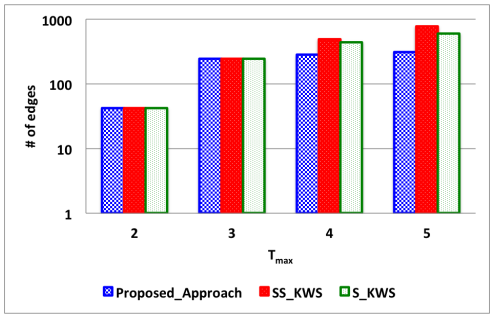
Notice that the explosive increase in size of query plans indicates that the performance of S-KWS and SS-KWS will greatly degrade when the number of query keywords and T_{max} increase, while the proposed scheme can scale well with the increase in number of query keywords and T_{max}.



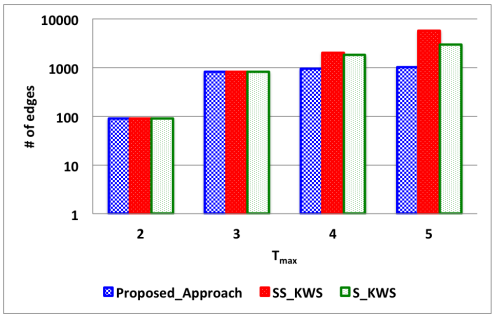
(a) # of keywords = 2.



(b) # of keywords = 3.

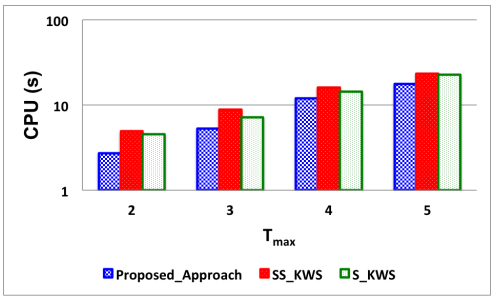


(c) # of keywords = 4.

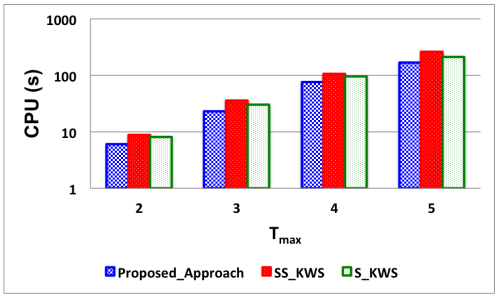


(d) # of keywords = 5.

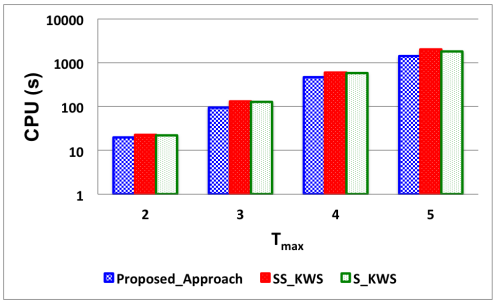
Figure 14: DBLP dataset: Number of edges in the query plans



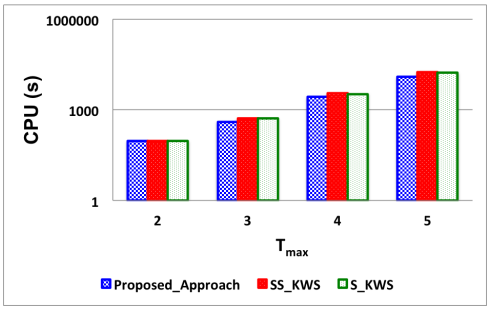
(a) # of keywords = 2.



(b) # of keywords = 3.

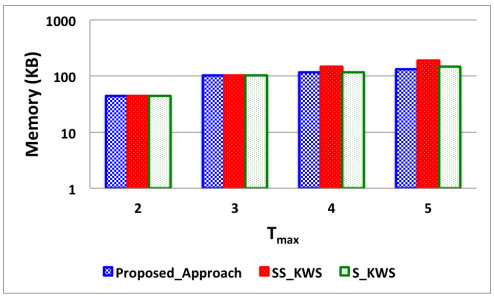


(c) # of keywords = 4.

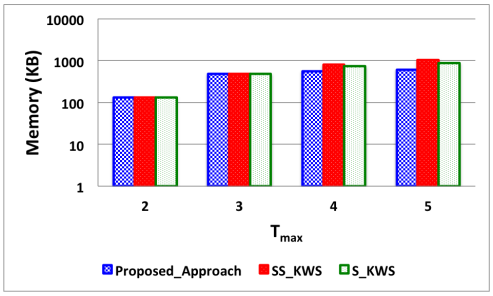


(d) # of keywords = 5.

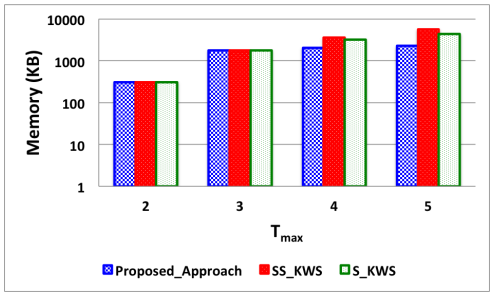
Figure 15: DBLP dataset: Construction time of query plans



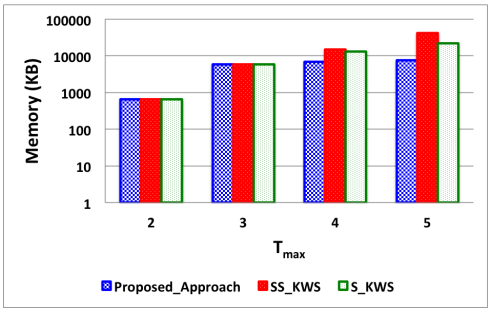
(a) # of keywords = 2.



(b) # of keywords = 3.



(c) # of keywords = 4.



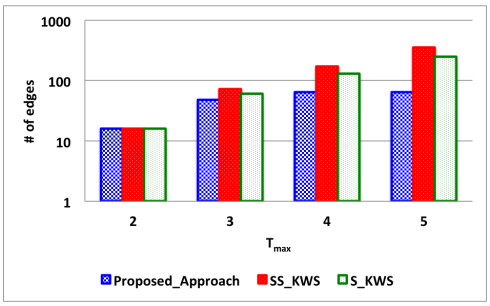
(d) # of keywords = 5.

