SWIFT: Mining Representative Patterns from Large Event Streams

Yizhou Yan [1*], Lei Cao [2*], Samuel Madden [2], Elke A. Rundensteiner[1]
[1]*Worcester Polytechnic Institute Worcester, MA, USA*
[2]*Massachusetts Institute of Technology, Cambridge, MA USA*
(yyan2,rundenst)@cs.wpi.edu, (lcao,madden)@csail.mit.edu

ABSTRACT

Event streams generated by smart devices common in modern Internet of Things applications must be continuously mined to monitor the behavior of the underlying system. In this work, we propose a stream pattern mining system for supporting online IoT applications. First, to solve the pattern explosion problem of existing stream pattern mining strategies, we now design pattern semantics that continuously produce a compact set of patterns that maximumly compresses the dynamic data streams, called MDL-based Representative Patterns (MRP). We then design a one-pass SWIFT approach that continuously mines the up-to-date MRP pattern set for each stream window upon the arrival or expiration of individual events. We show that SWIFT is guaranteed to select the update operation for each individual incoming event that leads to the most compact encoding of the sequence in the current window. We further enhance SWIFT to support batch updates, called B-SWIFT. B-SWIFT adopts a *lazy update* strategy that guarantees that only the minimal number of operations are conducted to process an incoming event batch for MRP pattern mining. Evaluation by our industry lighting lab collaborator demonstrates that SWIFT successfully solves their use cases and finds more representative patterns than the alternative approaches adapting the state-of-the-art static representative pattern mining methods. Our experimental study confirms that SWIFT outperforms the best existing method up to 50% in the compactness of produced pattern encodings, while providing a 4 orders of magnitude speedup.

PVLDB Reference Format:
Yizhou Yan, Lei Cao, Samuel Madden, Elke A. Rundensteiner. SWIFT: Mining Representative Patterns from Large Event Streams. *PVLDB*, 12(3): 265-277, 2018.
DOI: https://doi.org/10.14778/3291264.3291271

1. INTRODUCTION

Motivation. Over the past decade, smart phones, tablets, sensors and other Internet of Things (IoT) devices have become ubiquitous. These devices continuously generate large volumes of data, typically in the form of streams composed of discrete events. Examples include control signals from Internet-connected devices like lights and thermostats as well as log files from smartphones that record the behavior of sensor-based applications over time. With the increasing prevalence of such event streams, discovering the frequent sequential patterns hidden in the data [1, 13] is more critical than ever. Our work in this space is motivated by several real-world applications we have built as described below.

Example 1. As a part of a collaboration with a major industrial lighting Lab, we have been tasked to find the frequent patterns that characterize typical lighting control messages exchanged between their smart lighting devices and servers in the cloud, to facilitate understanding whether the devices are performing as expected based on their configuration or are in an abnormal state. For example, each lighting device is expected to routinely handshake with its server. This handshake behavior regularly produces a synchronization message (denoted as S) followed by a location report message (denoted as L), that can be modeled by a frequent pattern $P = \langle S, L\rangle$. Missing this frequent pattern in some time interval indicates the devices might be faulty during this period.

Example 2. In a sensor-based mobile application that records data from users as they drive, the *frequent* pattern $P = \langle \mathit{StartTrip}(A), \mathit{RecordTrip}(B), \mathit{ReportLoc}(C), \mathit{Terminate}(D)\rangle$ captured from continuously generated log messages represents a typical behavior of the system, namely, after the user starts a trip, the system continues reporting, and then terminates when the trip finishes. The system is in a healthy condition if this frequent pattern is consistently observed over time. Otherwise, the system may be functioning abnormally.

Example 3. In a hospital infection control system [23] that continuously tracks healthcare workers (HCWs) for hygiene compliance (for example sanitizing hands and wearing masks), sensors are installed at doorways, beds, and disinfectant dispensers. In such healthcare settings, a HCW is required to wash her hands before approaching a patient. This behavior can be modeled as a *frequent* pattern $P = \langle \mathit{HandWash}, \mathit{Enter}, \mathit{ApproachPatient}\rangle$. Consistent observation of this frequent pattern indicates that the HCW performs hygiene behavior appropriately, whereas violations of it indicate a lack of proper hygiene protocol during a particular shift of the HCW.

State-of-the-Art and Limitations. Existing stream pattern mining techniques [5, 6], which detect sequential patterns from streams using traditional mining semantics [2, 24], suffer from a *pattern explosion* problem [2, 24]. That is, they tend to produce a huge and highly redundant collection of frequent patterns that are difficult to understand. For example, in our industrial lighting project, we found that using *SeqStream* [5] with closed frequent pattern semantics [24] produces 1,082 patterns under a standard configuration of parameters, even though there are only 13 distinct message types in

[*] Equal contribution

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291271

total and fewer than 1,000 events per window in the lighting control message stream. Among these patterns, 477 patterns contain the same 3 event types "synchronization (S), location report (L), and alarm (A)" in the same order. As such, most of these patterns are not informative. For example, all 6 length-3 patterns which could be formed by the three event types S, L, A are identified as frequent. However, only $\langle SLA\rangle$ is an actual sequence of alarms that should occur in the system.

In short, existing techniques are neither effective nor efficient at supporting online stream applications. These techniques have quadratic or worse CPU computation complexity, resulting in prohibitively large response time for large data sets. Worse yet, human operators are overwhelmed when having to continuously examining such large pattern sets over time.

Challenges. In this work we thus target on designing an effective yet efficient stream pattern mining system amenable for online IoT applications. Designing such a system is challenging, because the need to solve the pattern explosion problem contradicts the stringent response time requirement for the real time monitoring of the system status and prompt action when abnormal phenomena occur.

More specifically, to solve the pattern explosion problem, the produced set of patterns has to be very compact. On the other hand, however, this compact set of patterns has to be rich enough to capture the typical behaviors of the system. Otherwise missing the representative patterns might lead to a loss of huge fortune or even human life. Intuitively, the pattern explosion problem could be solved by using some post-processing strategy that carefully selects a compact set of patterns from all possible patterns. For example, the minimum description length (MDL) principle [20], widely used in text compression, could be used as a mechanism to select a reduced set of patterns [14, 15, 22]. These MDL-decided patterns, if used as a dictionary, have the potential to maximally compress the data into a compact pattern encoding. Therefore, they are considered to effectively represent the whole sequence. However, the problem of selecting a set of patterns based on MDL is NP-hard [21, 28]. Therefore, this approach inevitably adds substantial extra costs to the already expensive pattern mining task, while in online IoT applications, the approach has to be lightweight to meet the real time response requirement.

Proposed Approach. In this work, we propose the first stream pattern mining system that continuously finds the representative patterns in sliding window event streams using the MDL principle. It uses a new algorithm we call SWIFT[1]. Key contributions include:

- We present *MDL-based Representative Patterns (MRP)*, which are the first stream pattern mining semantics that leverage the MDL principle to continuously model a compact set of representative patterns in sliding window stream.
- We describe a one-pass algorithm, SWIFT, that efficiently discovers a set of representative patterns that match the MRP semantics using an incremental strategy. SWIFT, as a one-pass solution that touches each new arrival event only once, is well-suited to IoT streaming data. This allows SWIFT to scale to large volume event streams – processing up to 100,000 events per second. We prove that SWIFT is guaranteed to select the update operation for each individual incoming event that most reduces the description length of the sequences in the current window.
- We also design an extended SWIFT approach, called B-SWIFT, for supporting batch updates. The key idea is that instead of immediately updating the pattern set whenever an individual event arrives, B-SWIFT adopts a *lazy update* strategy that refreshes the MRP pattern set only once per window. We show that B-SWIFT conducts update operations guaranteed to cause a change in the MRP pattern set representing the current window.
- Evaluation by our industry lighting lab collaborators shows that it successfully solves their use cases and produces more useful patterns than the alternative approaches which are adapted from the state-of-the-art static compressing pattern mining work [14, 15, 22]. Furthermore, we demonstrate that SWIFT outperforms the best existing method by up to 50% in compressing the real IoT event sequences, while at the same time providing up to a 4 orders of magnitude speedup. Additional experiments on synthetic data also confirm its scalability to large volume event streams.

2. PRELIMINARIES

2.1 Basic Terminology

An **event sequence stream** corresponds to a series of events continuously produced by a single device. At each time t_i, the device generates an event e_i of type E_i, denoted as (e_i, t_i). Typically a sliding window (of size W) [9] is applied to specify a finite subset of the most recent events from the event streams. A **window sequence** (or **sequence**) is a list of ordered events falling in the current window at time t_i, denoted as $S = \langle (e_{i-W+1}, t_{i-W+1})(e_{i-W+2}, t_{i-W+2}) \ldots, (e_i, t_i)\rangle$. The window slides when a new event (e_{i+1}, t_{i+1}) is received. The new event is inserted into the current window. The obsolete event (e_{i-W+1}, t_{i-W+1}) produced W time units ago is discarded from the window. Besides sliding by one, the window can slide every B time units or after every B tuples. The new events produced after t_i are added to the window, while the events produced before $t_{i-W+B+1}$ are removed.

A **sequence pattern** (or **pattern**) $P = \langle E_1 E_2 \ldots E_m\rangle$ is an ordered list of event types E_i. An **occurrence** of P in window sequence S, denoted by $O_P^S = \langle (e_1, t_1)\ (e_2, t_2)\ \ldots\ (e_m, t_m)\rangle$, is a list of events e_i ordered by time t_i, where $\forall\ (e_i, t_i) \in O_P^S\ (i \in [1 \ldots m])$, $(e_i, t_i) \in S$ and e_i represents an event type $E_i \in P$.

