Mining Frequent Co-occurrence Patterns across Multiple Data Streams

Ziqiang Yu
School of Computer Science & Technology
Shandong University
Jinan, China
zqy800@gmail.com

Xiaohui Yu*
School of Computer Science & Technology
Shandong University
Jinan, China
xyu@sdu.edu.cn

Yang Liu
School of Computer Science & Technology
Shandong University
Jinan, China
yliu@sdu.edu.cn

Wenzhu Li
School of Computer Science & Technology
Shandong University
Jinan, China
sduliwenzhu@gmail.com

Jian Pei
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada, V5A 1S6
jpei@cs.sfu.ca

ABSTRACT

This paper studies the problem of mining frequent co-occurrence patterns across multiple data streams, which has not been addressed by existing works. Co-occurrence pattern in this context refers to the case that the same group of objects appear consecutively in multiple streams over a short time span, signaling tight correlations between these objects. The need for mining such patterns in real-time arises in a variety of applications ranging from crime prevention to location-based services to event discovery in social media.

Since the data streams are usually fast, continuous, and unbounded, existing methods on mining frequent patterns requiring more than one pass over the data cannot be directly applied. Therefore, we propose DIMine and CooMine, two algorithms to discover frequent co-occurrence patterns across multiple data streams. DIMine is an Apriori-style algorithm based on an inverted index, while CooMine uses an in-memory data structure called the Seg-tree to compactly index the data that are already seen but have not expired yet. CooMine employs a one-pass algorithm that uses the filter-and-refine strategy to obtain the co-occurrence patterns from the Seg-tree as updates to the streams arrive. Extensive experiments on two real datasets demonstrate the superiority of the proposed approaches over a baseline method, and show their respective applicability in different senarios.

1. INTRODUCTION

We study the problem of mining frequency co-occurrence patterns across multiple data streams. Given a set of unbounded streams of objects s_i $(i = 1, 2, \ldots, n)$, we call a set of objects $\mathcal{O} = (o_1, o_2, \ldots, o_k)$ a frequent co-occurrence pattern (FCP) if (1) they appear in at least θ streams within a period of time no longer than τ, and (2) their appearance within each of those streams happens within a time window of size no greater than ξ, where θ, τ, and ξ are user-specified thresholds.

1.1 Motivation

FCPs usually indicate strong correlations between these objects, and their timely discovery has applications in a wide spectrum of contexts. The following are some typical examples.

- *Crime prevention*. An increasing number of cities are now having traffic surveillance cameras installed on major roads and intersections for traffic management and public safety. A picture is taken when a vehicle passes by, and a structured vehicle passing record (VPRs) is sent to the data center for processing. Thus, each camera effectively produces a continuous stream of VPRs. Finding FCPs across these streams can help uncover groups of vehicles that travel together within a short time span, which is often a good indicator for potential gang crimes.

- *Discovering emerging topics*. Each user of a microblogging platform (e.g., Twitter) can be considered to produce a stream of words by posting microblogs. The intensive co-occurrence of a set of keywords in many streams (microblogs) over a short period of time can often imply the emergence of a new topic.

- *Location-based services*. Check-in apps (e.g., Foursquare) allow mobile users to check-in to the places they are located. Finding FCPs across the streams in real-time, where each stream consists of the checkin locations of a user, would help discover groups of people that are currently hanging out together, so that location-based advertising can be better targeted (e.g., offering group buying deals).

- *E-Commerce*. The browsing trace of a particular user on a e-commerce website contains a stream of items she has visited. When a set of items appear in the traces of a lot of users over a short period, it could be a sign of strong correlation

*Corresponding author.

10.5441/002/edbt.2015.08

between those items, and sales strategies can be adjusted accordingly in real-time (e.g., offering combo deals for those products).

In all of the examples above, a common theme is the need to discover the frequent co-occurrence of a set of items (vehicles, people, keywords, etc.) over a short time span in multiple streams.

1.2 Challenges

Mining FCPs in realtime present some unique challenges. First, the streams are unbounded and often contain a vast volume of data with high arrival rates, allowing only one pass over the data. For example, the peak of 143,199 tweets per second were recorded by Twitter on August 3, 2013, and on average, 58 million tweets are produced per day. As another example, in the City of Jinan, a provincial capital in eastern China with a population of 6 million, the traffic surveillance cameras (each generating a stream) produce 20 million VPRs per day on average. Second, mining FCPs is a cross-stream operation, making it highly complex when many streams are involved. For instance, there are around 3,000 traffic surveillance cameras installed in Jinan; not to mention the more than 600 million active registered users of Twitter.

Existing methods for mining frequent patterns cannot be directly applied to solve the FCP mining problem. Most of the algorithms for mining frequent patterns from databases require multiple passes over the data, rendering them inapplicable in the streaming data context. Methods proposed for data streams [4, 10, 18, 15], on the other hand, focus on mining patterns from a *single* stream where the frequent patterns are defined as those that occur more often than a given threshold in *that* stream. Contrastingly, the problem we tackle is concerned with the occurrences of patterns within *multiple* streams, where the number of streams in which a pattern appears is an important parameter.

Probably the most related work to ours is that by Guo et al. [8] which considers frequent patterns in multiple data streams. But according to the their definition, whether a pattern is considered frequent totally depends on the number of its occurrences within a single stream. The frequent patterns are discovered separately from each stream, and then analysis of those patterns is performed to find the interesting ones based on their presence across multiple streams. In contrast, in order for a pattern to be considered frequent in our problem, it has to appear in at least θ streams within a short period of time. It is not the number of times a patten appearing in any single stream that matters; it is the number of streams the pattern appears in. Moreover, the algorithms proposed by Guo et al. are approximate, whereas we focus on computing exact solutions.

One seemingly promising strategy to simplify this problem is to first combine all streams into one, and then mine the FCPs on the combined data stream with some time window constraints. However, the complexity of the problem remains the same as discovering FCPs from the combined data stream still requires the differentiation between the component streams (e.g., we count a pattern twice if it appears in two different component streams, but only once if it appears twice in the same component stream), as well as restriction on the time interval of the pattern occurring in these data streams.

1.3 Our proposal

To address the above challenges, we propose two algorithms for discovering FCPs across multiple streams. They both adopt a filter-and-refine strategy with two stages. The first stage produces a set of candidate co-occurrence patterns (CPs) from the data streams, and the second stage generates the FCPs from those candidate CPs. To limit the scope of search, we divide each data stream into overlapping segments, guaranteeing that any FCP can only appear in those segments. This allows us to solve the problem of finding FCPs by focusing only on the recent segments in each stream.

As we are dealing with unbounded streams, the search for new candidate CPs is an ongoing process. The key is to efficiently search for new candidate CPs when there are newly arrived objects in any of the streams. In the DIMine approach, we introduce an inverted index called DI-Index to index the existing segments, and it computes FCPs based on the DI-Index with an Apriori-style heuristic. Although straightforward to implement, it needs to be improved in terms of memory consumption and maintenance cost.

In the CooMine approach, we propose a compact in-memory data structure called Seg-Tree to maintain the existing segments, which are dynamically updated as streams proceed, including the deletion of segments when they are guaranteed not to contain any FCPs. When a new segment is generated resulting from newly arrived objects, we search the existing segments using the Seg-Tree to find the common CPs they all contain, which form the set of candidate CPs that must be examined further. With all candidate CPs obtained, an Apriori-style algorithm is then used to compute the FCPs.

1.4 Contributions and outline

The contributions of this paper can be summarized as follows.

- For the first time, we tackle the problem of mining frequent co-occurrence patterns across data streams, an operation that has extensive applications in a variety of contexts but cannot be readily solved using existing frequent pattern mining techniques.

- We propose the DIMine and CooMine approaches with several facilities specifically designed for discovering FCPs across unbounded streams. The CooMine method includes the segmentation of data streams to limit the scope of processing, the Seg-Tree structure that compactly indexes and stores the segments for further processing, the SLCP algorithm that searches the candidate CPs for each new incoming segment, and an Apriori-style algorithm for generating FCPs from candidate CPs.

- Extensive experiments are conducted on two real data sets (a traffic surveillance data set and a Twitter data set), which demonstrate the superiority of two proposed approaches over the baseline method. Indeed, the CooMine method has been deployed in the City of Jinan's Traffic Surveillance and Public Safety Control System, and it has helped the early detection of dozens of criminal activities including vehicle thefts and burglaries over a period of six months.

The remainder of the paper is organized as follows. Section 2 reviews the related work and discusses the distinctions between our proposal and existing methods. Section 3 introduces the preliminaries, and presents the DIMine method as a first attempt to solve the FCP mining problem. Section 4 presents the Seg-tree, which is a novel index structure to maintain the valid segments. The CooMine algorithm based on Seg-tree is described in detail in Section 5. Section 6 presents the experimental results, and Section 7 concludes this paper.