Figure 16: DBLP dataset: Memory usage of query plans

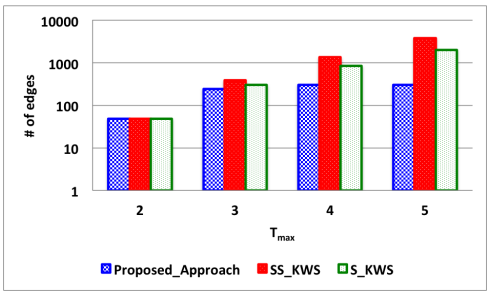
5.3. Performance Comparison

5.3.1. Dataset Giving Advantage to SS-KWS

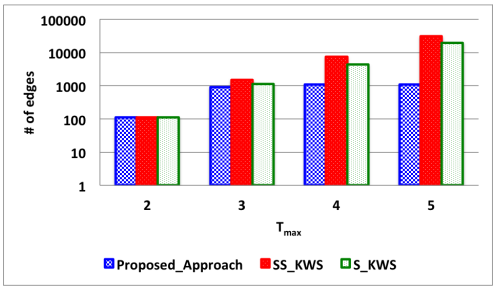
This experiment was done on the datasets of DBLP [1] and TPCH [2] specially prepared so that SS-KWS outperforms S-KWS. We compared CPU running time, memory usage, and total number of probings. The results for DBLP are shown in Figures 20, 21, 22 and 23. Figures 24, 25, 26 and 27 show the results of TPCH



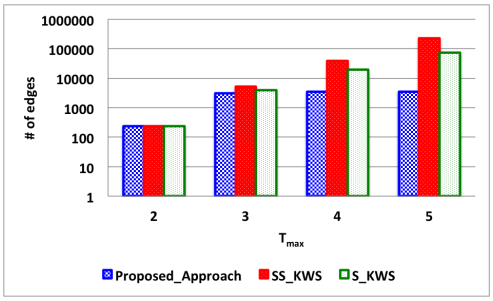
(a) # of keywords = 2.



(b) # of keywords = 3.

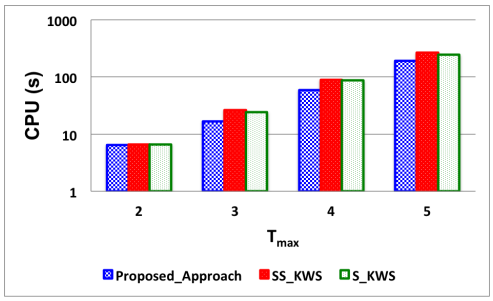


(c) # of keywords = 4.

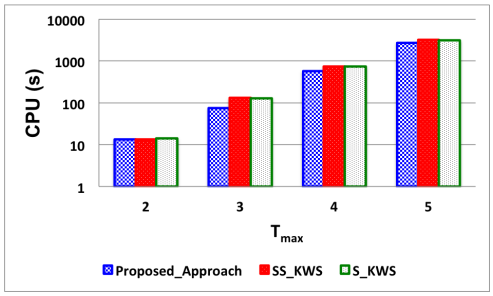


(d) # of keywords = 5.

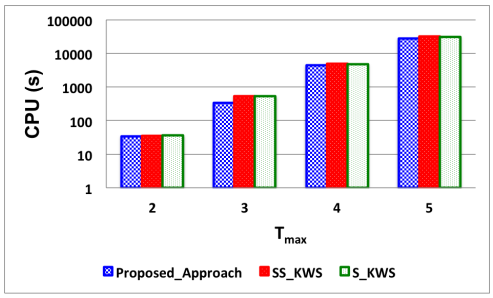
Figure 17: TPCH dataset: Number of edges in the query plans



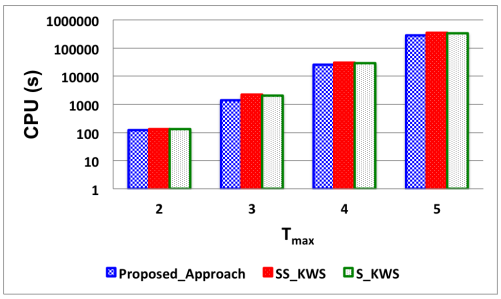
(a) # of keywords = 2.



(b) # of keywords = 3.



(c) # of keywords = 4.

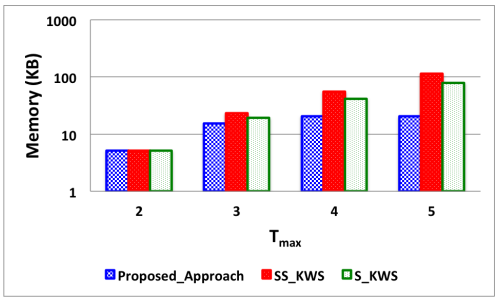


(d) # of keywords = 5.

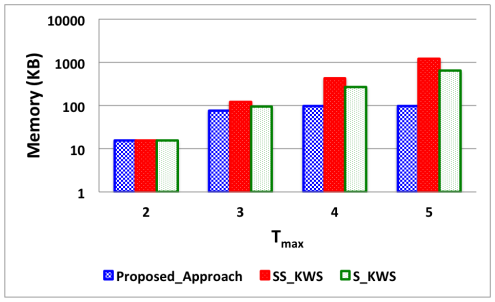
Figure 18: TPCH dataset: Construction time of query plans

dataset.

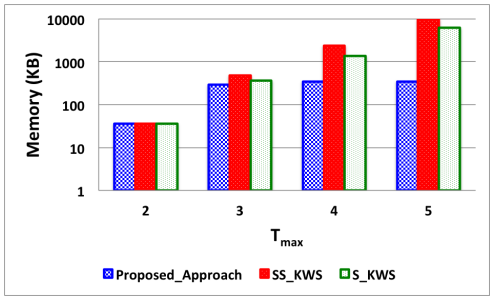
First, we measured the CPU running time and the memory usage when varying the number of keywords (Figures 20(a) and 20(b) for DBLP and Figures 24(a) and 24(b) for TPCH). As can be seen, for both datasets, CPU running time and the memory usage in FM/PM and SS-KWS increase exponentially, but do not in the proposed scheme. As an evidence, the number of prob-



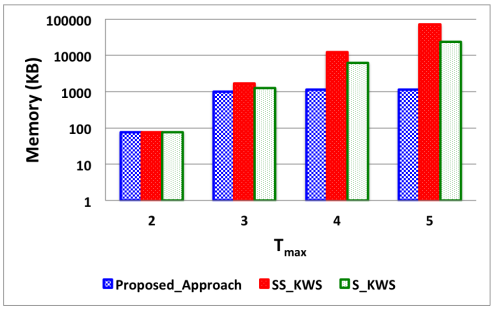
(a) # of keywords = 2.



(b) # of keywords = 3.



(c) # of keywords = 4.



(d) # of keywords = 5.

Figure 19: TPCH dataset: Memory usage of query plans

ings also exponentially increases in FM/PM and SS-KWS as shown in Figures 20(c) for DBLP and 24(c) for TPCH. This is due to the explosion in the size of the query plans of FM/PM and SS-KWS as explained in the above experiments. Similar tendency can be observed when varying T_{max} from two to five (Figures 21 and 25).

Next, we increased the size of window from 10 min, 20 min, 30 min, 40 min, and 50 min. As expected, when the size of window increases, the CPU running time, memory usage, and number of probings of all approaches also increase as shown in Figures 22 and 26 for DBLP and TPCH respectively. This is because fewer tuples in the buffers of all approaches are expired and deleted as a result of the increase in the size of window. Figures 23 (DBLP) and 27 (TPCH) show the impact on the performance of all approaches when varying keyword frequency. When the keyword frequency increases, more tuples tend to contain the keywords of the query. As a result, there were more tuples that need to be joint. Therefore, the CPU running time, memory usage, and number of probings of all approaches also increase. Nevertheless, the total number of CNs does not increase when increasing window size and keyword frequency. Therefore, there is no change in the size of query plans of all approaches, which cause little impact on the performance.