We say a pattern $Q = \langle E'_1 E'_2 \ldots E'_l\rangle$ is a **sub-pattern** of a pattern $P = \langle E_1\ E_2\ \ldots E_m\rangle$ $(l \leq m)$, denoted $Q \sqsubseteq P$, if integers $1 \leq i_1 < i_2 < \cdots < i_l \leq m$ exist such that $E'_1 = E_{i_1}$, $E'_2 = E_{i_2}$, $\ldots$, $E'_l = E_{i_l}$. Alternately, we say P is a **super-pattern** of Q. For example, pattern $Q = \langle AC\rangle$ is a sub-pattern of $P = \langle ABC\rangle$.

2.2 Mining Representative Patterns with MDL

The Minimum Description Length (MDL) principle introduced in [20] is widely used in data compression as a measure to select the encoding model that best compresses the data [11, 12, 3]. Given a set of models $\mathbb{M}$, the best encoding model $M_i \in \mathbb{M}$ is the one that minimizes $L(M_i) = L(D_i) + L(S|D_i)$, where D_i corresponds to the dictionary of M_i, $L(D_i)$ represents the length of D_i, and $L(S|D_i)$ represents the length of the data after being encoded with dictionary D_i. $L(D_i)$ and $L(S|D_i)$ are measured in the number of characters.

In [15, 22], to solve the pattern explosion problem, MDL is used as a metric to select the pattern set that most compresses a static sequence dataset, where the occurrences of the patterns in this set do not overlap with each other – each event is used by at most one pattern occurrence. This pattern set is considered as the best, because it lets analysts understand the key features of the sequence dataset with a minimal amount of information.

For example, consider the sequence fragment $S = \langle (s, 1)(l, 2)\ (a, 3)(s, 4)\ (l, 5)(a, 6)\ (s, 7)(l, 8)(s, 9)(l, 10)\rangle$ extracted from the lighting application shown in Fig. 1. Using traditional pattern

[1] Sliding WIndow-based Frequent paTterns

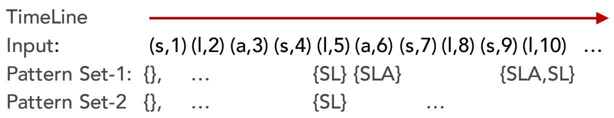

TimeLine
Input: (s,1) (l,2) (a,3) (s,4) (l,5) (a,6) (s,7) (l,8) (s,9) (l,10) ...
Pattern Set-1: {}, ... {SL} {SLA} {SLA,SL}
Pattern Set-2 {}, ... {SL} ...

Figure 1: Stream Representative Pattern Mining Example

semantics [2] to mine the frequent patterns, we consider a pattern as frequent if its frequency is larger than a predefined *support* threshold. When the support threshold is set to 2, this produces two frequent patterns P_1: $\langle SLA\rangle$ and P_2: $\langle SL\rangle$. P_1 has 2 occurrences and P_2 has 4 occurrences. In this case, events (s,1), (l,2), (s,4), (l,5) are used in both P_1 and P_2. Put differently, the two occurrences of P_1 ($\langle(s,1)(l,2)(a,3)\rangle$ and $\langle(s,4)(l,5)(a,6)\rangle$) overlap with the first two occurrences of P_2 ($\langle(s,1)(l,2)\rangle$ and $\langle(s,4)(l,5)\rangle$).

Thus we can construct the pattern set $\mathbb{P}_1 = \{P_1 : SLA, P_2 : SL\}$. Each pattern in $\mathbb{P}_1$ uses 2 occurrences. The first two occurrences of $\langle SL\rangle$ are not used, because they overlap with the two $\langle SLA\rangle$ occurrences. Alternatively, we could construct a pattern set $\mathbb{P}_2 = \{P_2 : SL\}$ that contains only one pattern $\langle SL\rangle$ but with all of its 4 non-overlapping occurrences.

Using $\mathbb{P}_1$ and $\mathbb{P}_2$ as two distinct dictionaries, the sequence S could be encoded as $S'_1 = \langle P_{1[1,2,3]}\ P_{1[4,5,6]}\ P_{2[7,8]}\ P_{2[9,10]}\rangle$ and $S'_2 = \langle P_{2[1,2]}\ (a,3)\ \ P_{2[4,5]}\ (a,6)\ P_{2[7,8]}\ P_{2[9,10]}\rangle$. We record the timestamps of the events covered by each pattern for compactness of presentation.

We use M_1 and M_2 to denote the encoding models corresponding to $\mathbb{P}_1$ and $\mathbb{P}_2$. Based on description length, $L(M_1)$ = $L(\mathbb{P}_1)$ + $L(S'_1)$ = 7 + 4 = 11, while $L(M_2)$ = $L(\mathbb{P}_2)$ + $L(S'_2)$ = 3 + 6 = 9. Therefore, by the MDL principle, $\mathbb{P}_2$ should be selected.

3. PROBLEM FORMULATION

We now introduce our semantics for continuously mining representative patterns from event stream based on the MDL principle.

Adapting MDL. In the traditional definition of minimum description length (MDL), the length of the *dictionary* $L(\mathbb{P}_i)$, which is part of the length of the encoding $L(M_i)$, corresponds to the number of characters used by the dictionary. This penalizes long patterns. In the example shown in Sec. 2.2, $\mathbb{P}_2$ containing one pattern P_2 = $\langle SL\rangle$ outweighs $\mathbb{P}_1$ that contains two patterns $P_1 = \langle SLA\rangle$ and $P_2 = \langle SL\rangle$, because the length of $\mathbb{P}_1$, denoted by $L(\mathbb{P}_1)$ = 7, is much larger than the length of $\mathbb{P}_2$, $L(\mathbb{P}_2)$ = 3. This unfortunately causes it to miss the longer $\langle SLA\rangle$ pattern which corresponds to the behavior of the lighting devices' alarm reporting. In our context, summarizing system behavior with the longest possible repeating pattern is intuitively preferable to using shorter patterns, because long patterns can model complex system behaviors.

Given this observation, we now slightly alter MDL to use *the number of distinct patterns* in the dictionary $\mathbb{P}_i$ as $L(\mathbb{P}_i)$. With this revised MDL definition, $L(\mathbb{P}_1)$ + $L(S'_1)$ = 2+4 = 6, while $L(\mathbb{P}_2)$ + $L(S'_2)$ = 1+6 = 7. Therefore, $\mathbb{P}_1$ would now be selected, which indeed matches our target problem.

MDL-based Representative Pattern (MRP) Semantics. We now define our proposed semantics to capture the set of patterns that most succinctly summarizes the input sequence.

DEFINITION 3.1. ***MDL-based Representative Patterns (MRP).*** *Given one window sequence $S = \langle(e_{i-w+1}, t_{i-w+1})$ $(e_{i-w+2}, t_{i-w+2})\ldots(e_i, t_i)\rangle$, the MDL-based representative pattern set or in short MRP is a set of patterns $\mathbb{P} = \{P_1, P_2, ..., P_k\}$ that together minimize the description length of S: $L(S|\mathbb{P})$+ $|\mathbb{P}|$, where*

(1) $\forall$ $O^S_{P_i}$ of $P_i \in \mathbb{P}$ and $\forall$ $O^S_{P_j}$ of $P_j \in \mathbb{P}$ ($j \neq i$), if event $e \in O^S_{P_i}$, then $e \notin O^S_{P_j}$;

(2) $\forall P_i \in \mathbb{P}$, $num(P_i, S) > 1$;

(3) Given a pattern $P_i \in \mathbb{P}$, $\forall$ events (e_x, t_x) and (e_y, t_y) $\in O^S_{P_i}$, where e_x and e_y are adjacent to one another, $\mid t_y - t_x \mid -1 \leq eventGap$.

In Def. 3.1, Condition (1) requires that the occurrences of the patterns in $\mathbb{P}$ do not overlap with each other, following the semantics adopted in [15, 22]. Condition (2) requires that each pattern P_i in $\mathbb{P}$ has at least two occurrences ($num(P_i, S) > 1$) in S. This excludes trivial patterns from $\mathbb{P}$ such as a pattern that corresponds to the whole sequence S itself. Condition (3) corresponds to the widely adopted event gap constraint [4], where $eventGap$ is a constant specified by the query specification. Then, the gap (either a time interval or the number of events) between any two adjacent events in a pattern occurrence cannot be larger than the $eventGap$ threshold.

Using the above MRP semantics, the stream representative pattern mining problem is defined next.

DEFINITION 3.2. ***Stream MRP Mining Semantics.*** *Given an event stream S produced by one device, a window size W, and a slide size B, the MRP mining problem is to continuously produce the MRP (Def. 3.1) from the sequence falling into the current window of S whenever the current window slides.*

Problem Complexity. The problem of mining MRP from a given input sequence is shown to be NP-hard [15]. Hence, an exhaustive search for the optimal result is practically infeasible. We thus instead must design an efficient heuristic strategy to meet the response time requirements of online applications.

To address this, we developed our SWIFT and B-SWIFT single-pass MRP mining algorithms presented next.

4. SWIFT: MINING MDL-BASED REPRESENTATIVE PATTERN SET

SWIFT is a one-pass strategy that processes each incoming event only once upon its arrival. We prove in Lemma 4.2 that SWIFT, while lightweight, always chooses the *best* pattern update operation for each incoming event that *most* reduces the description length of the sequence in the current window. Our complexity analysis in Sec. 4.1.4 also demonstrates the efficiency of our incremental update strategy.

Table 1: Meta Data Structures

Meta data	Description
Encoded Sequence	Sequence encoded using the patterns
Reference table	Patterns and their identifiers ⟨(key: pattern), (value: unique identifier of the pattern)⟩
Merge Candidate ($\mathbb{MG}$)	Singletons or patterns that could be merged to form new patterns
Match Candidate ($\mathbb{MC}$)	Potential matches of current patterns
Occurrence Pointers (*ocrPt*)	Pointers to the occurrences of elements; key: element; value: list of pointers to element occurrences

Overall Process of SWIFT. SWIFT consists of two major operations, namely *insert* and *expire*. Each time when a new event arrives, the oldest event in the current window will be removed from the current window if the current window is full. This triggers the *expire* operation that is responsible for determining the expiration of potential pattern occurrences. The *insert* operation is triggered by a new arrival event. It updates the current pattern set by either producing new pattern or new occurrence of existing pattern.