2. RELATED WORK

2.1 Mining frequent patterns from databases

Mining frequent patterns (FPs) from databases is one the classic topic in data mining and has been well studied [2]. Agrawal et al. [3] present the Apriori algorithm to discover the association rules in databases, followed by a great volume of studies (e.g., [19, 18]) that also adopt Apriori-like approaches. All of them require repeatedly scanning the databases to generate candidate frequent patterns. Han et al [9] propose the FP-Tree that can discover frequent patterns without candidate generation, but the method still requires two database scans. Since the data streams are unbounded, in general only one-pass over the data is allowed, rendering those methods inapplicable.

The methods proposed by Tanbeer et al. [20] are claimed to require only a single pass over the database. However, the premise is that all relevant information is captured and stored during this pass over the database, which is then processed in later stages. This is clearly infeasible for unbounded streams.

2.2 Mining frequent patterns from data streams

Mining frequent patterns in data streams has also received considerable attention. The works based on the landmark model (e.g., [15, 21]) aim at mining frequent patterns from the start of the stream to the current moment. Other studies [4, 13, 12, 17] utilize the sliding window technique to discover the recent frequent patterns from data streams. All of these approaches obtain only approximate results with an error bound.

Some methods are proposed to obtain the exact set of recent frequent patterns from data streams [4, 12]. Chang et al [4] propose a data mining method for finding recent frequent itemsets adaptively over an online data stream with diminishing effect for old transactions. Leung and Khan propose a novel tree structure (DSTree) to capture important data from the streams to mine exact frequent patterns [12].

Although these approaches can mine frequent patterns in data streams, they cannot be directly employed to solve the FCP mining problem. Most of them focus on mining frequent patterns in a single data stream, while our task is finding FCPs across multiple data streams.

The only exception to our knowledge is the H-Stream algorithm to discover frequent patterns from multiple data streams [8]. At a first glance, that problem is very similar to ours, but indeed they are quite different. As discussed in Section 1, that work aims to search the frequent patterns that take place in multiple data streams, but does not care about the time interval of the frequent patterns occurring in these data streams. In our problem, the time interval of any FCP across multiple data streams cannot be greater than the specified threshold. Second, this work provides an approximate method but our approach can obtain the exact results. Therefore, the H-Stream algorithm cannot be applied to solve our problem.

The aforementioned existing works about mining frequent patterns from data streams always assume the streams are composed of transactions, and they can directly discover frequent patterns within transactions. But in our problem, the streams just consist of continuous unbounded objects and we have to first determine which objects in one data stream probably can construct a FCP, making the problem more complex.

2.3 Mining spatio-temporal patterns

An extensive body of literature exists on mining interesting patterns from spatio-temporal data [7, 11, 14, 5]. Some [11, 5] study the discovery of valuable trajectories of moving objects, while Li et al. [14] focus on mining spatial association rules. Moreover, there also exist works [7, 16] that study the problem of mining temporally annotated sequences. However, none of them addresses the same problem as mining FCPs, and methods proposed therein cannot be directly applied to our problem.

3. PRELIMINARIES AND A FIRST ATTEMPT

In this section, we first introduce the terminology used in the following discussion and formally define the problem, and then present a first attempt to the FCP mining problem.

3.1 Preliminaries and problem definition

DEFINITION 1. *(***Data stream***) A data stream is a continuous ordered (by timestamps) sequence of objects.*

Data streams are unbounded, and thus it is infeasible to store the data stream locally in its entirety. For an object o_i in the data stream, its ID and timestamp are id_i and t_i respectively.

DEFINITION 2. *(***Co-occurrence pattern, or CP***) For a set of objects $\mathcal{O}$=$\{o_1, o_2, \cdots, o_k\}$ that appear in a data stream s_i, we say that $\mathcal{O}$ is a co-occurrence pattern CP$_k$ (where k is the number of objects) if $t^{\mathcal{O}^{s_i}}_{max} - t^{\mathcal{O}^{s_i}}_{min} \leq \xi$, where $t^{\mathcal{O}^{s_i}}_{max} = \max\{t_1, \cdots, t_k\}$, $t^{\mathcal{O}^{s_i}}_{min} = \min\{t_1, \cdots, t_k\}$, and ξ is a user-specified threshold.*

DEFINITION 3. **(Frequent co-occurrence pattern, or FCP***) A co-occurrence pattern CP$_k$ that appears in a set of l streams $\{s_1, s_2, \ldots, s_l\}$ is deemed a frequent co-occurrence pattern, FCP$_k$, if it satisfies the following conditions: (1) $l \geq \theta$, where θ is a user specified threshold; and (2) $T^{\max}_{\mathcal{O}} - T^{\min}_{\mathcal{O}} \leq \tau$, where $T^{\max}_{\mathcal{O}} = \max\{t^{\mathcal{O}^{s_1}}_{max}, \cdots, t^{\mathcal{O}^{s_l}}_{max}\}$, $T^{\min}_{\mathcal{O}} = \min\{t^{\mathcal{O}^{s_1}}_{min}, \cdots, t^{\mathcal{O}^{s_l}}_{min}\}$, and τ is a user-specified threshold and $\tau \gg \xi$.*

To understand the differences between our problem and the frequent pattern mining problem defined in earlier literature, we also present below the definition of frequent pattern in data streams given in [6].

DEFINITION 4. *(***Frequent patterns, or GHP-FP [6]***) Let the* frequency *of an itemset I over a time period (τ) in the data stream s_i be the number of transactions (where transactions correspond to non-overlapping time windows) in which I occurs. The support of I is the frequency divided by the total number of transactions observed within the time interval t_w of s_i. The itemset I is deemed a frequent pattern if its support is no less than a user-specified threshold, δ.*

We use some examples to illustrate the differences between the above definitions. In Section 1, we have discussed the scenario of each surveillance camera producing a continuous stream of VPRs, and the stream can be divided into time windows, where each window can considered as corresponding to a transaction defined in Definition 4. Consider the following cases.

Case 1: A group of cars are captured by one camera within time span ξ. This group forms a CP even if it is not captured by any other cameras.

Case 2: Within the time interval τ, a group of cars are captured by the same camera together in k non-overlapping time windows and the total number of such windows within τ is f. If $k/f \geq \delta$, this group constitutes a GHP-FP. It is not necessarily a CP unless this group of cars appear within time span ξ ($\xi \ll \tau$) at least once.

Case 3: Within the time interval τ , a group of cars are captured by m different cameras and they pass each camera within the time span ξ. If $m \geq \theta$ (where θ is a threshold that corresponds to the support δ in Definition 4), then this groups is deemed a FCP. By definition, it is also a CP.

Problem Statement. Given a set of data streams of objects and the values of parameters θ, ξ and τ, the problem of mining frequent co-occurrence patterns is to identify, on the fly, the FCPs within the streams as streams evolve over time.

To facilitate the search for FCPs, we divide each stream into overlapping *segments*. Each segment is a sequence of objects ordered by their timestamps, and the time span of a segment is the time interval between its first and last objects.

DEFINITION 5. *(***Segment***) For a given data stream s, a segment $G(o_1 o_2 \cdots o_m)$ is a subsequence of s that satisfies both of the following conditions:*
(1) $|t_i - t_j| \leq \xi, \forall\ o_i, o_j$ in G; and
(2) The time span of G must be maximal with respect to ξ. That is, that does not exist a subsequence of s, G', such that G' is a segment and G is a strict subsequence of G'.

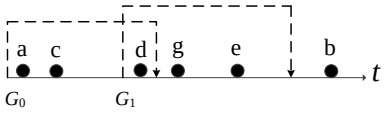


Figure 1: A segment. The temporal relationship between the objects is as follows: $t_d - t_a < \xi$, $t_g - t_a > \xi$, $t_g - t_d < \xi$, $t_g - t_c > \xi$, $t_e - t_d < \xi$, $t_b - t_d > \xi$.

Example 1. Figure 1 shows a stream of objects, where a is the first object of the segment G_0. Because $t_d - t_a < \xi$ and $t_g - t_a > \xi$, d is the last object of G_0. Note that the objects c, d, and g do not constitute a segment because $t_g - t_c > \xi$.

DEFINITION 6. *(***Prefix of the segment***) For a given segment G_i $(o_1 o_2 \cdots o_m)$, its prefix is a subsequence of G, $o_1 o_2 \cdots o_j$, where $1 \leq j \leq m$.*

The definition of segment dictates that any data stream can be uniquely partitioned. In addition, any CP must be covered by some segment(s). Therefore, the problem of mining FCPs accross multiple streams can be converted to finding the FCPs contained in the segments of the data streams.