5.3.2. Dataset Giving Advantage to S-KWS

Next experiment was done on the relational streams of DBLP and TPCH. In this experiment, the datasets were prepared to favor FM and PM of S-KWS. The trend is similar to that in the above experiment.

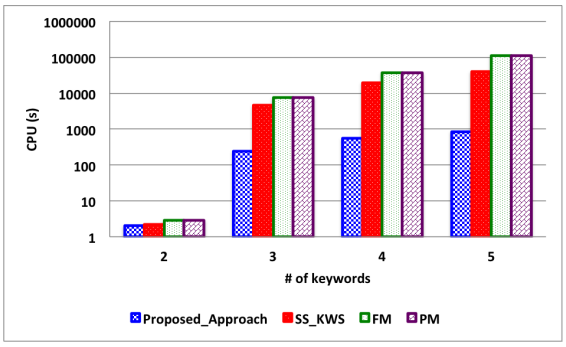
The results are shown in Figures 28, 29, 30 and 31 for DBLP. Figures 32, 33, 34 and 35 show the results of TPCH dataset. As can be seen, the results are similar to the above experiments. The proposed scheme greatly outperforms the existing approaches for all experimental parameters. Notice that, FM/PM of S-KWS outperforms SS-KWS for this dataset.

6. Related Work

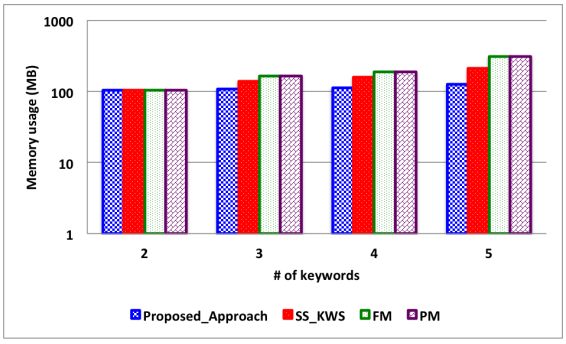
So far, many proposals have been done to enable keyword search on permanently-stored-relational data [6, 3, 7, 9, 17, 23, 21, 28, 20, 26, 15, 4, 14, 18, 12] and few proposals on relational data streams [19, 24].

The works on keyword search on permanently-stored-relational data [6, 3, 7, 9, 17, 23, 21, 28, 20, 26, 15, 4, 14, 18, 12] can be typically categorized into two groups: graph-based approach and candidate network-based approach.

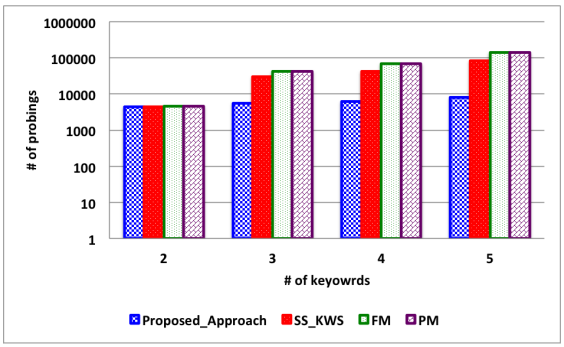
In graph-based approach, relational data is model as connected graphs, where nodes represent tuples and edges represent relationships between tuples. Therefore, keyword search over relational data by using this



(a) CPU running times



(b) Memory usage



(c) # of probings

Figure 20: DBLP dataset (advantageous to SS-KWS): Varying # of keywords

approach is to find connected trees that contain all query keywords. There are a lot of variants of this approach, which aim at finding top-k connected trees. Some famous works are BANKS-I [8], BANKS-II [16], and BLINKS [12]. BANKS-I [8] and BANKS-II [16] were proposed by using backward and bidirectional searching techniques for efficient retrieval of top-k search results. BLINKS [12] proposes a bi-level indexing and query processing scheme for top-k keyword search on graphs. Data graphs are partitioned into blocks in order to reduce index space.

In candidate network-based approach, keyword search over relational data is done by following two main processing steps: 1) pre-processing step, and 2) filtering step. In the pre-processing step, all results' templates, called candidate networks (CNs), are generated by using the given keywords and schema of relational data. In the filtering step, all CNs are used to evaluated against all tuples in the relational data to find the search's results. Evaluating all CNs independently is not efficient because a lot of CNs might have some common parts, whose processing can be shared. For this purpose, DISCOVER [15] and DBXPLORER [4] were proposed. In DISCOVER [15], a plan is built by combining all CNs for efficient evaluation. DBXPLORER-II [14] adopts IR-style document-relevance ranking technique to keyword search over relational data. It focuses on computing top-k matches of keyword search rather than all matches. Since the total number of CNs can be very big, and evaluation of all CNs is costly, [23] proposes an algorithm to rank all CNs, and only top-k CNs are chosen to evaluate against relational data. SPARK [18] proposes a simple but effective ranking method for processing keyword search on relational databases. This new ranking method allows minimal accesses to the databases.

S-KWS [19] is the first work to enable keyword search over relational data streams. Similar to previous works for permanently-stored relational databases, all possible CNs are created from the given keyword search and relational data streams' schemas. They propose two main approaches, full operator mesh and partial operator mesh, which make use of common edge attached to the root nodes to elevate query searching. And the most recent work to enable keyword search on relational data streams is SS-KWS [24], which is proved to outperform S-KWS [19] when most tuples from the relational data streams match CNs that have common edges at leaf-nodes.

7. Conclusion

In this paper, we have proposed an improved method of keyword search over relational streams. In the proposed scheme, candidate networks are merged into a novel data structure called MX-structure, and keyword search is efficiently processed based on the proposed algorithm with the help of MX-structure.

To prove the effectiveness of the proposed approach, extensive experiments have been done on both synthetic and real datasets. A variety of parameters, such as number of query keywords, T_{max}, window size, and key-

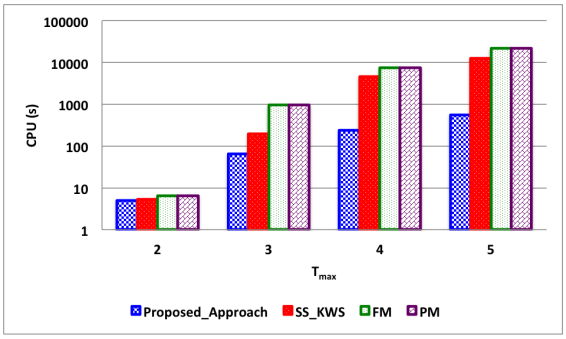
word frequency, are used to measure how they affect the efficiency of the proposed approach and the comparative approaches. The experimental results show that the proposed scheme significantly outperforms the comparative approaches with regards to any parameters. Experimental results also prove that the performance of the comparative approaches greatly degrades when the number of query keywords and/or T_{max} are increasing because their query plans become exponentially big in terms of number of edges, so their performances become inefficient. The proposed approach can scale very well with respect to any parameter, and in particular greatly outperforms the comparative approaches when the number of query keywords and/or T_{max} are increased. Therefore, our proposed approach is more suitable for real search engine.