Table 1 summarizes the meta-data structures maintained by SWIFT for the window of the event stream. The "encoded sequence" maintains the compressed sequence with the raw events covered by the pattern occurrences replaced by the symbols representing the patterns. The "reference table" maintains the patterns and their identifiers. In addition, the "merge candidates", "match candidates", and "occurrence pointers" are maintained to accelerate both *insert* and *expire* operations. Details of these meta data structures are described in Sec. 4.1.

4.1 Insert Operation

We use the example sequence $S = \langle (a, 1)(b, 2)(c, 3)(a, 4)(b, 5)(c, 6)(a, 7)(b, 8)(c, 9)(d, 10)(a, 11)\rangle$ with events arriving one at a time. The window size is 10.

Overview of Insert Process. Given an incoming event, the *Insert* operation updates the pattern set $\mathbb{P}$. Insert classifies the update into two categories, namely "merge" and "match" based on whether a new pattern will be produced. The benefit of this separation is twofold. First, it allows us to design distinct search strategies that efficiently support the merge and match updates. Further, it ensures that each new pattern occurrence generated by a merge or match update will not trigger recursive update operations, saving significant CPU time as shown later in Lemma 4.1.

Given a new event e_i of type E_i, in the first merge case, e_i is *merged* with one existing singleton event or pattern occurrence of type E_j to form a *new pattern* $\langle E_j, E_i\rangle$ that is not in the current pattern set $\mathbb{P}$. $\langle E_j, E_i\rangle$ is called a *merge pair*. In the second match case, e_i would *match* one of the *current patterns* after merging with some previous singleton events or pattern occurrences of type E_k. In this case no new pattern is produced. We call $\langle E_k, E_i\rangle$ a *match pattern*. For each incoming event, multiple alternative merge pairs or match patterns might be available. From these, the *best one* is selected based on the MDL principle. That is, we choose the option that achieves the largest MDL benefit at this current state, i.e., the one that most reduces the minimum description length of the current window sequence.

As shown in Alg. 1, first the incoming event e_i of type E_i is appended to the encoded sequence at the tail (Line 2). At the same time e_i is inserted into the *occurrence pointer list* (Line 3) corresponding to E_i. Then *FindMerge* finds a singleton event type or a pattern denoted as E_j from the *merge candidates* ($\mathbb{MG}$), which after merged with E_i, achieves the largest MDL benefit compared to other merge candidates. The selected merge pair $\langle E_j, E_i\rangle$ is maintained in *mergePair* (Line 4). Similarly, the best match candidate is found from the *match candidates* ($\mathbb{MC}$) and maintained in *matchPattern* (Line 6). If both the selected merge pair and the match candidate cannot reduce the description length of the current sequence – in this case both *mergeBenefit* and *matchBenefit* are less than 0, no update on the existing patterns occurs (Lines 8-9). Otherwise, the one with the largest MDL benefit is applied (Lines 11-17).

By Alg. 1, the performance of the insert operation relies on the efficiency of procedure *FindMerge* for *merge pair identification* and procedure *FindMatch* for *match pattern identification*. Next, we introduce two efficient strategies separately for these procedures.

4.1.1 Merge Pair Identification

Given an incoming event e_i of type E_i, our merge pair identification strategy efficiently finds the *element* E_j (either a pattern or a singleton event type) that can be merged with E_i with the assistance of the *merge candidate* ($\mathbb{MG}$) and *occurrence pointer* ($ocrPt$) meta data structures.

Algorithm 1 Insert a new incoming event

```
 1: function INSERTEVENT(EVENT $e_i$, METADATA meta)
 2:     meta.encoded.add($e_i$)        ▷ add to encoded sequence tail
 3:     meta.ocrPt.get($e_i$).add($e_i$)
 4:     mergePair = FINDMERGE($e_i$, meta)
 5:     mergeBenefit = mergePair.benefit
 6:     matchPattern = FINDMATCH($e_i$, meta)
 7:     matchBenefit = matchPattern.benefit
 8:     if mergeBenefit $< 0$ && matchBenefit $< 0$ then
 9:         return meta
10:     else
11:         if mergeBenefit $\geq$ matchBenefit then
12:             newEle = MERGE(mergePair,meta)
13:         else
14:             newEle = MATCH(matchPattern,meta)
15:             mergePair = FINDMERGE(newEle, meta)
16:             if mergePair.benefit $> 0$ then
17:                 newEle = MERGE(mergePair,meta)
```

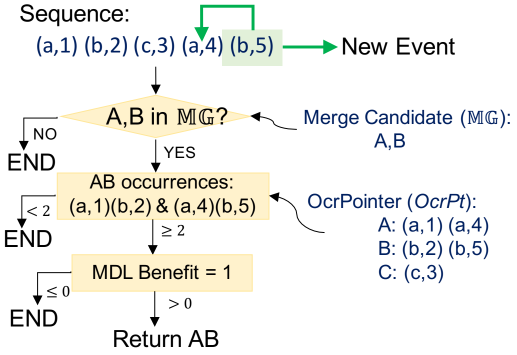


Figure 2: Merge Pair Identification

The $\mathbb{MG}$ structure maintains all elements with at least two occurrences. Specifically, an element will be excluded from $\mathbb{MG}$ if it has only one occurrence in the current window, because it has no chance to merge with any other element to form a new pattern. Further, we maintain a list of occurrences for each element in a hash table *ocrPt* with the element as key and the list of pointers to the occurrences of the corresponding element as value. *ocrPt* allows us to locate the occurrences of any element in constant time.

Algorithm 2 Merge Pair Identification

```
function FINDMERGE(ELEMENT $e_i$, METADATA meta)
    mergePair.benefit = $-\infty$
    if ocrPt.get($e_i$).size $> 1$ then
        for $e_j \in \mathbb{MG}$ do
            if $e_j.t$ - $e_i.t$ - 1 $>$ eventGap then
                return
            benefit = COMPUTEMDL($e_i$, $e_j$, meta)
            if benefit $> 0$ & benefit $>$ mergePair.benefit then
                mergePair.pair = ($e_i$,$e_j$)
                mergePair.benefit = benefit
        $\mathbb{MG}$.add($e_i$)
    return mergePair
```

Alg. 2 shows the merge pair identification process. First, given an incoming event e_i (type E_i), if an event of type E_i only appears once in the sequence, then event e_i cannot be involved in any new pattern. The algorithm then terminates (Lines 2-3). Otherwise, it will check whether the item e_j (of type E_j) which precedes incoming event e_i (type E_i) can form a merge pair $\langle E_j, E_i \rangle$. Note e_j is an *item* of the *encoded sequence*. Therefore, E_j corresponds to either a singleton event type or a pattern. A valid merge pair can be formed between E_i and E_j only if the following conditions are satisfied: (1) the gap between e_i and e_j is no larger than $eventGap$; (2) E_j is in the merge candidate $\mathbb{MG}$ structure; (3) the occurrences of E_j and E_i can form at least 2 occurrences of a new pattern $P = \langle E_j E_i \rangle$.

Evaluating the first two conditions is straightforward. To evaluate Condition (3), we first have to obtain all occurrences of E_i and E_j. This can be done in constant time using *ocrPt*. Then we find all valid occurrences of $P = \langle E_j E_i \rangle$ by joining the E_i list with the E_j list. Any join pair that satisfies the *eventGap* constraint is a valid occurrence of P. Since occurrences are organized by the arrival time of their last event, a *sort-merge join* algorithm can be applied here. The complexity is linear in the number of the occurrences of E_i and E_j.

EXAMPLE 4.1. *Fig. 2 demonstrates the merge process using the example sequence. Here eventGap is set as 0. Assume event* $(b, \mathit{5})$ *arrives. Since* $(b, \mathit{5})$ *can merge with the event* $(a, \mathit{4})$ *to form a valid occurrence of pattern* $\langle AB \rangle$ *(satisfying Condition (1)), we evaluate whether* $\langle AB \rangle$ *is a valid merge pair. Event type A has two occurrences* $(a, \mathit{1})$ *and* $(a, \mathit{4})$*. Event type B has two occurrences* $(b, \mathit{2})$ *and* $(b, \mathit{5})$*. Therefore, A and B are both in merge candidates and Condition (2) is satisfied. Second, we get the occurrences of A and B using ocrPt and evaluate Condition (3).*

Two occurrences of $\langle AB \rangle$ *can be constructed including* $(a, \mathit{1})(b, \mathit{2})$ *and* $(a, \mathit{4})(b, \mathit{5})$*. We then compute the MDL benefit gained by merging A and B. In this case, the MDL benefit is 1. More specifically, 4 singletons are replaced by 2 pattern identifiers of* $\langle AB \rangle$ *in the encoded sequence with the cost of producing one extra pattern in the reference table. Since eventGap is 0, only the event or pattern directly adjacent to* $(b, \mathit{5})$ *has the opportunity to form a valid occurrence of any pattern with* $(b, \mathit{5})$*. Therefore no more merge pair can be produced.* $\langle A, B \rangle$ *is selected as the best merge pair.*

4.1.2 Match Pattern Identification

Given a new event e_i, e_i can form matches with some existing patterns by merging with one existing pattern occurrence or singleton event. Intuitively, this can be done using a method similar to the merge process described above. That is, given a new event e_i, we first augment e_i with the item directly in front of e_i in the encoded sequence to form an occurrence O_{P_t} of a temporary pattern P_t. Then we examine whether P_t matches with any existing pattern. If it does, O_{P_t} has to be recursively augmented and examined, because matching a longer pattern will reduce the description length more. In each iteration one more item in front of the previous one is attached to the head of O_{P_t}. This process stops if O_{P_t} does not match any pattern. Clearly, this process is expensive because of the potentially large number of pattern match operations.