3.2 A first attempt: the DIMine approach

Our first attemp to solve the problem of mining FCP is a simple method called DIMine. Following the discussion in Section 3.1, the basic approach is to divide the data streams into segments as the streams evolve, and then discover FCPs from those segments. To this end, we use an inverted index called DI-index to index all existing segments, and also maintain a list storing the information related to each segment including its starting and ending times as well as the ID of the data stream it belongs to. The DI-index can be implemented as a hash map, with each entry taking the form of $(o_i, \mathcal{G}_i)$, where o_i is an object and $\mathcal{G}_i$ stores the set of IDs of the segments containing o_i.

For an incoming segment G, DIMine finds the newly formed FCPs caused by G in the following steps using the anti-monotone Apriori heuristic based on the DI-Index.

(1) For each object o_i in G, it finds the entry $(o_i, \mathcal{G}_i)$ from the DI-Index and then sets $\mathcal{G}_i = \mathcal{G}_i \cup G$, It then verifies whether o_i is a FCP_1 (a FCP with only one object) by straightforward counting based on $\mathcal{G}_i$.

(2) It next iteratively generates the set of candidate FCPs with (k+1) objects from FCP_k ($k \geq 1$) caused by G, and picks those that conform to the FCP definition. Assuming that P_{k+1} is a candidate FCP with the set of objects $\{o_1, \cdots, o_{k+1}\}$ and $\mathcal{G} = \mathcal{G}_1 \cap \cdots \cap \mathcal{G}_{k+1}$, if the segments in the set $\mathcal{G}$ involve no less than θ data streams within the time interval τ , then P_{k+1} is a FCP.

As the DI-Index is a hash map, the DIMine approach can find the segments containing a given object o with constant time. Also, the use of the Apriori heuristic helps prune the search space, making the algorithm more efficient.

In addition to finding the FCPs, we also need to worry about the maitenance of the DI-Index, as streams constantly evolve over time, and some objects may become obsolete. The DIMine approach needs to scan all entries of the DI-index frequently to remove the identifiers of the obsolete segments because the expired data will not only cause false positive results but also increase the memory consumption. This task incurs non-trivial cost. Assuming that in a stable state, the number of segments being indexed is n and the average number of objects in a segment is p, then it takes $O(n \cdot p)$ time to detect all expired segments. As will be shown in the experimental results in Section 6, this is actually much more expensive than finding the FCPs.

4. THE SEG-TREE

Addressing the problems with DI-Mine, we present the CooMine approach that has better maintenance efficiency with an index that is more compact when significant overlapping exists between adjecent segments. We call this index the Seg-Tree.

4.1 Motivation

There are two critical issues to be addressed in mining FCPs across multiple data streams. First, as a large volume of segments may be generated at a varying rate, it is important to effectively index them with in a space-efficient manner for further processing. Second, for a new incoming segment, not every existing segment can form FCPs with it; thus it is necessary to quickly narrow down the search space. As such, we need an effective index structure that meets the following requirements: (1) good support for mining FCPs; (2) efficient handling of the insertion of new segments and deletion of obsolete segments; and (3) being frugal with its memory usage.

4.2 Overview of the Seg-tree structure

To satisfy the above requirements, we propose the Seg-Tree, a main-memory-based structure, to manage the valid segments. A segment G is valid, if and only if $t_{now} - t_G^f \leq \xi$, where t_{now} is the current time and t_G^f is the timestamp of the first object of G.

The Seg-tree index structure has three components: the *Seg-tree* itself and two auxiliary structures,*Hlist* and *Tlist*. Figure 2 shows a Seg-tree that indexes the segments in Figure 3.

The Seg-tree is a trie-like structure (but with notable differences) with its nodes corresponding to the objects. (We hereinafter use nodes and objects interchangeably when there is no ambiguity.) Edges exist between objects that appear adjacent to each other in some segment. Segments can be continuously inserted based on their prefixes, and obsolete segments can be removed as time progresses. The nodes are doubly-linked between child and parent. The details of insertion and deletion will be discussed later in Sections 4.4 and 4.5.

The two auxiliary structures are used to quickly locate specific nodes in the Seg-tree. We maintain a separate linked list of references to the nodes corresponding to the same object, and the head of each list is stored in Hlist. In Figure 2, we show the Hlist containing the heads for some of the lists.

Tlist, on the other hand, stores references to all the tail nodes of the Seg-tree, where a tail node refers to the last object of a segment. Each node in the Tlist is a reference to a tail node in the Seg-tree.

Once a tail node expires, we can use Tlist to quickly locate the corresponding segment in the Seg-tree and remove it. Figure 2 shows the links of tail nodes k and o.

4.3 Node structure

For a segment G_j, its last object is represented in the Seg-tree by a tail node tn_j and other objects by *ordinary nodes*. An ordinary node has four attributes $\langle Id$, *object*, *distance*, *count*, *reference*$\rangle$), where Id is the identifier of the node, *object* is the reference to the object represented by this node, *distance* represents the distance (number of edges) from this node to tn_j, *count* records the number of segments containing this node, and $reference$ points to the next node corresponding to the same object in the Seg-tree. For the new segment G_j, for each node, $distance$ can be easily calculated, *object* is known, $count$ is set to 1, and *reference* is initialized to null.

The tail node tn_j has five attributes $\langle Id$, *object*, *distance*, *count*, *reference*, $\mathcal{L}\rangle$. Here, *Id*, *distance*, *count*, and *reference* have the same meaning as that for ordinary nodes. $\mathcal{L}$ is a set of tuples $\{G_j, l_j\}$, where G_j is the segment with tn_j being the tail node and l_j is the length of segment G_j. Because tn_j may be the tail nodes of multiple segments, for each segment G_j, we have to record its length l_j.

With the help of Tlist, the Seg-tree structure has the following property. For the segment G_i, since tn_i records the length (l_i) of G_i, backtracking l_i-1 steps from tn_i to the root of the Seg-tree can uniquely determine the segment G_i, which is linear with respect to the length of the segment.

4.4 Constructing the Seg-tree

The Seg-tree is initialized as a root node (*null*). A new segment G_j can be inserted into the Seg-Tree using the following steps.

- We search for the longest matching prefix (pre_j) of G_j in the Seg-tree, the details of which will be discussed later.
- If pre_j exists, the remaining objects in G_j are appended to pre_j. For a segment, its longest matching prefix probably exists in different branches of the Seg-tree. In case this happens, the first being found will be picked. Otherwise, G_j will be directly added to the root.
- If pre_j exists, we need to update attribute values of the nodes in pre_j after G_j being inserted into the Seg-tree; the attribute values of other nodes remain unchanged.
- We insert the tail node tn_j into Tlist, and append each node of G_j to the respective linked list for the corresponding object.

Example 2. In Figure 2, the Seg-tree manages the segments in data streams s_1 and s_2 (as shown in Figure 3). Now we describe the process of inserting the segments from s_1. At the beginning, the Seg-Tree contains the root only. When the segment G_0 (*b, c, d*) appears, it is added to the root. As to the segment G_1 (*c, d, f, k*), since its prefix (*c, d*) exists in the tree, the objects *f, k* can be appended to the existing prefix. The segment G_2 (*h*, *m*, *n*) has no matching prefix, so it is inserted at the root. The segments G_3 (n, c, p, o) and G_4 (h, b, k, r, s, t) also have existing matching prefixes n and h separately in the Seg-tree, so they can be appended to their existing prefixes.

The Seg-tree is similar to *trie* [1], but they have significant differences. In a trie, the children of any node must have the common prefix. But in the Seg-tree, if a segment has a matching prefix in any branch of the Seg-tree (not necessarily starting from the root), then this segment can be appended to the existing prefix that are shared by multiple segments. Because there may be extensive overlapping between adjacent segments, this sharing can be quite effective in saving memory. For the trie, on the other hand, most of the overlapping segments cannot be compacted because they start with different objects.