In this work, we have noticed that CN-based approach has some limitations. In particular some CNs are not used due to the biased keyword distribution in relational streams. For the future work, we plan to exploit such locality to enhance the performance by generating and processing only CNs that can produce results.

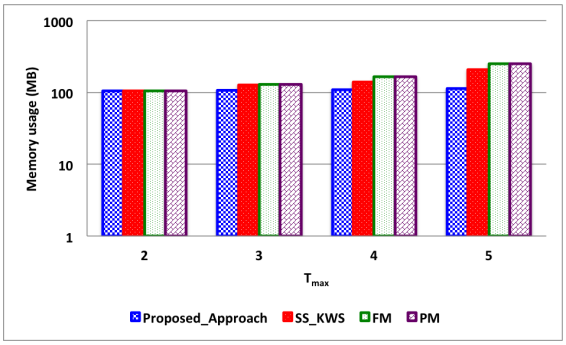
In addition, currently, our proposed approach and the comparative approaches, which are CN-based approach, only support a single keyword search at a time over relational streams. It is close to impossible to process multiple keyword searches at the same time by using the above CN-based approaches because of the explosive blow up of all CNs. This is important and we hope that the idea of our proposed approach can be used to explore other approaches that does not rely on CN with an attempt to enable the processing of multiple keyword search over relational streams.

Acknowledgment

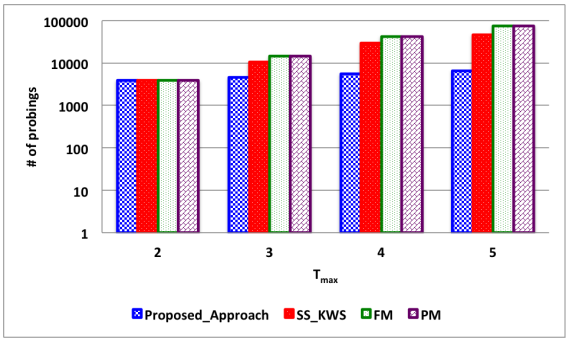
This work has been partly supported by the **NICT BigClouT** project.



(a) CPU running times

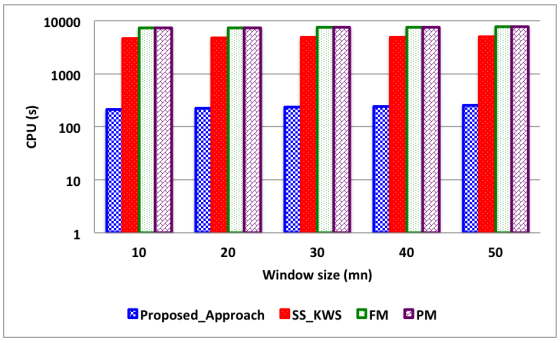


(b) Memory usage

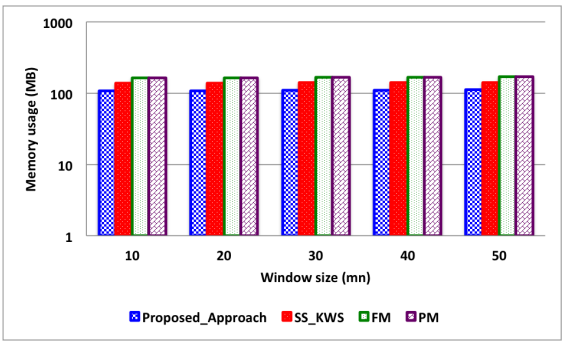


(c) # of probings

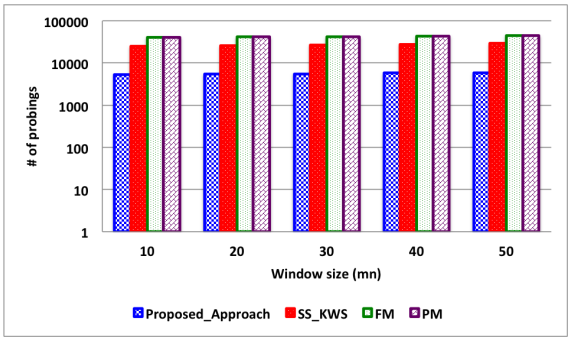
Figure 21: DBLP dataset (advantageous to SS-KWS): Varying T_{max}



(a) CPU running times

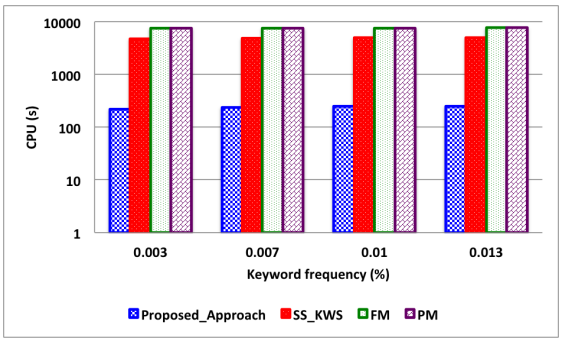


(b) Memory usage

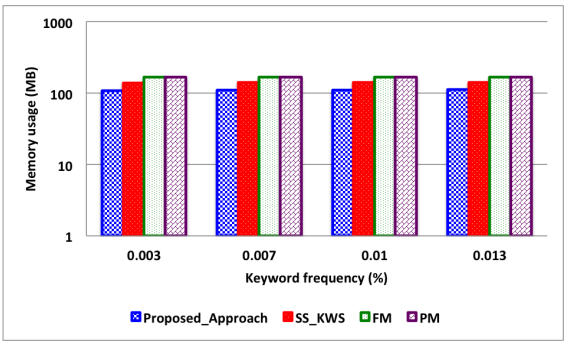


(c) # of probings

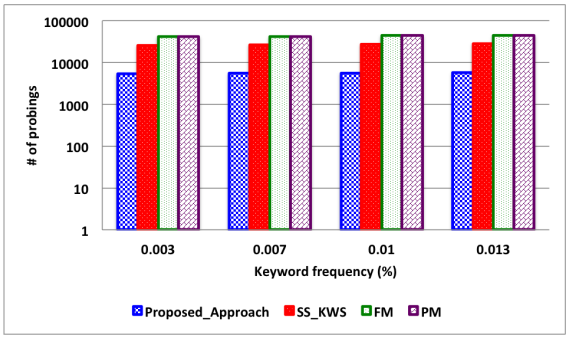
Figure 22: DBLP dataset (advantageous to SS-KWS): Varying window size