To reduce the costs, we propose a match strategy that efficiently identifies the best match for the new event. Given a new event e_i, we *no longer* recursively look backward for possible item combinations in front of e_i that could potentially form matches with existing patterns *on the fly*. Instead our strategy continuously predetermines future events that could form valid matches with the current patterns so far in the window. Then, when an expected event e_i arrives, the matches can be constructed immediately.

Our match strategy relies on the *match candidates structure (*$\mathbb{MC}$*)* that we dynamically construct and maintain. It contains all possible match candidates. Each candidate is in the following format [expected-event, match-pattern, currentPos, timestamp]. For example, $[B, P_2(ABC), \mathit{1}, [\mathit{7}]]$ indicates that a type B event is *expected* to form a match with pattern $P_2 = \langle ABC \rangle$. "1" represents the position of A in P_2. This indicates that an A event has already been received. "$[\mathit{7}]$" represents the position of the A event in sequence S, which we use to evaluate the event gap constraint in Def. 3.1. We index match candidates according to the expected events, with candidates sharing the same "expected-event" being grouped together. Therefore, given a new event e_i, all match candidates that are expecting event type E_i can be accessed efficiently.

Next, we show how the $\mathbb{MC}$ structure is constructed. The matches are also derived with the update process of $\mathbb{MC}$.

Create New Match Candidates. A new event e_i (type E_i) will generate a set of match candidates. Each candidate corresponds to one pattern P_i that has E_i as prefix. The *expected-event* E_j corresponds to the event type next to E_i in P_i. The *currentPos* is set as 1, because E_i is the first event type in P_i received so far.

Note given an event type E_i, to quickly locate the patterns with E_i as prefix, existing patterns are indexed based on the prefixes using a hash map. Using this map, given an event e_i, the existing patterns with E_i as prefix can be obtained in constant time. Specifically, in this hash map the prefix E_i is used as the key. Accordingly, the patterns with E_i as prefix correspond to the value of the hash map.

Update Match Candidate. Given an incoming event e_i, the match candidates expecting event type E_i will transit to a new state. Specifically, we update the $expected\text{-}event$ to the next event type expected by $match\text{-}pattern$. We then insert the position of e_i into the $timestamp$ field. For example, given a match candidate MC_i $\{B, P_2(ABC), \mathit{1}, [\mathit{7}]\}$, after event $(b, 8)$ arrives, MC_i transits to $\{C, P_2(ABC), \mathit{2}, [\mathit{7}, \mathit{8}]\}$. In this process, if e_i is the last event expected by pattern P_i in MC_i, then a match of pattern P_i will be generated. If MC_i is finally selected by some insert, no new match candidate will be produced, because e_i has been used by an occurrence of P_i and because we enforce the non-overlapping constraint.

Alg. 3 shows the process of match pattern identification. For each incoming event e_i, we transit the match candidates expecting type E_i event and produce the full matches (Line 5). Then the MDL principle is applied to select the best match mechanism (Line 6) among all full matches, which is the one with the largest MDL benefit (Lines 7-9). The final decision on whether the merge or match operation should be conducted for e_i is made afterwards in the *InsertEvent* function (Alg. 1). In addition, if insert operation does not form any new pattern occurrence involving e_i by either match or merge, a set of new match candidates corresponding to patterns starting with event type E_i is initialized and inserted into $\mathbb{MC}$ (Lines 10-12).

EXAMPLE 4.2. *Fig. 3 shows the match process of pattern* $P2 : \langle ABC \rangle$*. First, event* $(a, 7)$ *arrives. Since there is no candidate in* $\mathbb{MC}$ *so far, no transition action is triggered. Moreover, since a new pattern* $\langle ABC \rangle$ *has been formed which starts with event A, a new match candidate* $\{B, P_2, \mathit{1}, [\mathit{7}]\}$ *is initialized and inserted into* $\mathbb{MC}$*. This indicates that one event has arrived at time 7 that matches the first event type of pattern* P_2*. A type* B *event is expected now.*

As shown in Fig. 3, the match candidate can be considered a finite state automaton. With the arrival of event $(a, 7)$*, the automaton transits from state* S *to state* S_1*. Then event* $(b, 8)$ *comes next. It transits the match candidate from state* S_1 *to state* S_2*, which is* $\{C, P_2, \mathit{2}, [\mathit{7}, \mathit{8}]\}$*. No new match candidate is generated in this*

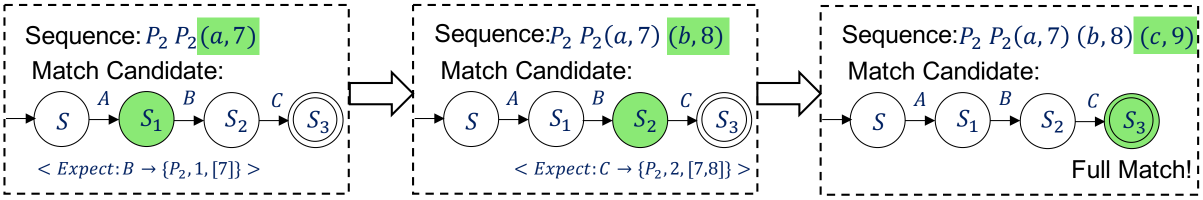

Sequence: P_2 P_2 $(a,7)$
Match Candidate:
S
A
S_1
B
S_2
C
S_3
< Expect: $B \rightarrow \{P_2, 1, [7]\}$ >
Sequence: P_2 $P_2(a,7)$ $(b,8)$
Match Candidate:
S
A
S_1
B
S_2
C
S_3
< Expect: $C \rightarrow \{P_2, 2, [7,8]\}$ >
Sequence: P_2 $P_2(a,7)$ $(b,8)$ $(c,9)$
Match Candidate:
S
A
S_1
B
S_2
C
S_3
Full Match!

Figure 3: Example of Matching Pattern $\{P_1 : ABC\}$

Algorithm 3 Match Pattern Identification.

```
function FINDMATCH(EVENT $e_i$, METADATA meta)
    matchPattern.benefit = $-\infty$
    if $\mathbb{MC}$.get($E_i$).size $> 0$ then
        for $a_i \in \mathbb{MC}$.get($E_i$) do
            transit to next state
            benefit = COMPUTEMDLBENEFIT($a_i$, meta)
            if benefit $> 0$ & benefit $>$ matchPattern.benefit then
                matchPattern = $a_i$
                matchPattern.benefit = benefit
    if matchPattern.benefit $< 0$ & mergePair.benefit $< 0$ then
        for $P_i \in \mathbb{P}$ & $P_i$.startWith($E_i$) do    ▷ traverse patterns
            $\mathbb{MC}$.add(INIT($P_i$))
        return matchPattern
```

step, because no pattern starts with B. Finally, after the event $(c, 9)$ arrives, a match of pattern P_2 is formed. The MDL benefit is then computed as MDL = (len(P_2) − 1) = 2.

Avoiding Recursive Updates. Intuitively, to further reduce the description length, the new pattern occurrence e generated by this just described merge or match update process should be considered as a new incoming event that may trigger recursive update operations. However, this will introduce large processing costs. SWIFT successfully avoids this *recursive update* process because of our customized merge pair and match pattern identification mechanisms.

LEMMA 4.1. *Given a new pattern occurrence e produced by the insert operation presented in Sec. 4.1, at most one update operation can be triggered by e.*

Proof. We prove Lemma 4.1 by showing: (1) if e is produced by *merge*, no further update is required; (2) if e is produced by *match*, e can trigger at most one additional *merge* update, or none.

Proof of (1): by proving that (a) e cannot produce any new match; and (b) e cannot produce any new merge.

Proof of Condition (a): (by contradiction). Suppose there exists a match MC_i for e which can further reduce the description length, this MC_i is guaranteed to be a better option than the match or merge operation that produced e itself when handling the incoming event e_i. In that case, this new match MC_i would have already been selected by Algorithm 1. In other words, e would not be produced at all. Therefore, there does not exist such a match for e.

Proof of Condition (b): (by contradiction). Let $P = \langle E_j E_i \rangle$ be the new pattern generated by merging E_j and E_i. Suppose there exists another event E_x in front of P that can merge with P. This indicates that E_j and E_x were not merged to $\langle E_x E_j \rangle$ previously. The only reason why this may not have happened would have been that E_j had an option that gained larger MDL benefit than merging with E_x. However, in that case, E_j would not exist as singleton, because E_j would already have been merged to produce either a new pattern or a match with an existing pattern. This contradicts the fact that E_j does exist and is being merged with E_i. Therefore, the new pattern P cannot merge with any other event type E_x again.

Proof of (2): by proof of Condition (a), e cannot produce any new match. Suppose e produces a new merge, then by Condition (1), this new merge cannot produce any further update. Therefore, e can only trigger at most one additional merge. ■

4.1.3 The Optimality of Insert

LEMMA 4.2. *Given an incoming event e_i, insert always chooses the update operation for e_i that most reduces the MDL score of the current sequence.*

Proof. We prove Lemma 4.2 by proving: (1) SWIFT always gets the best merge pair and (2) SWIFT always gets the best match.

We first prove Condition (1) by contradiction. Let (e_i, e_j) denote the merge pair found by SWIFT. Suppose there exists a singleton or a pattern e'_j that can form a merge pair (e_i, e'_j) with a larger MDL benefit, then e'_j must have at least two occurrences. Thus it should be included in $\mathbb{MG}$. In this case, (e_i, e'_j) in fact should have been returned as the best merge pair. This contradicts the fact that SWIFT returned (e_i, e_j) as the merge pair. Therefore, (e_i, e'_j) cannot get a larger MDL benefit compared to (e_i, e_j). Condition (1) holds.