Algorithm 1 *Searching prefix* Algorithm

Require:
The new segment G_j, the Seg-Tree;
Ensure: :
The Seg-Tree after G_j being inserted;
1: $\mathcal{H}$=Hlist(o_1); //$\mathcal{H}$ is the set of nodes that have the same identifier with the first node o_1 of G_j
2: pre_j=$\emptyset$;
3: **for** n_i:$\mathcal{H}$ **do**
4: pre_i=pre_i+n_i; count=1; n_j=n_i;
5: **while** $count < G_j.length$ **do**
6: flag=false; count++;
7: $\mathcal{N}_c$= child nodes of n_j;
8: **for** n_c:$\mathcal{N}_c$ **do**
9: **if** n_c==G_j.next() **then**
10: pre_i=pre_i+n_c; n_j=n_c; flag=true;
11: break;
12: **end if**
13: **end for**
14: **if** flag==false **then**
15: break;
16: **end if**
17: **end while**
18: **if** $pre_j.size < pre_i.size$ **then**
19: pre_j =pre_i;
20: **end if**
21: **end for**

(1) *Searching matching prefixes.* Searching the matching prefixes in the Seg-tree for an incoming segment is critical for its insertion. We can utilize the Hlist to accelerate this process. To search the matching prefix (pre_j) of a segment G_j ($o_1 o_2 \cdots o_{l_j}$), we first determine the set ($\mathcal{H}$) of nodes matching o_1 based on the Hlist. Next, for each node n_r ($n_r \in \mathcal{H}$), we shall traverse its children to find the node matching o_2. If the child n_c matches o_2, then $\{n_r, n_c\}$ is the current matching prefix of G_j, and we only need to scan the children of n_c to expand pre_j, while other children of n_r can be safely discarded because it is impossible for n_r to have two children that correspond to the same object. In this iterative fashion, pre_j can be determined. The details of searching prefixes are shown in Algorithm 1.

THEOREM 1. *For a given segment G_j with length l_j, the time complexity of searching for the longest matching prefix of G_j from the Seg-tree is $O(l_j)$.*

PROOF. Suppose that there are f nodes corresponding to the object o_1 (the first object of G_j). For any node n_r matching o_1, we at most traverse l_k levels of the subtree rooted at n_r. Hence, the dominating time cost of searching the longest existing prefix of G_j is $f \cdot l_k \cdot t_p$, where t_p is the unit time of traversing one level of the subtree. Since the parameters f and t_p are constants, the time complexity of searching the prefix of G_j is $O(l_j)$. □

(2) *Updating the attribute values of the nodes being inserted.* When G_j is inserted into the Seg-tree, for any node $n_i \in pre_j$, its attribute values have to be updated. Assuming that ($distance_i$, $count_i$ and $reference_i$) and ($distance'_i$, $count'_i$ and $reference'_i$)

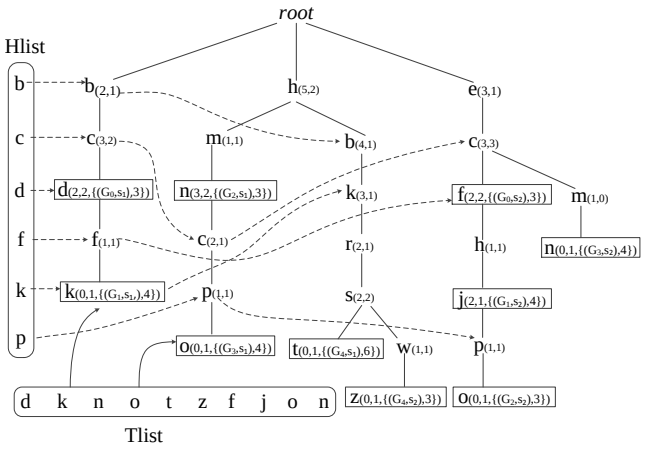


Figure 2: A sample Seg-tree. In this tree, the node in bold boxes are tail nodes and others are ordinary. The node annotation $h_{(5,2)}$ indicates that the distance between h and the farthest tail node t of G_4 is 5, and it appears in two segments. For the tail node $t_{(0,1,(((G_4,s_1),6))}$, the tuple $((G_4, s_1)$,6)) represents that t is the tail node of the segment G_4 in s_1, and the length of G_4 is 6.

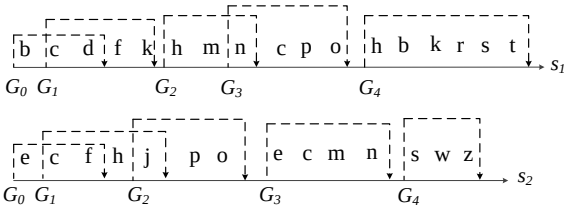


Figure 3: The segments managed by the Seg-tree shown in Figure 2

are the original and updated attribute values of n_i, the attribute values can be updated as follows.

- $distance'_i$=$max\{distance_i, distance'_i\}$. If n_i belongs to multiple segments, $distance'_i$ records the distance between n_i and the farthest tail node of the segments containing n_i.
- $count'_i$=$count'_i + 1$, which means that the number of segments that n_i appears in increases by one.
- $reference'_i$ is still null. If a node n'_i corresponding to the same object as n_i is inserted later, then $reference'_i$ will point to n'_i.
- If the tail node tn_j also belongs to pre_j, then aside from doing the above updates, we also need to add the tuple $\{G_j, l_j\}$ into $\mathcal{L}'_j$, that is, $\mathcal{L}'_j = \mathcal{L}'_j \cup \{G_j, l_j\}$.

Example 3 In Figure 2, before inserting the segment G_1 from stream s_1, the $distance$ and $count$ values of c and d in the Seg-tree are $(1, 1)$ and $(0, 1)$ separately. When G_1 is inserted, since c and d belong to the matching prefix of G_1, the values will be updated to (3,2) and (2,2) respectively.

4.5 Removing obsolete segments from the Seg-tree

As streams proceed, some segments in the Seg-tree will become obsolete. These obsolete segments will waste memory and have negative influence on mining FCPs, and they thus should be removed from the Seg-Tree in time, taking into the cost of removal. On one hand, the existing segments become obsolete frequently, so it is unwise to continuously monitor all segments and delete a segment as soon as it becomes obsolete. On the other hand, if the expired segments are retained in memory for too long, its impact on the memory consumption as well the search efficiency will become an issue.

As a good trade-off between effects of obsolete segments and the deletion cost, a Lazy Deletion (LD) strategy is introduced. In this strategy, we do not delete all expired segments at once, but only remove those that are relevant to the new incoming segment that needs to be processed. The deletion can also be triggered by the used memory exceeding the specified threshold, at which time we will scan the Tlist to find and remove all obsolete segments. Here, the relevant obsolete segments for a new segment can be determined by Algorithm 3 to be presented in Section 5.

Deleting an obsolete segment G_e involves two specific tasks: (1) decreasing the attribute value *count* of nodes in G_e and removing the nodes with *count* being zero from the Seg-tree; and (2) inserting the disconnected subtrees into the Seg-tree. Deleting G_e can possibly cause one or more subtrees being disconnected from the Seg-Tree, and these subtrees need to be added back into the Seg-Tree. For a disconnected subtree, we define its *single prefix* as the path from the root to the first node with more than one child. We treat the single prefix as a segment and insert it into the Seg-tree using the insertion method. The other objects of the disconnected subtree are appended to the prefix. In this way, any disconnected subtree can be inserted back into the Seg-tree.

The deletion cost can be reduced with the help of the Tlist. In the Tlist, we order the tail nodes by their arrival time, and thus we only need to determine the latest obsolete tail node, and then we can infer that all the tail nodes preceding this node are all obsolete. Hence, the obsolete segments can be quickly determined.

4.6 Comparison of Seg-tree and FP-tree

We are inspired by the FP-tree [9] in designing the Seg-tree to support the mining of FCPs. Compared with FP-tree, the Seg-Tree has the following advantages for the problem addressed in this paper.

(1) To construct the FP-tree, the objects in the transactions need to be sorted according to their counts. But in the Seg-tree, the objects in the segments do not need to be sorted, saving on the sorting cost. To be more specific, the count of each object changes frequently as data streams evolve, and therefore the order of objects in existing transactions needs to be adjusted constantly as required by the FP-tree. The FP-tree thus has to be updated or rebuilt frequently, causing inhibitive maintenance cost. As such, the FP-tree is not suitable for mining FCPs across data streams in real-time.

(2) In our problem, there may exist extensive overlapping between nearby segments. If we employ the FP-tree to index them without sorting, no compaction can be achieved according to the insertion rules defined in [9]. When the Seg-tree is used, however, many overlapping segments can be compacted tightly because they have common matching prefixes in the Seg-tree.

(3) The FP-tree deals with static datasets and does not specifically consider the effect of frequent updates, while in our case, the Seg-Tree has to be constantly updated and thus efficient maintenance cost is vital. By adopting the LD strategy to delete the obsolete data at different time granularities, we strive to achieve a balance between memory consumption and deletion cost.

5. THE COOMINE APPROACH

We propose the CooMine approach that can utilize the Seg-tree for mining FCPs. We first give an overview of this approach, and

then describe each component in more detail.