(a) CPU running times

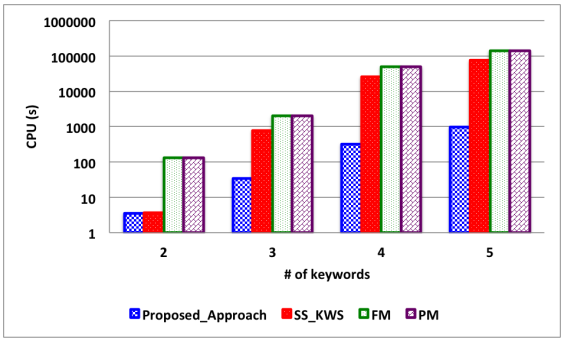


(b) Memory usage

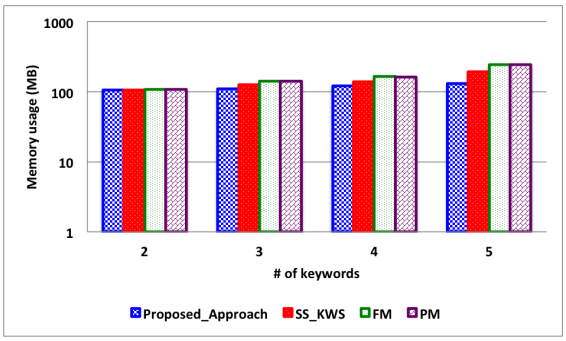


(c) # of probings

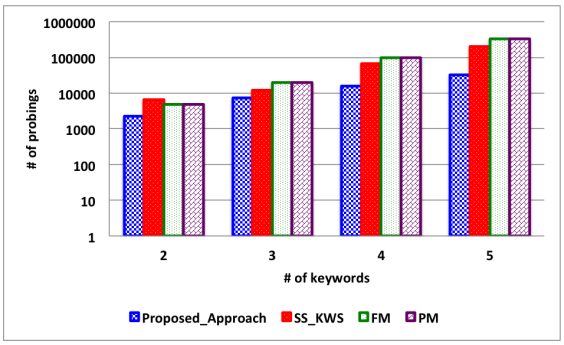
Figure 23: DBLP dataset (advantageous to SS-KWS): Varying keyword frequency



(a) CPU running times

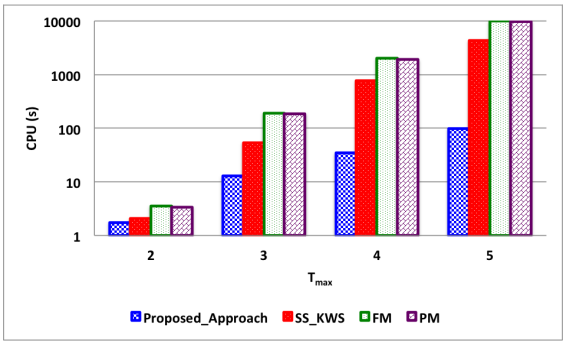


(b) Memory usage

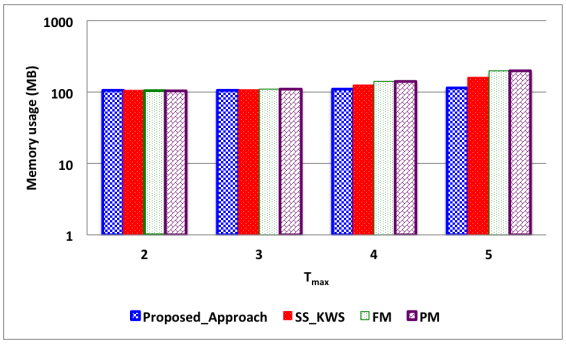


(c) # of probings

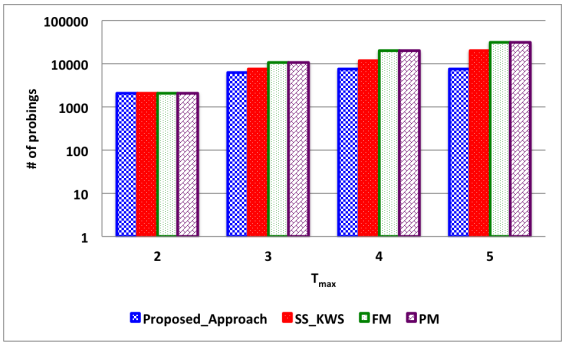
Figure 24: TPCH dataset (advantageous to SS-KWS): Varying # of keywords



(a) CPU running times

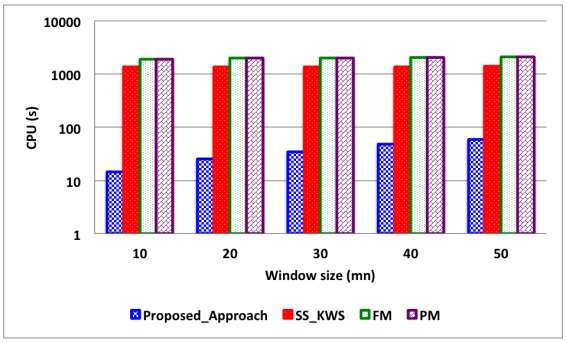


(b) Memory usage

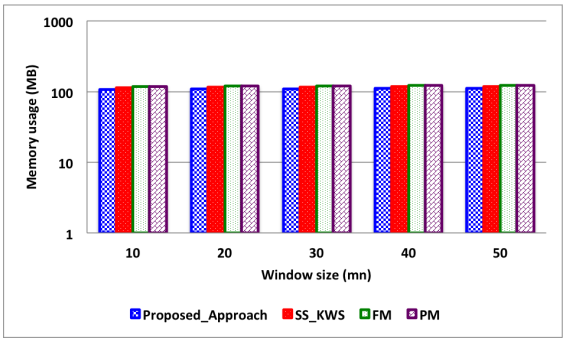


(c) # of probings

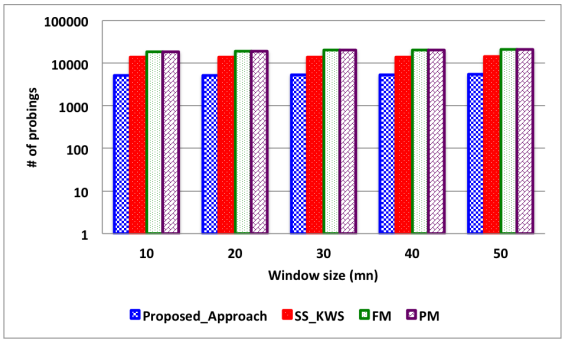
Figure 25: TPCH dataset (advantageous to SS-KWS): Varying T_{max}