Next, we prove Condition (2). Let MC_i denote the best match found by SWIFT for event e_i. Suppose there exists a match $MC_j = \langle E_1 E_2 \dots E_i \rangle$ that achieves a larger MDL benefit than MC_i. Since a match candidate is guaranteed to be initialized and then updated as $e_1, e_2, \dots, e_{i-1}$ arrives sequentially, therefore MC_j must exist in $\mathbb{MC}$ when event e_i comes. Since MC_j has a larger MDL benefit than MC_i, SWIFT should have returned MC_j instead of MC_i. This contradicts the fact that MC_i was returned. Therefore, there is no MC_j that has larger MDL benefit than MC_i. Thus Condition (2) also holds. This proves Lemma 4.2. ■

4.1.4 Time Complexity Analysis of Insert

We analyze the amortized time complexity of the insert operation per event. Its complexity is determined by the two processes, namely *merge pair identification* and *match pattern identification*. Let N denote the number of events received in the current window. Let $|E|$ denote the number of event types. Let $|P|$ denote the number of patterns found in the current window.

Merge Pair Identification. Given one incoming type E_i event e_i, assume its frequency in the window is x. To find the best merge pair, in the worst case we have to examine at most $x \times \tau$ elements in front of the type E_i events, where τ = *eventGap* + 1. Therefore, the processing time of the type E_i event is determined by the frequency of E_i and τ. Then the amortized time complexity of merge pair identification per event is $O(E(f(x)) \times \tau)$, where $E(f(x))$ indicates the *expectation* of the frequency of each event type.

Match Pattern Identification. The match pattern identification is composed of two processes, namely the *match candidate update* and *match candidate creation* described in Sec. 4.1.2. Given an incoming event e_i, the match candidate update process updates match candidates that are expecting event type E_i. At the same time, the match candidate creation process creates a new match candidate for each pattern that starts with E_i. Assume in the current window there are y patterns that contain E_i, then the number of updates and creations is no more than y. The amortized time complexity of match pattern identification per event is $O(E(f(y)))$, where $E(f(y))$ indicates the *expectation* of the number of patterns in which each event type is involved in.

Overall the time complexity of *insert* is determined by $E(f(x))$ and $E(f(y))$. In general, $E(f(x))$ and $E(f(y))$ tend to increase when the number of events N or the number of patterns L increases, while $E(f(x))$ and $E(f(y))$ tend to decrease when the number of event types $|E|$ increases. Since typically N, $|P|$, and $|E|$ all increase together when the data volume increases, our SWIFT is scalable to high volume stream data as also confirmed by our empirical study (Sec. 6.4).

4.2 Expire Operation

Once an obsolete event is discarded from the current window, the *expire* operation is triggered to update the existing patterns. The expire operation can be classified into two categories: (1) expiring a singleton event; and (2) expiring an event involved in an occurrence of one pattern.

Expiring a singleton event is straightforward. Since it is not involved in any pattern, it can be simply removed from all meta data structures including the "encoded sequence", "merge candidate" $\mathbb{MG}$, "match pattern" $\mathbb{MC}$, and "occurrence pointers" *ocrPt*.

Expiring an event used by an occurrence O of one pattern is more complicated. After an event is discarded, the remaining events in O become singleton events. They can potentially merge with existing singletons or patterns to form new patterns or to match some existing patterns. Furthermore, expiring one occurrence of a pattern P_i might make P_i no longer frequent if P_i only had two occurrences. In such a case, the events in the other occurrence of P_i also would have to be handled.

The idea of the expire operation is to treat singleton events produced due to event expiration as new events. In other words, such expiration triggered events are processed one by one, leveraging the insert operation (Sec. 4.1).

However, unlike for true incoming events which always correspond to the latest arrival in the sequence, these events e_i already arrived earlier than other events. Therefore, e_i has to form merge or match pairs with events that arrived later than e_i. Accordingly, the *merge* and *match* processes in the insert operation have to be slightly modified.

First, the modification of the *merge process* is minor. Given one such new expire event e_i, now the merge pair has to be discovered in the events that arrived later than e_i. The search is conducted in the arrival order of the events.

Second, the modification of the *match process* mainly concerns the match candidates $\mathbb{MC}$ meta data. Originally $\mathbb{MC}$ maintains the "to be completed" patterns and their expected events precomputed beforehand. In the expire operation, the event e_i arrived earlier than any other events. Therefore, it is impossible to prepare the match candidates beforehand. Alternatively, we build a temporary match candidate set for these events on the fly. These events are processed in their arrival order. As shown in Fig. 4, initially the match candidate is empty. After the earliest event $e_i \in E_i$ is processed, the patterns with E_i as prefix are inserted into the candidate set. Then when processing the next "expire" event e_{i+1}, the candidates which are expecting E_{i+1} are updated to expect the next event type. New match candidates are also constructed corresponding to the patterns with E_{i+1} as prefix. After all these singleton "expire" events have been processed, this temporary match candidate set is discarded.

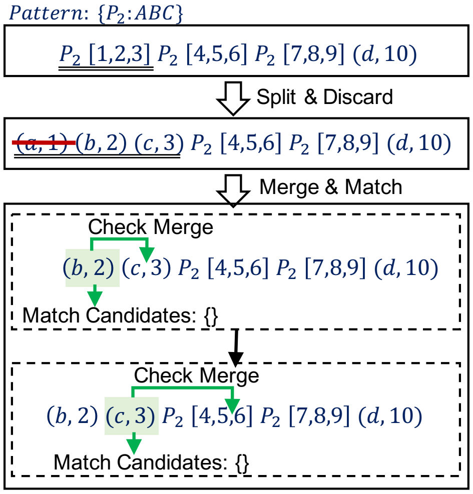

Pattern: {P2:ABC}
P2 [1,2,3] P2 [4,5,6] P2 [7,8,9] (d, 10)
Split & Discard
~~(a, 1)~~ (b, 2) (c, 3) P2 [4,5,6] P2 [7,8,9] (d, 10)
Merge & Match
Check Merge
(b, 2) (c, 3) P2 [4,5,6] P2 [7,8,9] (d, 10)
Match Candidates: {}
Check Merge
(b, 2) (c, 3) P2 [4,5,6] P2 [7,8,9] (d, 10)
Match Candidates: {}

Figure 4: Process of pattern expiration for running example

EXAMPLE 4.3. *In our running example as shown in Fig. 4, the input sequence is encoded into S'* $= \langle P_{2[1,2,3]}, P_{2[4,5,6]}$ $P_{2[7,8,9]}(d, 10)\rangle$ *with* $\{P_2 : ABC\}$ *after 10 events have arrived. Since the window is full now, the earliest event* $(a, 1)$ *has to be expired before accepting the new coming event* $(a, 11)$. *Fig. 4 shows the expiration process. By checking the encoded sequence S', we find that the expiring event* $(a, 1)$ *is involved in one occurrence of pattern* P_2. *This occurrence is then split into singleton events* $(b, 2)$ *and* $(c, 3)$, *with each subsequently treated as a new event. In this case, no match or merge can be formed on the two singletons, since no existing pattern has B or C as prefix. Therefore, they are simply inserted back into the sequence at their original positions. After the first event has been expired, now the sequence is encoded as* $\langle(b, 2)(c, 3)P_{2[4,5,6]}\ P_{2[7,8,9]}(d, 10)\rangle$.

5. SWIFT WITH BATCH UPDATES

In addition to the event-at-a-time update, batch updates are often preferable in streaming data analytics to support high velocity streams. In such cases, it is not necessary to update results every time a new event arrives. Instead, results are updated only after a batch of new events has arrived. Intuitively, batch updates can be supported by directly applying our SWIFT method to process each new event in the batch one at a time. However, this solution will waste significant computational resources on unnecessary update operations triggered by each event as described below.

First, expiring an event involved in an occurrence O_P of some pattern P will generate a set of singleton events. In the batch update, applying the expire operation to process each of these events one at a time tends to trigger many unnecessary merge and match operations, because the new pattern occurrences produced by the previous expire operation might expire again and be discarded when handling the next expiring event. Second, expiring one occurrence of a pattern P that only has two occurrences would result in the split of the other occurrence of P, because P is no longer frequent. Instead, when dealing with batch updates, a new occurrence of P might be produced in the new incoming batch. In this case, P is ultimately frequent again − having been only temporarily infrequent. Therefore, it had not been necessary to split the other occurrence of P to begin with in hindsight.

To address these drawbacks, we enhance SWIFT to directly support batch updates, now called B-SWIFT. B-SWIFT employs a *lazy update* strategy. This ensures that only the update operations that could cause a change in the final set of patterns are applied. Next, we introduce our batch expire (Sec. 5.1) and batch insert (Sec. 5.2) operations, using the following example: $S = \langle (a, 1)(b, 2)(a, 3)(c, 4)(d, 5)(a, 6)(c, 7)(d, 8)$ $(a, 9)(b, 10)(a, 11)(c, 12)(d, 13)(a, 14)(b, 15)\rangle$, batch size = 5 and window size = 10. B-SWIFT outperforms SWIFT in both computational costs and description length as confirmed in our experiments (Sec. 6).

Similar to the original SWIFT approach, B-SWIFT supports two major operations, namely expire and insert. However, now the expire and insert operations are conducted on a whole event batch. Next, we introduce in detail the batch expire and insert operations.

5.1 Batch Expire Operation

Instead of expiring each event independently, the batch expire operation, called *B-Expire*, processes the batch of expired events as a whole. In general the batch expiration process can be divided into three categories: (1) expire a singleton event; (2) expire all events of a pattern occurrence; and (3) expire a subset of the events of a pattern occurrence.

First, to be expired, singleton events will be discarded without further processing. Second, if all events of a pattern occurrence expire, all events in this occurrence will be discarded immediately. Third, the process of expiring a subset of the events of a pattern occurrence is analogous to the process of expiring one event from a pattern occurrence as discussed in Sec. 4.2. The expired events are removed at once, while the remaining events in this occurrence are inserted back into the encoded sequence as singleton events.

However, the second and third types of expire operations might make the pattern P no longer valid because of the loss of one occurrence. Unlike the event-at-a-time expire operation, the split of the other occurrence of P is *postponed* and kept as a *to-be-split* pattern P_s in a list. This optimization is based on the observation that new occurrences of P_s might be formed in the newly incoming batch such that eventually P_s might still be valid. These *to-be-split* patterns will thus not be handled until after processing the new incoming batch.