5.1 Overview of the CooMine approach

As discussed in Section 3, any FCP is covered by at least θ segments. We thus propose the CooMine approach to mine FCPs from the segments in each data stream. Since the streams are constantly changing with newly arrived objects, the task of mining FCPs is a continuous process, with actions triggered by the creation of each new segment as new objects arrive. For the new segment G_j, the *CooMine* approach consists of two components.

(1) *Searching for the largest common CPs*: If the new segment G_j can form new FCPs with existing segments, the FCPs must be covered by their common CPs. The largest common CP between any two segments refers to the largest set of common objects between them, so any common CP of two segments must be covered by their largest common CP. Therefore, we design an algorithm to find the largest common CPs between G_j and the existing segments to narrow down the search scope. Hereinafter, we use LCP to represent the longest common CP.

(2) *Mining FCPs from the LCPs*: Since the set of LCPs is finite and its size is usually small, mining FCPs boils down to the problem of finding the CPs that appear in at least θ data streams within time interval τ. This problem can be solved with an Apriori-style algorithm.

5.2 The SLCP algorithm for searching LCPs

A naive method to find the LCP is to compare the new segment with every existing segment. However, since only a few existing segments have common CPs with the new segment, comparisons with many segments are wasteful. Also, the cost will be huge when the Seg-tree is large.

To cut down the search cost, we design the *SLCP* algorithm to find the LCPs between the new segment G_j and existing segments. This algorithm first searches the *relevant segments* for each node of G_j, where the relevant segments are defined below in Definition 7. Next, it can deduce which common nodes with G_j each relevant segment has. The set of common nodes between each relevant segment and G_j is a LCP.

DEFINITION 7. *(*Relevant segment*): For any node n_i (except the root) in the Seg-tree, if the segment G_i contains n_i, then G_i is a relevant segment of n_i, and tn_i is a relevant tail node of n_i.*

Algorithm 2 describes the SLCP algorithm. First, for each node n_i of G_j, we find all nodes that have the same identifier with n_i based on Hlist (Line 4). Second, for any node n_j that corresponds to the same object as n_i, we determine its valid relevant segments (Line 5). Third, once the relevant segments of each node are determined, the LCPs are computed (Line 6-8).

In the SLCP algorithm, discovering the LCPs is a non-trivial problem. In the procedure of building the Seg-tree, any segment itself will be discarded after being inserted into the Seg-tree. Meanwhile, the ordinary nodes of the Seg-tree no longer record the information of their relevant segments for memory saving. In this case, for any ordinary node, we cannot directly determine the segments containing it. Therefore, we need to find an efficient way to determine the relevant segments for the specified nodes.

5.2.1 Searching relevant segments

We propose the *DistanceBound* method that can efficiently find relevant segments for the specified nodes. The *DistanceBound* method can prune the search scope by utilizing the distance between each node and its furthest relevant tail node as bound to accelerate searching relevant segments for the specified node. For the specified node

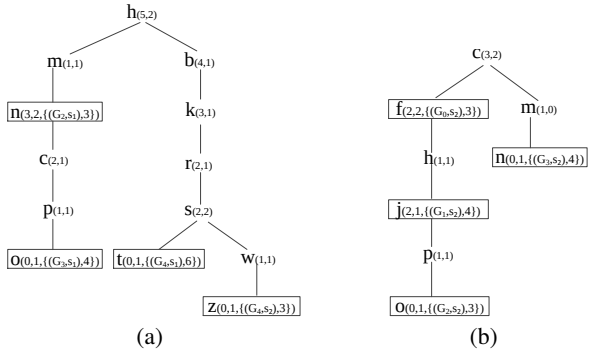


Figure 4: Two subtrees of the Seg-tree

n_j, this method visits n_j and guarantees that by visiting Dis_j levels of the tree rooted at n_j, it can find all relevant tail nodes for n_j according to Theorem 2. When this method visits any child n_c of n_j, only $min\{Dis_c, Dis_j - 1\}$ levels of the tree rooted at n_c need to be traversed. In this way, the search algorithm can quickly converge and the search space can be reduced.

Algorithm 3 describes the process of searching relevant segments for n_j, which consists of four steps:

Step 1: Build a queue *Q* and enqueue the pair $\langle n_j, stepcount\rangle$ into *Q*, where *stepcount*=Dis_j. (Lines 1-4)

Step 2: Get the first pair $\langle n_x, stepcount\rangle$ from *Q*. (Line 6)

Step 3: If n_x is a relevant tail node of n_j, insert n_x into the set $\mathcal{R}$. Otherwise, for each child n_c of n_x, set *stepcount* as $min\{Dis_c$, *stepcount*-1$\}$ and enqueue $\langle n_c, stepcount\rangle$ into *Q* if $stepcount \neq 0$. (Lines 7-15)

Step 4: If *Q* is not empty, go to step (2); otherwise, return the set $\mathcal{R}$ and the search ends. (Line 5)

In Step 3 , for the tail node n_x, if we assume that its corresponding segment is G_x with length l_x and the distance from n_j to n_x is D_{jx}, then G_x is a relevant segment of n_j if l_x is no less than D_{jx} and G_x is valid.

THEOREM 2. *For any node n_j in the Seg-tree, all relevant tail nodes of n_j can be found by visiting at most Dis_j levels of the tree rooted at n_j.*

PROOF. Suppose that there is a tail node tn_j that cannot be discovered even if after taking (Dis_j-1) steps in each branch of n_j. Since n_j and tn_j belong to the same segment, tn_j and n_j must be in the same branch. Because tn_j cannot be found by taking Dis_j steps from n_j in this branch, the distance from n_j to tn_i must be greater than Dis_j. However, Dis_j is the distance between n_i and the farthest relevant tail node. Contradiction. □

The *DistanceBound* algorithm can effectively prune the search space based on the attribute $distance$ of nodes. Figure 4 shows two subtrees of the Seg-tree in Figure 2. In Figure 4(a), the node h has attribute Dis_h as 5; the *DistanceBound* algorithm thus probably needs to take 5 steps in its each branch to find the relevant tail nodes for h. However, when it visits the node m and finds that Dis_m is equal to 1, it then only needs to take one step in the branch of m to search the relevant tail node n. The nodes after n do not need to be scanned. As to the node c in Figure 4(b), since Dis_c is 3, we only need to traverse its left branch by 3 steps to search for the relevant tail node j, and nodes after j can be ignored.

5.2.2 Obtaining the LCPs

For each node n_i in G_i, all relevant segments of n_i can be determined by the *DistanceBound* algorithm. If we employ a hash table

Algorithm 2 Searching for LCPs (*SLCP*)

Require:
The new segment G_j, the Seg-Tree;
Ensure: :
The LCPs between G_j and existing segments;
1: $\mathcal{H}_i$=null; // a set of nodes have the same identifier with n_i
2: The *Map*$< G_i, P_i >$ map=null; // P_i is the LCP between G_j and G_i
3: **for** n_i:G_j **do**
4: H_i=Hlist(n_i);
5: $\mathcal{R}_i$=*DistanceBound* (H_i); // $\mathcal{R}_i$ includes segments covering n_i
6: **for** G_i:$\mathcal{R}_i$ **do**
7: map.add($< G_i, n_i >$);
8: **end for**
9: **end for**
10: Output map;

Algorithm 3 *DistanceBound* Algorithm

Require:
H_i;
Ensure: :
The relevant segments of n_i;
1: *stepcount*=0, $\mathcal{R}$=null, *Queue*=null;
2: **for** n_k:H_i **do**
3: *stepcount*=Dis_k and generate the pair $\langle n_k$, *stepcount*$\rangle$;
4: *Queue*.put($\langle n_k$, *stepcount*$\rangle$);
5: **while** *Queue*$\neq$ empty **do**
6: n_f=*Queue*.getfirst();
7: **if** n_f is a tail node and G_f covers n_k and G_f is valid **then**
8: $\mathcal{R}_k$.add(G_f)
9: **end if**
10: **for** n_c:children of n_f **do**
11: *stepcount*=min$\{Dis_f, (stepcount - 1)\}$;
12: **if** *stepcount* $\neq 0$ **then**
13: *Queue*.put($\langle n_c$, *stepcount*$\rangle$);
14: **end if**
15: **end for**
16: **end while**
17: $\mathcal{R}$=$\mathcal{R} \cup \mathcal{R}_k$;
18: **end for**
19: return $\mathcal{R}$;

to store the relevant segments with the key being the ID of each relevant segment and the value being a list of nodes common to this segment and G_i, then the largest set of common nodes between each relevant segment and G_i can be immediately determined, and the largest set of common nodes is a LCP between G_i and the corresponding relevant segment.