(a) CPU running times

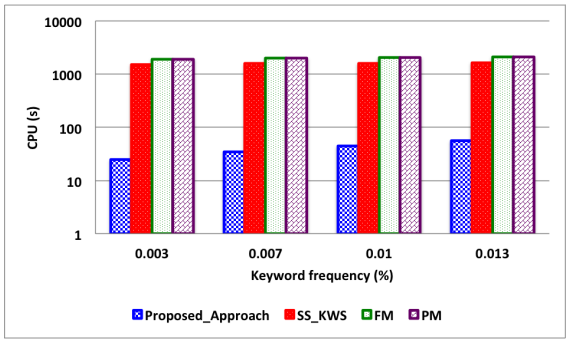


(b) Memory usage

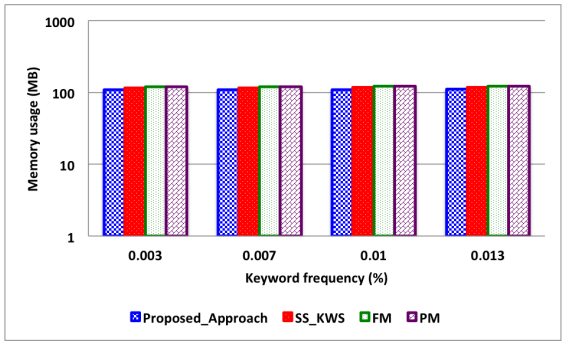


(c) # of probings

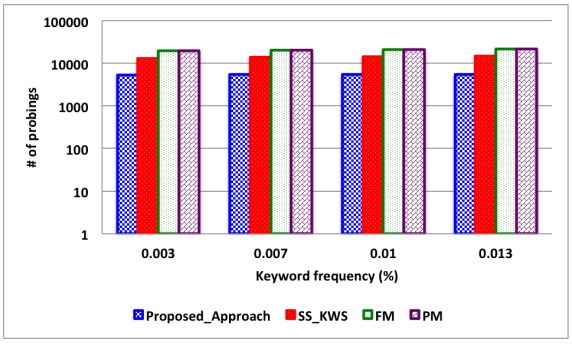
Figure 26: TPCH dataset (advantageous to SS-KWS): Varying window size



(a) CPU running times

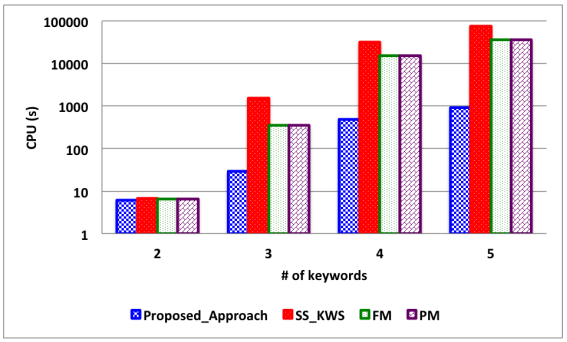


(b) Memory usage

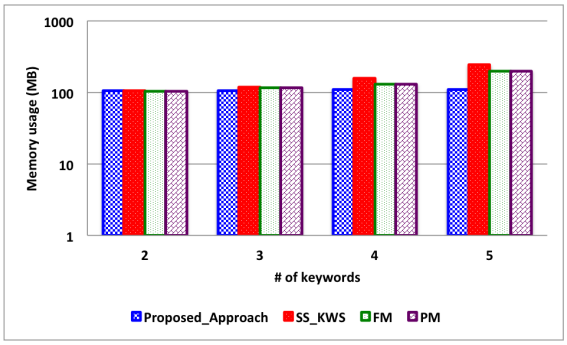


(c) # of probings

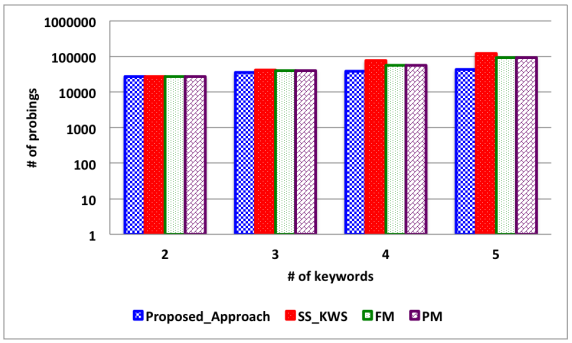
Figure 27: TPCH dataset (advantageous to SS-KWS): Varying keyword frequency



(a) CPU running times

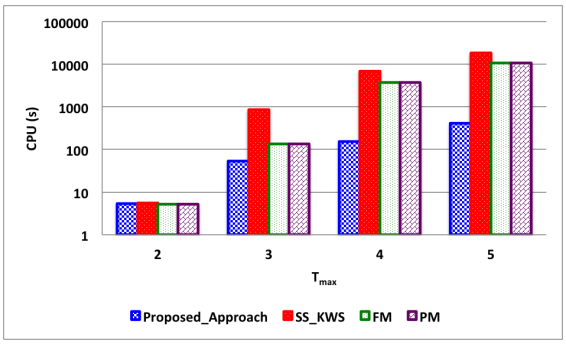


(b) Memory usage

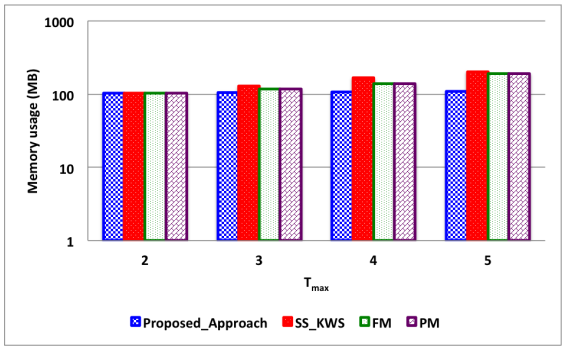


(c) # of probings

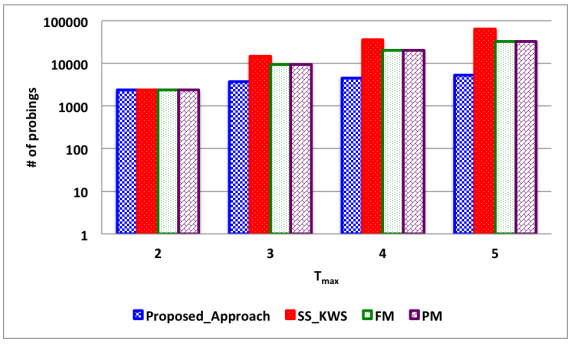
Figure 28: DBLP dataset (advantageous to S-KWS): Varying # of keywords