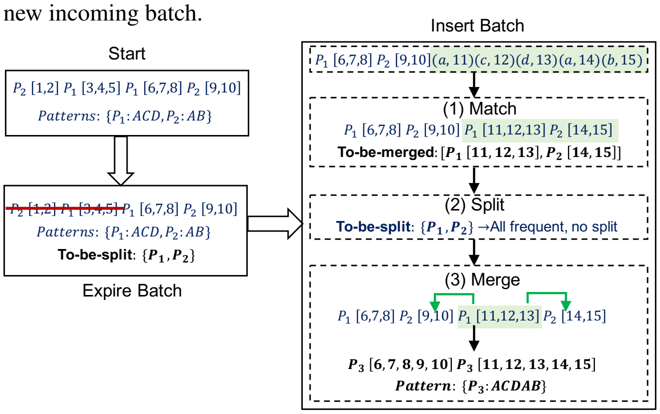

Start
P_2 [1,2] P_1 [3,4,5] P_1 [6,7,8] P_2 [9,10]
Patterns: {P_1:ACD, P_2:AB}
~~P_2 [1,2] P_1 [3,4,5]~~ P_1 [6,7,8] P_2 [9,10]
Patterns: {P_1:ACD, P_2:AB}
To-be-split: {P_1, P_2}
Expire Batch
Insert Batch
P_1 [6,7,8] P_2 [9,10](a, 11)(c, 12)(d, 13)(a, 14)(b, 15)
(1) Match
P_1 [6,7,8] P_2 [9,10] P_1 [11,12,13] P_2 [14,15]
To-be-merged: [P_1 [11, 12, 13], P_2 [14, 15]]
(2) Split
To-be-split: {P_1, P_2} →All frequent, no split
(3) Merge
P_1 [6,7,8] P_2 [9,10] P_1 [11,12,13] P_2 [14,15]
P_3 [6, 7, 8, 9, 10] P_3 [11, 12, 13, 14, 15]
Pattern: {P_3:ACDAB}

Figure 5: Expire & Insert a Batch

EXAMPLE 5.1. *Fig. 5 demonstrates the batch expiration process of our running example. As shown at the left of Fig. 5, after processing the first 10 events the sequence is encoded as* $S = \langle P_{2[1,2]}\ P_{1[3,4,5]}\ P_{1[6,7,8]}\ P_{2[9,10]}\rangle$, *with two patterns* $\{P_1 : ACD,\ P_2 : AB\}$ *formed. Since the batch size is 5, as the next batch of events arrives, the 5 earliest events are removed from the current window. In this case, both* P_1 *and* P_2 *have a whole occurrence expire, namely* $P_{1[3,4,5]}$ *and* $P_{2[1,2]}$. *Thus, they are discarded immediately. Furthermore, now* P_1 *and* P_2 *only have one valid occurrence. Thus they are inserted into the to-be-split pattern list.*

5.2 Batch Insert Operation

The batch insert operation *B-Insert* aims to avoid any unnecessary pattern split operations caused by batch expiration by prioritizing the order of the operations conducted on the new batch.

Match First. Given a new batch of events, unlike the event-at-a-time insert operation (*E-Insert*) which interleaves match and merge processes when handling each individual event, *B-Insert* conducts the match process sequentially on the events in the new batch, while the merge process is postponed till the end of *B-Insert*. It then determines whether a pattern P_s in the *to-be-split* pattern list does not need to be split due to a new occurrence of P_s produced in the match process.

In the example shown in Fig. 5, the new batch $\langle (a, 11)(c, 12)(d, 13)(a, 14)(b, 15)\rangle$ matches patterns $\{P_1 : ACD\}$ and $\{P_2 : AB\}$ and therefore can be rewritten as $\langle P_{1[11,12,13]} P_{2[14,15]}\rangle$. Further, since new occurrences of P_1 and P_2 are generated using this incoming batch, they are removed from the *to-be-split* list.

Merge Later. The singleton events either from the new batch or produced by the split of pattern occurrences due to the event expiration and pattern occurrences produced in *match* processes are inserted into *to-be-merged* list. These elements are then evaluated one by one using the merge process described in Sec. 4.1. Here instead of directly using the *FindMerge* algorithm (Alg. 2) introduced in Sec. 4.1, we design a batch version of the merge pair identification process *FindBatchMerge*. It continuously examines each element in the to-be-merged list until one merge pair with positive MDL benefit is formed or the to-be-merged list is empty.

In the running example, since no singleton is generated due to split, the *to-be-merged* list only contains the two match occurrences $P_{1[11,12,13]}$ and $P_{2[14,15]}$ formed in the new batch. Up to now, the encoded sequence is $\langle P_{1[6,7,8]} P_{2[9,10]} P_{1[11,12,13]} P_{2[14,15]}\rangle$, where $\{P_1 : ACD$, $P_2 : AB\}$. Next, the *Merge* process is executed. It starts with the element $P_{1[11,12,13]}$. Since this element can be merged with pattern P_2, it constructs a merge pair (P_1, P_2) and produces a new pattern $\{P_3 : ACDAB\}$. The encoded sequence then is updated to $\langle P_{3[6,7,8,9,10]} P_{3[11,12,13,14,15]}\rangle$. Since the *to-be-merged* is empty now, the batch merge process terminates.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup & Methodology

We experiment with both real-world and synthetic datasets. The results of the synthetic data experiments confirm the scalability of our SWIFT to large volume and high velocity event streams.

Real Datasets: (1) A *log file dataset* extracted from a mobile application that tracks driver behavior. The application is developed by a startup company based in Cambridge MA. We obtained log files from 10,000 devices ($|D| = 10,000$) with 1,790 types of events ($|E| = 1790$). The average length of each sequence is 34,097. In the experiments, we consider each device as one data stream.

(2) A *lighting dataset* produced by our industrial lighting collaborators. It consists of the control messages exchanged between commercial lighting devices and cloud servers. The data is from 283,144 devices ($|D| = 283,144$) with 13 types of events ($|E| = 13$). The average length of each sequence is 456.

This dataset features different characteristics than the log file dataset. The log file data represents complex interactions and state transitions, since in the mobile app multiple threads were performing independent actions, writing to the log at the same time. This results in many operations that often but not always occur in a certain order. The lighting device is instead a single thread. Therefore, the generated sequences are comparatively regular.

Synthetic Dataset: We generated event stream data to evaluate the scalability of SWIFT. Since the sequence generators used in the literature [6, 5] were designed to only generate a large number of short sequences and do not offer control of the number of patterns within the synthetic data, we developed a new event stream data generator. It supports a number of input parameters that allow us to control the key properties of the generated sequence stream data as listed in Tab. 2. These include the number of event types, the number of patterns, the average length of the patterns, the batch size and the window size of the stream. In particular we inject random noise, i.e., events that do not form any pattern occurrence, to mimic real-world data. The noise rate is configurable as shown in Table 2.

Table 2: Input Parameters to Sequence Data Generator

Symbol	Description
$\|E\|$	Number of event types
$\|P\|$	Number of patterns
$\|L\|$	Average length of the patterns
$\|B\|$	Batch size
$\|W\|$	Window size
$\|N\|$	Number of batches
e	noise rate

Experimental Setup. All experiments are conducted on a computer with Intel 2.60GHz processor, 500GB RAM, and 8TB DISK. It runs Ubuntu operating system (version 16.04.2 LTS). The code used in the experiments is available via GitHub: https://github.com/OutlierDetectionSystem/SWIFT.

Approaches. We evaluate six different systems. We adapt four static methods, namely *SeqKrimp*, *GoKrimp*, *SQS*, and *CSC* to the streaming context. All of them use MDL to mine compressive patterns. (1) *SeqKrimp*: the two-step static pattern mining method of [15]; (2) *GoKrimp*: the one-step static pattern mining method of [15] that directly mines representative patterns; (3) *SQS*: the static pattern mining method of [22] that mines the representative patterns by recursively scanning the data. (4) *CSC*: the one-step static pattern mining method of [14] (similar to GoKrimp; but allow overlapping among occurrences of different patterns); More details of the above four algorithms are described in the related work (Sec. 7); (5) SWIFT: our incremental streaming representative pattern mining approach (Sec. 4); (6) B-SWIFT: enhanced SWIFT to support batch updates (Sec. 5). Each time the window slides, the patterns in the new window are mined again. As we will illustrate, the SWIFT and B-SWIFT approaches outperform the state-of-the-art in almost every case in both efficiency and effectiveness.

Metrics. We evaluate the above approaches using the following metrics. First, similar to [15, 22, 14] we use the *average compression rate* (ACR) as metric to evaluate the effectiveness of our approach at compressing patterns. The lower the rate is, the better the set of compressed patterns is. Specifically, ACR is defined as the fraction of the average MDL score (averaged per window) over the window size, denoted as $rate = \frac{AvgMDL}{WindowSize}$. The MDL score represents the description length of a particular window sequence. We also measure the coverage rate, which is computed as the ratio of the event types covered by the produced patterns to all event types in each window.

In addition, to qualitatively evaluate how good SWIFT is at capturing typical patterns, we measure the number of captured useful patterns (*NOP* for short) and *precision* – the ratio of useful patterns over all produced patterns. Recall is not measured here, because it is impossible for the domain experts to manually find out all typical patterns from such large sequence data.

Furthermore, we measure the *processing time* per window for efficiency evaluation. It measures the overall CPU time consumed to generate the patterns for each window including the costs of both insert and expire.