5.3 Mining FCPs from the LCPs

5.3.1 Mining FCPs with the Apriori heuristic

For the new incoming segment G_j, if it can form new FCPs with existing segments, then these FCPs must be covered by their LCPs. Therefore, we can mine the FCPs from the these LCPs with the anti-monotone Apriori heuristic. The basic idea is to iteratively generate the set of FCPs with $(l + 1)$ objects based on the set of FCPs with l ($l \geq 1$) objects. Theorem 3 guarantees the correctness of the CooMine algorithm.

Table 1: Summary of common CPs

LCPs	segments
$\{m, n, \}$	$\{G_2, s_1\}$
$\{n, p, o\}$	$\{G_3, s_1\}$
$\{p, o\}$	$\{G_2, s_2\}$
$\{m, n\}$	$\{G_3, s_2\}$
$\{n\}$	$\{G_4, s_2\}$

Algorithm 4 CooMine Algorithm

Require:
The new segment G_i, The Seg-Tree;
Ensure: :
The FCPs of length k formed by G_i;
1: *LCPtable* = SLCP(G_i, Seg-tree);
2: l=1;
3: **while** $l < k$ **do**
4: **if** $l = 1$ **then**
5: Obtaining FCPs of length l based on the *LCPtable*;
6: **else**
7: Generating candidate FCPs with length l based on the FCPs of length $l - 1$;
8: Detecting each candidate FCP based on the *LCPtable* and discover the FCPs of length l;
9: **end if**
10: **end while**
11: Output the FCPs of length k.

THEOREM 3. *If a set of objects $\mathcal{O}_n$ is a FCP, then any of its subset $\mathcal{O}_n^{'}$ must also be a FCP.*

Since LCPs are found from valid segments, the time span of all obtained LCPs must be no greater than the threshold τ. According to the Definitions 2 and 3, if a CP occurs in more than θ data streams within the time interval τ, then this CP must be a FCP. Hence, the CooMine algorithm only needs to consider the number of data streams that the LCPs appear in. Because all LCPs between T_k and existing segments have been found by the *SLCP* algorithm, the CooMine algorithm (Algorithm 4) only needs to mine all FCPs from the LCPs.

Example 4. Assuming that G_0 ($mnpo$) is a new segment in data stream s_3 and the parameters k and θ (cf. Definition 3) are 2 and 3 separately. Table 1 shows the LCPs between G_0 and existing segments in Figure 2. Based on Table 1, CooMine can find the FCPs of size 1 ($\{m\}$, $\{n\}$, $\{o\}$, $\{p\}$), and then deduce the FCPs of size 2 ($\{m, n\}$, $\{p, o\}$). Since only one LCP has three objects, we assert that there does not exist a FCP of size 3.

5.4 Advantages of our approach

CooMine has the following advantages.

(1) The use of the Seg-tree to index the valid segments can help save on memory consumption. The segments are inserted into the Seg-tree based on the prefix rule, and many common objects in multiple segments can be compacted to save memory. In addition, the ordinary nodes in the Seg-tree do not record the segments that they belong to and only the tail nodes maintain this information. Since ordinary nodes make up the largest part of the Seg-tree and each ordinary node always takes place in multiple segments, omitting the segments information from ordinary nodes can save much on memory consumption.

(2) The maintenance cost of the Seg-tree is low. A new segment can be inserted into the Seg-tree with a small cost because its

Table 2: The parameters used in the experiments

parameters	meaning
ξ	the time interval in Definition 2
k	the number of objects in one FCP
τ	the time interval in Definition 3
θ	the number of streams in Definition 3
D_s	the scale of data

matching prefix can be quickly determined; the obsolete segments can be immediately determined with the help of the Tlist, and the LD strategy can remove the obsolete data at different granularities to reduce the deletion cost.

(3) CooMine first searches the LCPs covering all FCPs between each new segment and existing ones; this step can effectively narrow down the search scope for mining FCPs.

6. EXPERIMENTS

We conduct experiments to evaluate the performance of DIMine and CooMine methods, and compare them with the MatrixMine algorithm that is introduced as a baseline method. The parameters involved in experiments are illustrated in Table 2.

Specifically, we first evaluate the index structures employed in three methods with respect to memory consumption and maintenance cost, and then compare the performance of the three methods for computing FCPs. Finally, we test the influence of varying parameters on the CooMine algorithm in the following aspects: the time of discovering FCPs, the sustainable workload, and the number of FCPs.

In our experiments, we implement three methods (CooMine, DIMine, and MatrixMine) using Java and the indexes of three methods all employ the standard Java Collection classes as the storage structure to record the segments information. To make the results more accurate, every evaluation is repeated ten times and the average values are recorded as the final results.

6.1 Experimental setup

The experiments are conducted on a Dell OPTIPLEX 990, a PC with a 3.1GHz Intel i5-2400 processor and 8GB RAM. Two real datasets are used. One dataset is a traffic records dataset (TR dataset) of Jinan city in China. The TR dataset contains all VPRs (vehicle passing records) of each monitoring camera in Jinan on May 1, 2013. For each monitoring camera, we simulate its passing records as a data stream, then the TR dataset can be viewed as multiple data streams. For the TR dataset, D_s represents the number of VPRs, and each vehicle (identified by its license plate) is an object.

The other dataset (the Twitter dataset) is provided by Twitter for academic research purposes[1]. In Twitter dataset, each word is considered an object and a tweet corresponds to a segment. In this case, all of the objects in a segment appear at once, as a tweet as a collection of words is posted as a whole. The tweets by the same user constitute one data stream. D_s represents the number of all segments (tweets).

6.2 The baseline method

Since none of the existing methods is applicable to the problem of finding FCPs in real-time either because they address a different problem or because their cost is apparently too expensive when applied to this problem, we introduce the MatrixMine algorithm as the baseline method, and compare it with our proposed methods w.r.t. memory consumption, maintenance cost, mining time, and total cost.

MatrixMine also divides the data stream into segments and then mines FCPs from the segments. The information of all segments are maintained in an independent list and we can obtain the information of each segment based on its identifier from this list. MatrixMine maintains a matrix M that keeps track of each pair of co-occurring objects. If we assuming that there are n distinct objects ($\{o_1, \cdots, o_n\}$) in a particular application, then M is a $n \times n$ matrix, where each element $c_{i,j}$ of M corresponds to the pair $p_{i,j}$ that consists of objects o_i and o_j ($1 \leq i, j \leq n$). Because the pair $p_{i,j}$ probably occurs in different segments, the corresponding element $c_{i,j}$ has to maintain a set $\mathcal{I}_{i,j}$ that contains multiple tuples and each tuple is represented as $\langle G_j, s_h \rangle$, where G_j is the identifier of the segment that $p_{i,j}$ belongs to, and s_h is the data stream containing G_j.

For any pair $p_{i,j}$ of the new segment G_j, MatrixMine can determine whether $p_{i,j}$ is a FCP based on $\mathcal{I}_{i,j}$ because it records the identifiers of all segments covering the pair $p_{i,j}$. Once the FCPs with two objects are obtained, MatrixMine can iteratively generate the FCPs with more objects employing the Apriori heuristic.

The maintenance of the matrix M includes the insertion of newly generated pairs and the deletion of obsolete pairs. When a new pair $p_{i,j}$ appears, it must be inserted into M right away. To reduce the deletion cost of the Matrix structure, we also adopt the Lazy Deletion strategy to remove the obsolete data from the matrix.

6.3 The performance of index structures

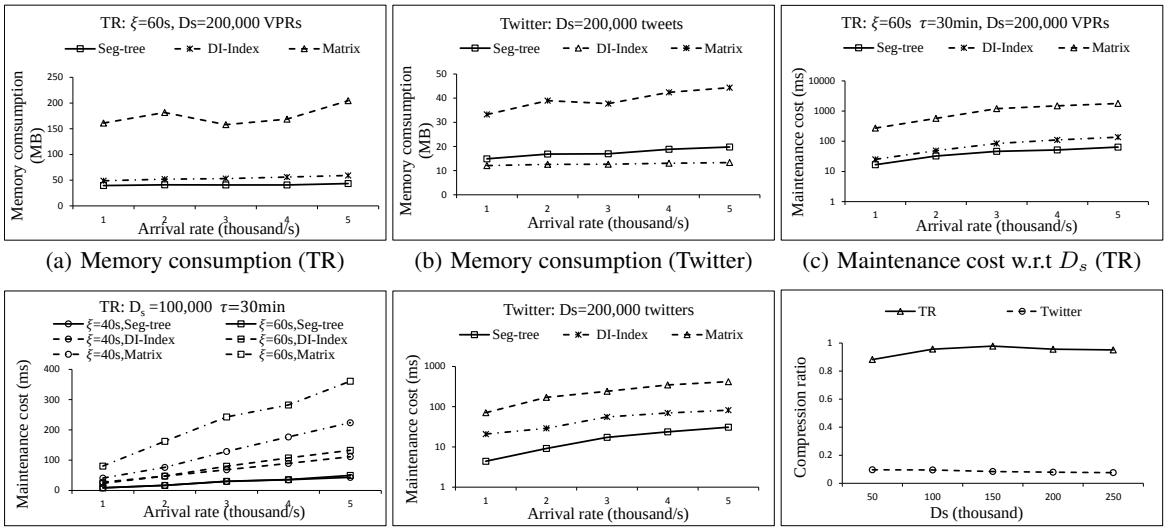
We compare the Seg-tree, the DI-Index, and the *Matrix* with respect to memory consumption and the maintenance cost, and the results are shown in Figure 5.