(a) CPU running times

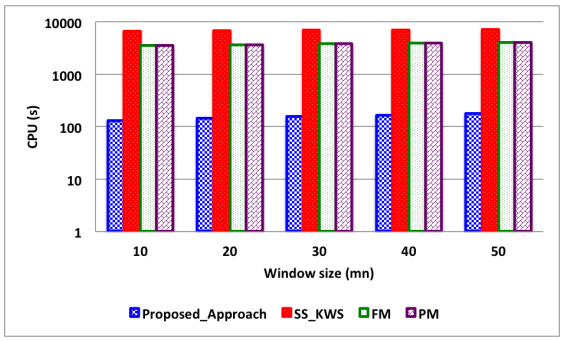


(b) Memory usage

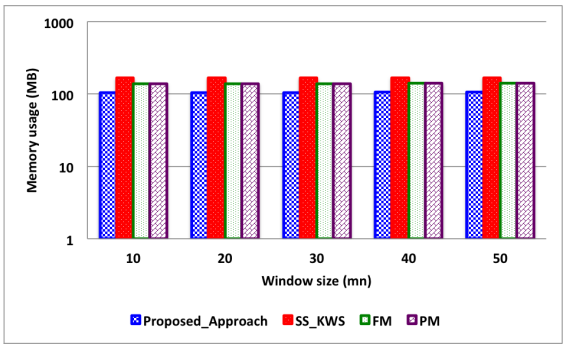


(c) # of probings

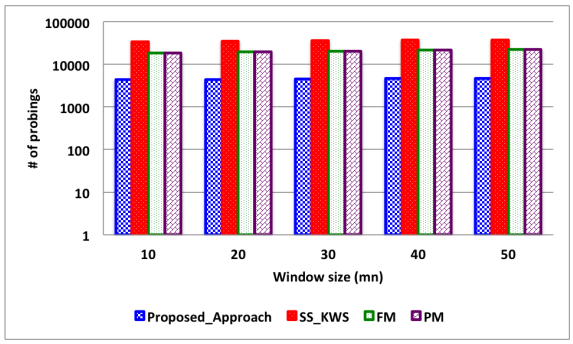
Figure 29: DBLP dataset (advantageous to S-KWS): Varying T_{max}



(a) CPU running times

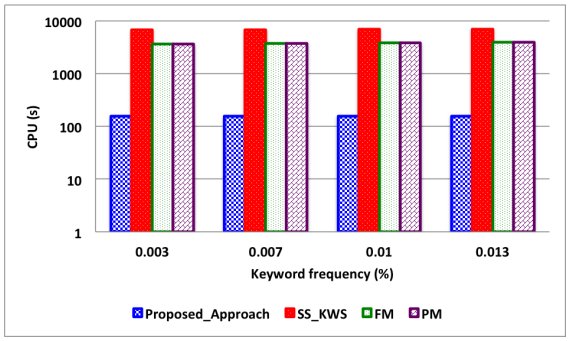


(b) Memory usage

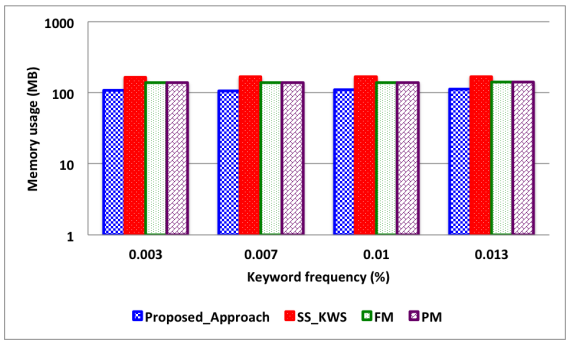


(c) # of probings

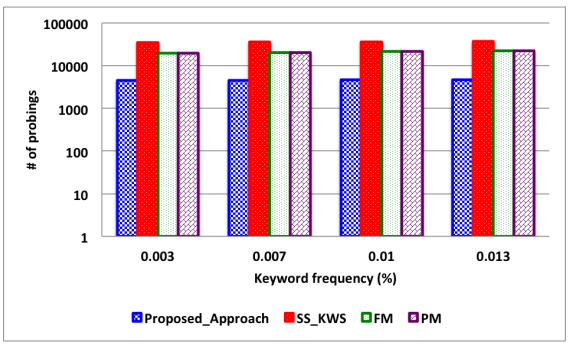
Figure 30: DBLP dataset (advantageous to S-KWS): Varying window size



(a) CPU running times

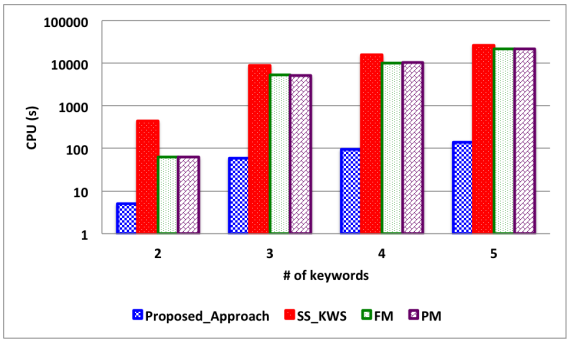


(b) Memory usage

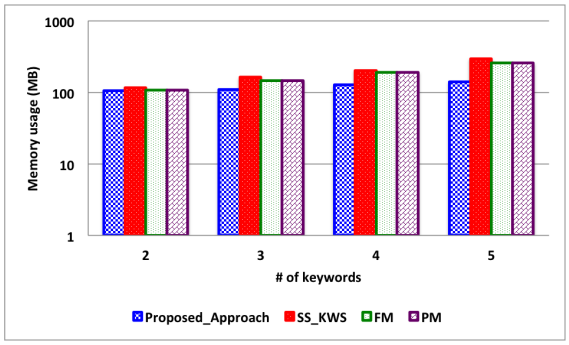


(c) # of probings

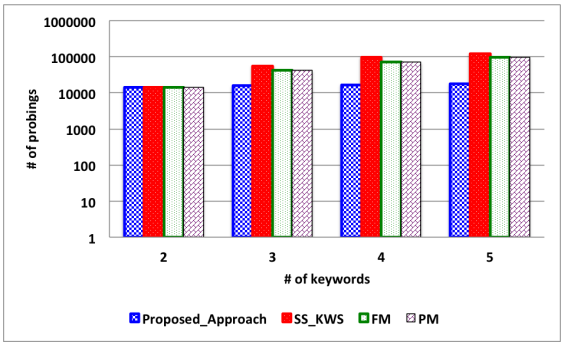
Figure 31: DBLP dataset (advantageous to S-KWS): Varying keyword frequency



(a) CPU running times

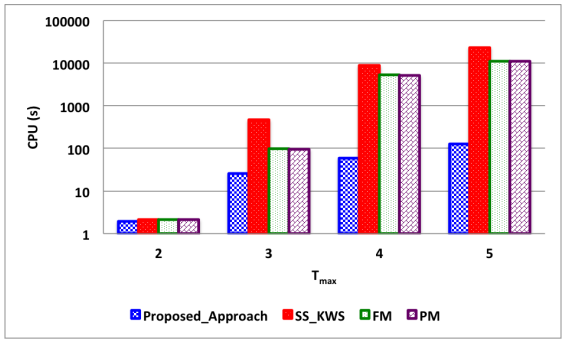


(b) Memory usage

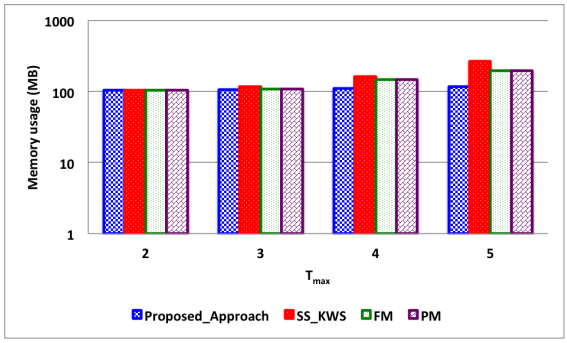


(c) # of probings

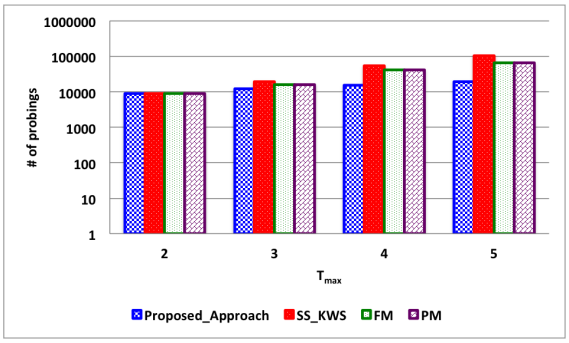
Figure 32: TPCH dataset (advantageous to S-KWS): Varying # of keywords