6.2 Evaluation of Effectiveness

6.2.1 Evaluation of Compression Rate (ACR)

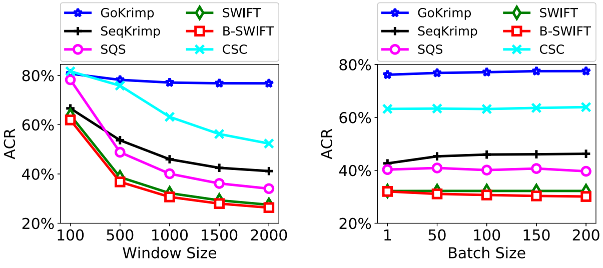
To demonstrate the effectiveness of SWIFT (Sec. 4) and B-SWIFT (Sec. 5) in generating compressing patterns, we evaluate the average compression rate (ACR). Overall, SWIFT and B-SWIFT outperform the state-of-the-art in terms of the ACR metric under various data characteristics and parameter settings.

Log File Dataset. Fig. 6 shows the results on the mobile app log file dataset. The input parameters are set to WindowSize=1000 and BatchSize=100 by default, except when varying the corresponding parameter. As shown in Fig. 6, both our two approaches, SWIFT and B-SWIFT, consistently outperform other methods w.r.t the average compression rate (ACR) in all cases. The good ACR values result from the decision we make for each incoming event. Given an incoming event, we always select the operation out of all merge and match options that minimizes the MDL score. In other words, each event makes its own choice based on the context in which it occurs. This results in a good ACR. GoKrimp is the worst in ACR (about 80%) in all cases, because it greedily selects the most frequent event types to form patterns. However, obviously two most frequent event types do not necessarily form any valid representative pattern. SeqKrimp outperforms GoKrimp, achieving 40% ACR. This is because of its two-step strategy that selects the representative patterns from the precomputed frequent pattern set. Clearly selecting representative patterns from a set of pattern candidates tends to be more effective than the naive strategy of forming patterns based on the frequency of each singleton event type. Although CSC allows overlapping among the occurrences of different patterns, CSC is only slightly better than GoKrimp due to the limited types of patterns it can support. Among the state-of-the-art methods, SQS [22] achieves the best ACR (35%), although still worse than our SWIFT. Its good ACR is achieved via a complex search strategy that requires an iterative scan of the data, resulting in an extremely slow execution time for SQS as shown in Sec. 6.3.

Furthermore, as shown in Fig. 6(a), as the window size goes larger, the ACR of all methods improves. This is expected. The larger the window is, the more patterns can be formed. Therefore, more singleton events will be replaced by patterns. This leads to a better compression rate.

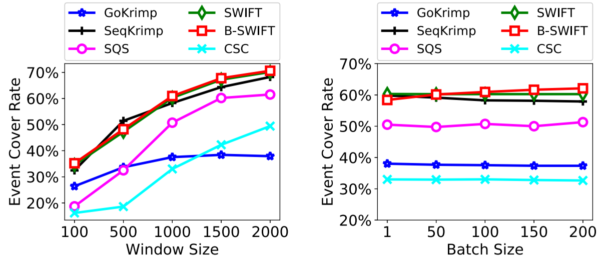
In addition, as shown in Fig. 6(b), as the batch size goes up, the ACR of B-SWIFT keeps improving, while the ACRs of other methods do not. This is because B-SWIFT makes the decision based on all events in the batch, while other methods process each event in isolation. When the batch size gets larger, B-SWIFT has more options to consider.

Note that the ACRs of our SWIFT approach are at least 0.2 on the two real datasets, which are very impressive. The reason is that our SWIFT is good at capturing long patterns which are preferred in modeling complex system behaviors. Our further analysis shows that in each window, most of the events are encoded by the patterns, although some of events are not covered by any pattern and remain as singleton events. If we consider each singleton event type as one special length-1 pattern, the *expected* length of all patterns (including the length-1 patterns) discovered by SWIFT is larger than 5. Therefore, replacing the raw events in the input sequence with the symbols representing the patterns, our SWIFT is able to represent an input sequence using at most one fifth of the original characters.



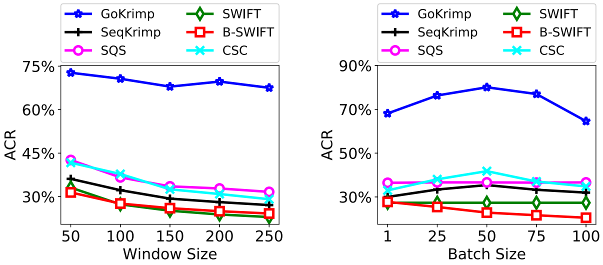
(a) Varying Window Size (b) Varying Batch Size

Figure 6: Average Compression Rate on Log File Data (smaller is better)



(a) Varying Window Size (b) Varying Batch Size

Figure 8: Coverage Rate on Log File Data (larger is better)



(a) Varying Window Size (b) Varying Batch Size

Figure 7: Average Compression Rate on Lighting Data

Lighting Dataset. Fig. 7 shows the results on the lighting dataset. The input parameters are set to WindowSize=100 and BatchSize=10 by default, except varying the corresponding parameter. The window size and batch size are relatively small, because each sequence in this real dataset is short. As shown in Fig. 7, again our SWIFT and B-SWIFT outperform all other alternatives in terms of the ACR. However, different from the log file data, SeqKrimp performs better than SQS. The reason is that this dataset is regular such that it is relatively easy to capture the frequent patterns. Therefore, using the discovered patterns as candidates, SeqKrimp is able to effectively capture the representative patterns. The ACR of CSC is relatively good in handling this simplistic lighting data – close to SQS. This indicates that its search strategy benefits from the small number of event types.

6.2.2 *Evaluation of Coverage Rate*

In this experiment, we report the coverage rate averaged on all windows (Sec.6.2). Since the lighting data only has 13 event types, this metric is not particularly helpful in distinguishing the coverage rates of different methods. Therefore, here we report the results on the log file dataset while also varying window and batch sizes. When varying window sizes, we fix batch size as 100. When varying batch sizes, the window size is fixed as 1000.

As shown in Fig. 8, our SWIFT methods outperform all other methods in almost all cases. In particular, similar to the compression rate, GoKrimp and CSC perform poorly, because they greedily select the most frequent event types to form patterns. Only SeqKrimp is slightly better than SWIFT when the window size is 500. SeqKrimp is the method that is the closest to our SWIFT in terms of the coverage rate, because it uses a two-step solution that selects patterns from all possible candidate patterns produced at the first step. This somewhat increases the diversity of the event types covered by the selected patterns. However, this also makes it 100 times slower than SWIFT.

6.2.3 *Qualitative Evaluation of Discovered Patterns*

We also qualitatively evaluate the effectiveness of SWIFT at capturing the useful patterns that indeed represent the typical behaviors of the system. In this experiment, we use the lighting dataset produced by the lighting application. The results were manually evaluated by the application engineers. As shown in Table 3, SWIFT outperforms other methods in both the number of detected useful patterns (NOP) and precision. More specifically, 9 patterns out of the 12 patterns detected by SWIFT indeed represent typical behavior in the lighting application such as handshake process between the devices and the server, reporting alarm, registering new devices, etc. GoKrimp is the worst in terms of the quality of the captured patterns because of its over-simplified search heuristic that only constructs patterns from the most frequent event types. Although CSC finds more useful patterns than GoKrimp, its precision is even lower than GoKrimp. This is so, because it allows the overlapping among the occurrences of different patterns. This enlarges the space of possible pattern candidates and makes it find more invalid patterns. However, the number of useful patterns found by CSC is lower than our SWIFT due to the constraint on the types of patterns it can construct as discussed in Sec. 7. As expected, SQS and SeqKrimp work better than GoKrimp. However, SQS and SeqKrimp are worse than SWIFT, although they use much more CPU time than SWIFT as will be demonstrated in Sec. 6.3.

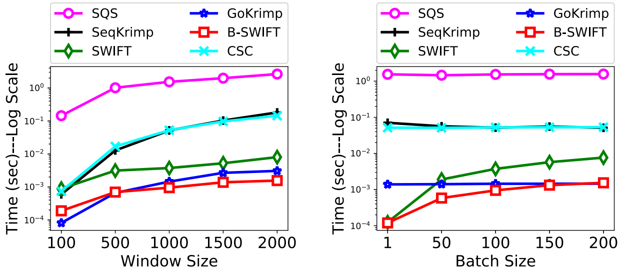
Table 3: Qualitative Evaluation

Methods	Number of meaningful patterns (NOP)	Precision
SWIFT	9	75.0%
SQS	7	58.3%
SeqKrimp	8	53.3%
GOKrimp	5	41.7%
CSC	7	38.9%

6.3 Evaluation of Efficiency

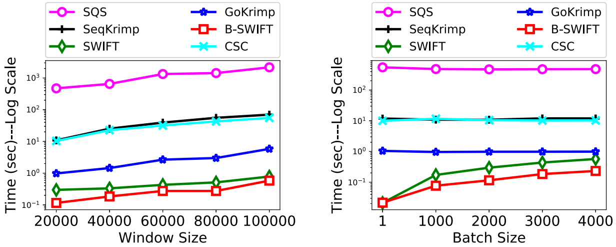
We investigate the processing time of our SWIFT approaches using the two real datasets by varying the $windowSize$ and $batchSize$. Overall, B-SWIFT consistently outperforms SWIFT and the state-of-the-art approaches by up to 4 orders of magnitude, while SWIFT that processes the incoming events one by one also outperforms the state-of-the-art by up to 3 orders of magnitude. Due to space constraint, we did not show the results on the small lighting dataset.