Memory consumption. Figure 5(a) shows the memory consumption of index structures on the TR dataset. Here, we test the memory consumption for processing the incoming data within one second with varying arrival rates when D_s is fixed, where D_s represents the volume of data that has been processed. The experimental results demonstrate that the memory consumed by Seg-tree is about 80% of that consumed by DI-Index and only 25% of that by *Matrix* because of the overlap between segments makes it possible to compact them tightly in the Seg-tree. *Matrix* consumes much more memory than Seg-tree and DI-Index, and the reason is the *Matrix* has to maintain a large volume of elements.

Figure 5(b) shows the result on the Twitter dataset, where the Seg-tree consumes slightly more memory than DI-Index. The reason is that most adjacent segments (tweets) do not overlap, and the Seg-tree cannot compact them as well as for TR. However, the Seg-tree still consumes much less memory than the *Matrix*.

Maintenance cost. Figure 5(c), 5(d) and 5(e) show the maintenance time of processing the data within one second with different arrival rates for the two datasets. In Figure 5(c), we evaluate the maintenance cost of the Seg-tree, DI-Index, and *Matrix* with the fixed values of the parameters ξ, τ, and D_s . The results demonstrate that the maintenance cost of the Seg-tree is much smaller than that of DI-Index and *Matrix*, while the *Matrix* has the largest maintenance cost. Specifically, the maintenance cost of the Seg-tree is approximately 50% of that consumed by DI-Index, while the maintenance cost of *Matrix* is almost 20 times as big as that of Seg-tree. In the Seg-tree, we can immediately determine the expired segments based on the Tlist and remove them, which can save much maintenance time. But in DI-Index and *Matrix*, we have to detect each element to remove the obsolete data, increasing the maintenance cost. Since the number of elements in *Matrix* is greater than that of elements in DI-Index, maintaining *Matrix* is more expensive than maintaining the DI-Index.

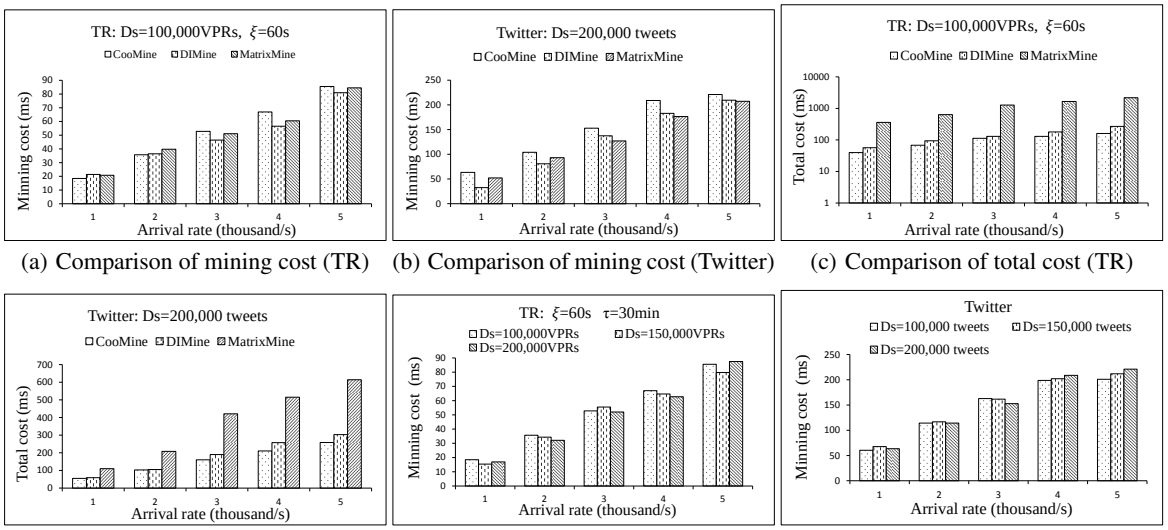
[1]Tweets2011. http://trec.nist.gov/data/tweets/



(a) Memory consumption (TR) (b) Memory consumption (Twitter) (c) Maintenance cost w.r.t D_s (TR)

(d) Maintenance cost w.r.t ξ (TR) (e) Maintenance cost w.r.t D_s (Twitter) (f) Compression ratio w.r.t ξ

Figure 5: Evaluation of index structures



(a) Comparison of mining cost (TR) (b) Comparison of mining cost (Twitter) (c) Comparison of total cost (TR)

(d) Comparison of total cost (Twitter) (e) Mining cost w.r.t D_s (f) Mining cost w.r.t D_s

Figure 6: Performance of mining algorithms

Figure 5(d) shows that the parameter ξ has little influence on the maintenance time of the Seg-tree, but it affects the maintenance cost of DI-Index and *Matrix* more significantly. The reason is that the sizes of segments will become larger when ξ takes greater values, and the larger segments have very slight influence on the maintenance time of the Seg-tree because each segment can be inserted and removed in its entirety regardless of its length. However, the larger segments contain more objects, which can produce more elements for DI-Index and *Matrix*, and they thus need much more time to maintain the larger segments.

Fig. 5(e) shows the maintenance cost of the three index structures for the Twitter dataset. Similar to the case of the TR dataset, the Seg-tree has the less maintenance cost than DI-Index and *Matrix*.

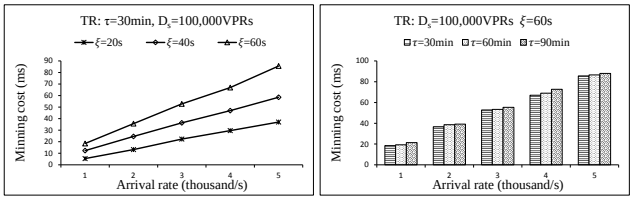
According to the aforementioned experimental results, we conclude that the Seg-tree and the DI-Index outperform the *Matrix* w.r.t. memory consumption and maintenance cost. In most cases, the performance of Seg-tree is better than that of the DI-Index structure except the memory consumption for processing the Twitter dataset.

Compression ratio. To help us better understand the memory consumption, assuming that the original data to be indexed is d_1 and the real data stored in the Seg-tree is d_2, the compression ratio of the Seg-Tree is defined as $(d_1\text{-}d_2)/d_1$. In Fig. 5(f), the compression ratio based on TR dataset is very high, which means the Seg-tree can compact the overlapping segments very well. As to the Twitter dataset, the compression ratio becomes very low as there are less overlapping between segments. Therefore, it would be helpful to look at the degree of overlapping between segments before deciding on the index structure for mining FCP.

6.4 Performance comparison of the mining algorithms

Comparison of mining cost. First, we compare the mining performance of CooMine, DIMine, and MatrixMine algorithms on the two datasets in Fig. 6(a) and 6(b). For the TR dataset, the mining time of the three algorithms is almost identical. However, the mining cost of the CooMine algorithm is larger than that of the DIMine and MatrixMine methods on the Twitter dataset. This is again due to the lower degree of overlapping between segments, which renders some optimizations of the CooMine algorithm not applicable in reduing the mining cost.

Comparison of total cost. In this group of experiments, we evaluate the total cost of the three algorithms for processing the data within one second with varying arrival rates based on two datasets, and the results are shown in Fig. 6(c) and 6(d). For the set of data being processed, the total cost includes the time of inserting this set of data into the index structure and removing the relevant obsolete data from the index structure, as well as the time of mining FCPs from this set of data. The results show that CooMine performs best, and both CooMine and DIMine outperform the MatrixMine approach dramatically regardless of the dataset.



(a) Mining cost w.r.t ξ (b) Mining cost w.r.t τ

Figure 7: FCPs w.r.t D_s

6.5 Evaluation of the CooMine algorithm

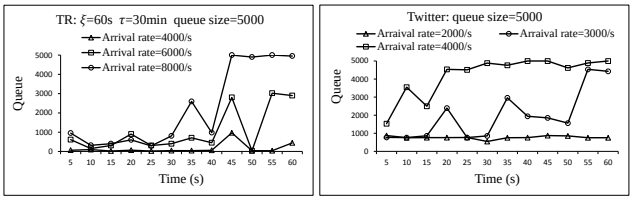
Since the CooMine algorithm is better than the other two methods with respect to the total cost, we only evaluate the influence of varying parameters on its performance.