(a) CPU running times

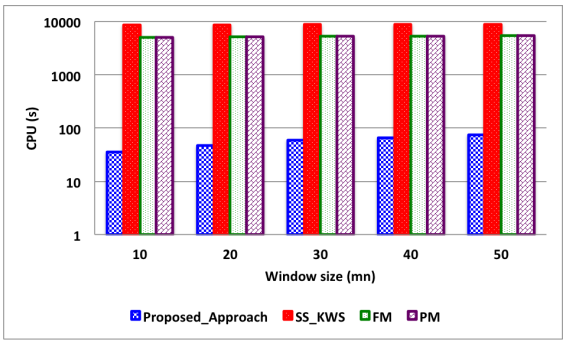


(b) Memory usage

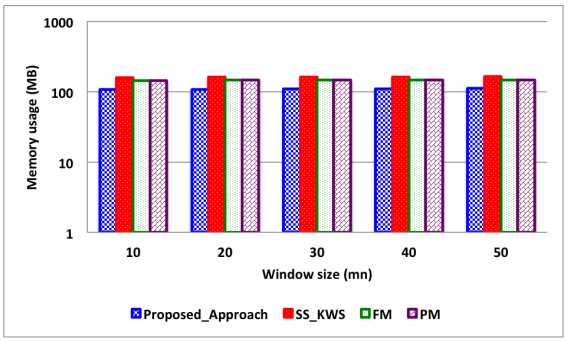


(c) # of probings

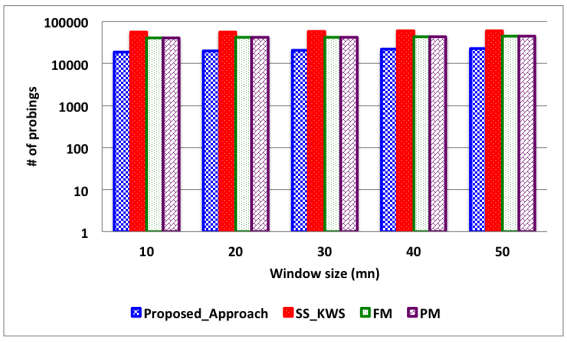
Figure 33: TPCH dataset (advantageous to S-KWS): Varying T_{max}



(a) CPU running times

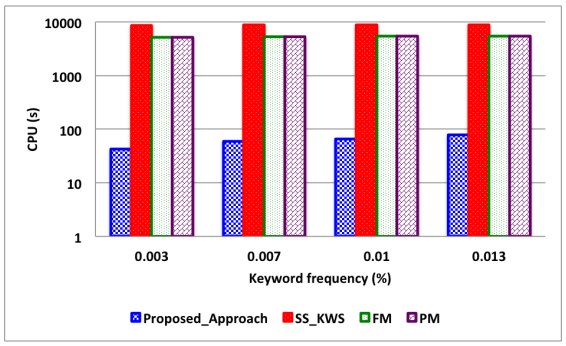


(b) Memory usage

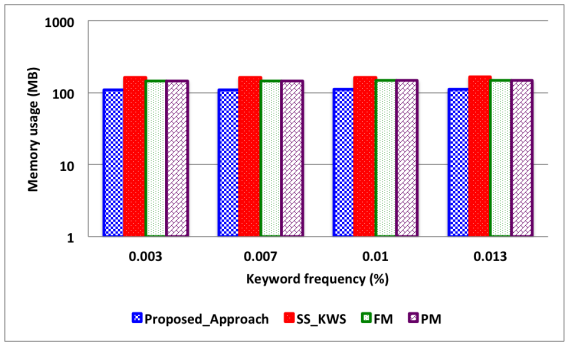


(c) # of probings

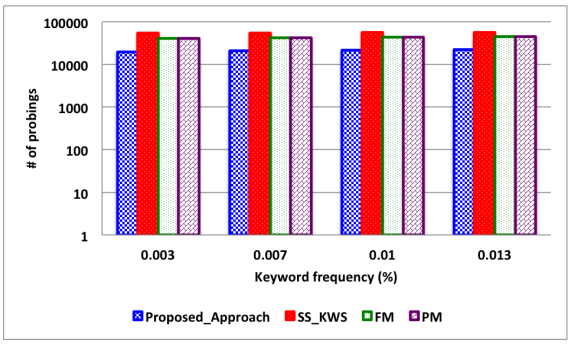
Figure 34: TPCH dataset (advantageous to S-KWS): Varying window size



(a) CPU running times



(b) Memory usage



(c) # of probings

Figure 35: TPCH dataset (advantageous to S-KWS): Varying keyword frequency

References

[1] Computer science bibliography. *http://dblp.uni-trier.de/xml/*, 2015.

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

[3] D. J. Abadi, D. Carney, U. Centinternel, M. Cherniack, C. Convery, S. Lee, M. Stonebraker, N. Tatbul, and S.B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12:120–139, 2003.

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

[5] 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.

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

[7] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: authority-based keyword search in databases. In *VLDB*, Toronto, Canada, 2004.

[8] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword search and browsing in databases using banks. In *In Proceedings of ICDE*, 2002.

[9] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *VLDB*, Toronto, Canada, 2004.

[10] 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.

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

[12] H. He, H. Wang, Y. Wang, and X. Zhou. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[13] K. Hogan. Interpreting hitwise statistics on longer queries. *Technicall report, Ask.com*, 2009.

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

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

[16] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *In Proceedings of VLDB*, 2005.

[17] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, Chicago, USA, 2006.

[18] Y. Luo, X. Lin, and W. P.S. Spark: top-k keyword query in relational databases. In *SIGMOD*, 2007.

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

[20] K. Mehdi, A. Aijun, C. Nick, G. Parke, S. Jaroslaw, and Y. Xiaohui. Meanks: meaningful keyword search in relational databases with complex schema. In *SIGMOD*, Utah, USA, 2014.

[21] 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.

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

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

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

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

[26] X. Yanwei, G. Jihong, and I. Yoshiharu. Scalable top-k keyword search in relational databases. In *DASFAA*, Busan, Korea, 2012.

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

[28] Z. Zhong, B. Zhifeng, L. Mong, and L. Tok. Towards an interactive keyword search over relational databases. *WWW journal (companion volume)*, 24:259–262, 2015.