Mining cost w.r.t. D_s. We test the influence of D_s on the performance of CooMine for mining FCPs on two datasets in Fig. 6(e) and 6(f). The results show that D_s has no evident effect on the mining cost because the CooMine algorithm only needs to search a small portion of the data to find FCPs.

Mining cost w.r.t. ξ and τ. Fig. 7(a) shows that the mining time is also affected by the parameter ξ. The larger value of ξ will give rise to longer segments, and the segments with larger sizes will cause more LCPs between each new segment and existing ones, which can increase the mining time. Fig. 7(b) demonstrates that the parameter τ has little impact on the mining cost. This is because although the larger value of τ can cause more valid segments, the CooMine algorithm can still efficiently narrow down the search scope.

Maximum sustainable workload. To evaluate the maximum sustainable workload of the CooMine algorithm, we introduce a buffer queue with 5000 storage units to cache the incoming data, and the CooMine algorithm will fetch the data from this queue. In this case, the usage rate of the buffer queue reflects the processing capacity of the CooMine algorithm.

In Fig. 8(a), we evaluate the buffer queue usage at different time points based on the TR dataset. When the arrival rate reaches 8000 VPRs per second, the maximum usage of the buffer queue is 5000. Therefore, the maximum sustainable workload of the CooMine algorithm based on the TR dataset is 8000 VPRs per second. Fig. 8(b) shows that the maximum sustainable workload based on the Twitter dataset is about 4000 tweets per second.

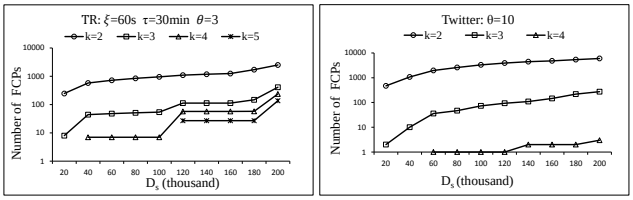


(a) Maximum sustainable workload (TR) (b) Maximum sustainable workload (Twitter)

Figure 8: FCPs w.r.t D_s

6.6 Effect of parameters on the number of FCPs

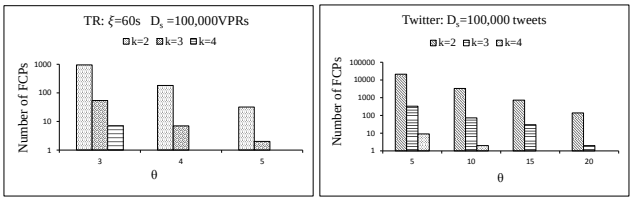
We now evaluate the influence of the parameters D_s and θ on the number of FCPs generated for the two datasets.



(a) Results on TR (b) Results on Twitter

Figure 9: The number of FCPs discovered w.r.t D_s

Fig. 9(a) and 9(b) show that the number of FCPs increases with more data being mined for the TR and Twitter datasets respectively. For a fixed volume of data, there exist more FCPs with smaller sizes (k).



(a) Results on TR (b) Results on Twitter

Figure 10: The number of FCPs discovered w.r.t θ

We also test the effect of the parameter θ on the number of FCPs in Figure 10(a) and 10(b). When the value of θ becomes larger, the number of FCPs drops sharply, which coincides with our intuition that the higher the threshold, the less FCPs will appear.

Finally, we analyze the FCPs from the Twitter dataset with θ equal to 60 and illustrate some typical hot events that the FCPs imply in Table 3 and Table 4, demonstrating that mining FCP is indeed useful in such applications.

6.7 Discussion

We have compared the CooMine, DIMine, and MatrixMine approaches with respect to the memory consumption, the maintenance cost, the mining cost, and the total cost on two datasets. The results demonstrate that CooMine and DI-Mine approaches outperform the MatrixMine method significantly for mining FCPs. Between the CooMine and DI-Mine approaches, we find that the CooMine approach based on the Seg-tree is better suited to process

Table 3: Typical FCPs from the Twitter dataset

FCPs	The number of streams	Hot event
super bowl	1378	event1
green bay packers	213	
win steelers	226	
jack lalanne dies	139	event2
airport killed	111	event3
airport domodedovo	101	
union state address	409	event4
obama sotu	456	
science fair	63	
health care	261	

Table 4: Hot events

Event	Meaning
event1	*Green Bay Packers and Pittsburgh Steelers played the Super Bowl on February 6, 2011.*
event2	*Jack lalanne, the American exerciser, and nutritional expert, died on January 23, 2011.*
event3	*The Domodedovo International Airport bombing on January 24, 2011.*
event4	*Barack Obama presented the 2011 State of the Union Address on January 25, 2011.*

data streams that have much overlap between segments, while the DIMine approach is more suitable for handling data streams without overlapping segments.

7. CONCLUSION

Mining frequent co-occurrence patterns (FCPs) across multiple data streams is essential to many real-world applications, but this problem has not been addressed by exiting works. In this paper, we design the DIMine and CooMine approaches to mine FCPs using only one pass over the data streams. In both approaches, we first divide each data stream into overlapping segments, and then mine the FCPs within those segments. The DIMine approach uses an inverted index (DI-Index) to maintain the valid segments in main memory and adopts an Apriori-style heuristic to iteratively discover FCPs based on this index. In the CooMine approach, we construct the Seg-tree, a memory-based index structure, to compactly index all valid segments. Based on the Seg-tree, the CooMine approach first finds the largest common CPs between each new segment and the existing ones to narrow down the search scope, and then discover the FCPs from the obtained common CPs. Finally, we introduce a baseline method and conduct extensive experiments to compare our proposed approaches with this baseline method. The experimental results demonstrate that our proposed approaches outperform the baseline method by a significant margin.

For future work, we would like to study how to extend the proposed approaches to a distributed environment to handle greater scales of data streams, when a single machine is no longer capable of managing the large volumes of data and computation.

8. ACKNOWLEDGMENT

This work was supported in part by the 973 Program (2015CB352 500), the National Natural Science Foundation of China Grant (6127 2092), the Shandong Provincial Natural Science Foundation Grant (ZR2012 FZ004), the Science and Technology Development Program of Shandong Province (2014GGE27178), the Independent Innovation Foun-dation of Shandong University (2012ZD012), the Taishan Scholars Program, and NSERC Discovery Grants.

9. REFERENCES

[1] Trie. http://en.wikipedia.org/wiki/Trie.

[2] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[3] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[4] J. H. Chang and W. S. Lee. Finding recent frequent itemsets adaptively over online data streams. In *SIGKDD*, pages 487–492, 2003.

[5] Z. Chen, H. T. Shen, and X. Zhou. Discovering popular routes from trajectories. In *ICDE*, pages 900–911, 2011.

[6] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu. Mining frequent patterns in data streams at multiple time granularities. *Next generation data mining*, pages 191–212, 2003.

[7] F. Giannotti, M. Nanni, F. Pinelli, and D. Pedreschi. Trajectory pattern mining. In *SIGKDD*, pages 330–339, 2007.

[8] J. Guo, P. Zhang, J. Tan, and L. Guo. Mining frequent patterns across multiple data streams. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 2325–2328, 2011.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

[10] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *TODS*, pages 51–55, 2003.

[11] J.-G. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *ICDE*, pages 140–149, 2008.

[12] C.-S. Leung and Q. I. Khan. Dstree: a tree structure for the mining of frequent sets from data streams. In *ICDM*, pages 928–932, 2006.

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD*, pages 39–44, 2005.

[14] Q. Li, Y. Zheng, X. Xie, Y. Chen, W. Liu, and W.-Y. Ma. Mining user similarity based on location history. In *SIGSPATIAL*, pages 34–45, 2008.

[15] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[16] A. Monreale, F. Pinelli, R. Trasarti, and F. Giannotti. Wherenext: a location predictor on trajectory pattern mining. In *SIGKDD*, pages 637–646, 2009.

[17] B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *ICDE*, pages 179–188, 2008.

[18] R. T. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD*, pages 13–24, 1998.

[19] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, pages 343–354, 1998.

[20] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong, and Y.-K. Lee. Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences*, 179(5):559–583, 2009.

[21] J. X. Yu, Z. Chong, H. Lu, and A. Zhou. False positive or false negative: mining frequent itemsets from high speed transactional data streams. In *VLDB*, pages 204–215, 2004.