Log File Dataset. The parameters are set as *WindowSize=1000* and *BatchSize=100*, except when they are under variation. As shown in Fig. 9, B-SWIFT consistently outperforms SWIFT, SeqKrimp, CSC, and SQS in all cases up to 4 orders of magnitude. GoKrimp is the second fastest method. However, it is not effective in finding representative patterns. Its ACR values are very poor (80%) in all cases as shown in Fig. 6. This is due to its over-simplified greedy



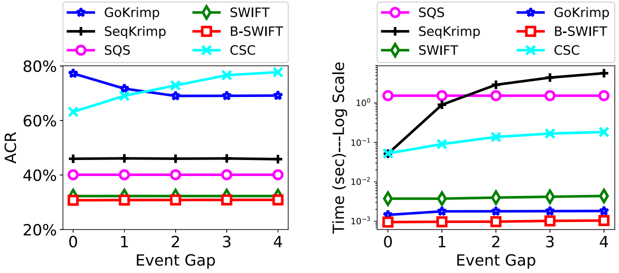
(a) Varying Window Size (b) Varying Batch Size

Figure 9: Processing Time on Log File Dataset



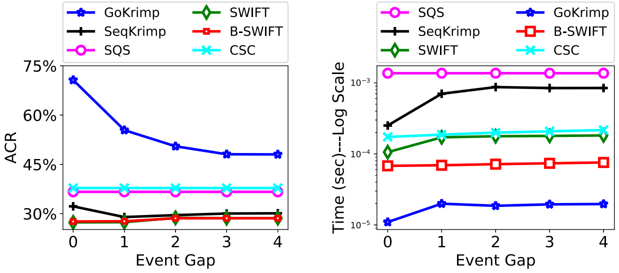
(a) Varying Window Size (b) Varying Batch Size

Figure 10: Processing Time on Synthetic Datasets



(a) Average Compression Rate (b) Processing Time

Figure 11: Varying Event Gap on Log File Dataset



(a) Average Compression Rate (b) Processing Time

Figure 12: Varying Event Gap on Lighting Dataset

search strategy. On the other hand, SQS is much slower than any other methods because of its complex iterative search strategy, although it has a relatively good ACR rate compared to SeqKrimp and GoKrimp. The CPU efficiency of CSC is very close to that of SeqKrimp in most cases.

Fig. 9(a) demonstrates the processing time results when varying window size from 100 to 2,000. B-SWIFT outperforms all other methods up to 3 orders of magnitude. As the window size increases, the processing time of B-SWIFT and SWIFT only increases gradually, while the processing time of SeqKrimp, CSC and SQS increases dramatically. Therefore, the larger the window size, the more SWIFT and B-SWIFT win. This confirms the scalability of SWIFT w.r.t. stream rates.

Fig. 9(b) shows the results of varying *batchSize*. Again, B-SWIFT consistently outperforms all other methods up to four orders of magnitude. The processing time of B-SWIFT is stable when the batch size increases, because B-SWIFT only performs the necessary update operations when handling a batch of new events. GoKrimp, SeqKrimp, CSC, and SQS reconstruct patterns in the new window from scratch whenever a new batch of events arrives. Therefore, their processing time does not change along the batch size. However, as shown in Fig. 9(b), even in the worst case, B-SWIFT is still 30 times faster than SeqKrimp and CSC and 3 orders of magnitude faster than SQS.

As for our two SWIFT approaches, the event-at-a-time SWIFT is more sensitive to the batch size than B-SWIFT, because each new event will trigger one update operation. When the batch size is set to 1, the processing time of SWIFT and B-SWIFT is almost identical. As the batch size increases, B-SWIFT outperforms SWIFT by up to 5 fold in its average processing time. This confirms the effectiveness of the *lazy update* strategy of B-SWIFT in eliminating the unnecessary update operations.

6.4 Efficiency Evaluation on Synthetic Data

We use synthetic datasets to evaluate how SWIFT and B-SWIFT perform on data streams with large window and batch sizes in order of 100,000 events. The parameters utilized to generate the synthetic datasets are set to $|E| = 5000$, $|P| = 1000$, $|L| = 10$, $|B| = 2000$, $|W| = 20000$, $|N| = 10000$ and $e = 0.01\%$ by default. We evaluate the processing time of SWIFT by varying the *windowSize* and *batchSize*.

As shown in Fig. 10, it takes our SWIFT approaches less than one second to process each window in all cases – hence meeting the real time response requirement of online applications, while it takes other approaches minutes or even hours. Fig. 10(a) demonstrates the processing time when varying window size from 20,000 to 100,000. Our SWIFT approaches, especially the B-SWIFT method, outperform all other methods up to 3 orders of magnitude. Better yet, our SWIFT approaches are scalable to the window size representing the volume of the event stream. Specifically, even when the window size increases to 100,000 events, the processing time is within one second.

Fig. 10(b) shows the processing time when varying the batch size from 1 to 4,000. Although the processing time for B-SWIFT and SWIFT both increase as the batch size gets larger, they are still faster than all other approaches up to 4 orders of magnitude. Even in the worst case when the batch size is 20% of the window size, B-SWIFT is still 4x faster than GoKrimp (with the later confirmed to fail in generating representative patterns), 51x faster than SeqKrimp and CSC, and 3 orders of magnitude faster than SQS. This confirms that our SWIFT approaches are scalable to a batch size practical in representing the typical velocity of the event stream.

6.5 Evaluation of the eventGap Parameter

We also conducted experiments to evaluate the impact of the eventGap parameter using the two real datasets we work with. We measure both the compression rate and the processing time. As shown in Fig. 11 and 12, our SWIFT consistently outperforms all other alternatives in both compression rate and processing times. First, as expected, when the eventGap increases, all methods use more CPU time because of the enlarged space of possible pattern candidates that must be examined. However, the processing time of our SWIFT and B-SWIFT increases more slowly than that of all other approaches. This is so, because our merge and match strategies effectively reduce the number of merge pair candidates and match pattern candidates. Second, it is interesting to see that as the eventGap rises, all methods do not show a clear improvement of the compression rate except for GoKrimp. However, although the compression rate of GoKrimp improves somewhat, it continues to remain much worse than that of other methods. This demonstrates that a large eventGap does not help in finding representative patterns, at least, not in the log file and the lighting datasets.

Although GoKrimp is slightly faster in speed than our SWIFT approach when handling the simple lighting dataset due to its oversimplified search strategy, it is ineffective in finding representative patterns and compressing the sequence as confirmed in Sec. 6.2.

7. RELATED WORK

Frequent Sequential Pattern Mining Semantics. Frequent pattern mining was first proposed in [2] to mine purchase patterns from a customer transaction dataset. A purchase pattern is called *frequent* if it occurs in more than *support* customer's transaction histories. However, this semantics suffers from the pattern explosion problem due to generating too many redundant patterns. To alleviate this problem, variations of the basic semantics [10, 25, 7, 8] were proposed. The *closed frequent pattern* semantics [10, 25] exclude a frequent pattern from the output when its support is identical to the support of any of its super-patterns. Therefore its pruning ability is weak. The *maximal frequent pattern* semantics [7, 8] assumes only the longest frequent patterns are representative and discards all sub-patterns of P when P is frequent. However, the longest patterns are not necessarily the only meaningful patterns. Sometimes P and its sub-patterns might both be representative if both of them occurs independently and frequently. Thus it tends to miss some representative patterns. Furthermore, it only handles the redundancy among a pattern and its sub-patterns. Partial overlapping relationships among the patterns are not considered. Therefore, maximal frequent pattern is neither effective in finding representative patterns nor in eliminating redundant patterns.

Frequent Pattern Mining in Data Streams. Techniques have been proposed to mine frequent patterns in sliding window streams, such as IncSpan [6] and SeqStream [5]. In particular, IncSpan [6] maintains the patterns semi-frequent in the previous window (i.e., patterns that are "almost frequent" in the data) to make the pattern mining incremental when the window moves. SeqStream [5] builds an Inverse Closed Sequence Tree (IST) structure to speed up the update of the frequent patterns. However, these techniques all focus on the traditional frequent pattern semantics and its variation (such as closed frequent pattern). Therefore, they cannot be used to find our MRP patterns.

MDL-based Frequent Pattern Mining in Static Datasets. In [15, 22, 14], the minimum description length (MDL) principle was applied to mine compressive frequent patterns from *static* sequence data. Algorithms that leverage this principle to mine patterns were also designed in these works.

SeqKrimp [15] is a two-step approach. It first generates a set of frequent patterns as candidates using the traditional pattern mining techniques. Then it greedily selects from the candidates a set of patterns that together minimizes the description length. SeqKrimp suffers from two problems. First, SeqKrimp selects representative patterns from the candidates. Therefore the patterns out of the candidates have no chance to be selected even if they are able to reduce the description length. Second, the two-step approach is expensive – especially the pattern candidate generation step. These two problems make SeqKrimp much worse than our SWIFT approaches in both effectiveness and efficiency as shown in our experiments.

Unlike SeqKrimp, GoKrimp [15] directly mines the representative patterns. However, GoKrimp produces patterns only from the most frequent event types. Therefore, it tends to miss the patterns that do not contain super-frequent events. As shown in our experiments, although GoKrimp is much more efficient than SeqKrimp, it is not effective in compressing the sequence due to its oversimplified search heuristic. Similar to GoKrimp, SQS [22] also directly mines the representative patterns from the static sequence dataset. The patterns are constructed iteratively. In each iteration the pattern P is produced, which achieves the largest MDL gain among the possible patterns. Each iteration needs at least one scan of the sequence dataset. Therefore, SQS is extremely slow despite its core shortcoming of also not being as effective as SWIFT.

In [14] the CSC approach was proposed, CSC adopted a strategy that is very similar to the methods proposed in [15]. CSC has two versions: CSC-1 similar to SeqKrimp [15] and CSC-2 similar to GoKrimp [15]. The key difference is that CSC allows overlap among the occurrences of different patterns, while all other methods we compared against in this work and our SWIFT do not allow such overlap. CSC has some strong limitations. First, it does not support patterns that contain any event type E appearing more than once. Second, if two occurrences of a pattern P have a different gap between adjacent events, they consider those two to be different and hence they cannot be compressed. This restricts compression opportunities. As shown in our experiments, CSC is much worse than our SWIFT approaches in both effectiveness and processing time.

Furthermore, the above approaches all deal with static data. In the streaming context, these methods have to mine the patterns from scratch whenever the window moves. Clearly, this is not efficient. In contrast, our SWIFT naturally fits continuously evolving event streams as it only processes each incoming event once.

Complex Event Processing. Complex event processing (CEP) [18, 19, 27] matches continuously incoming events against a given query pattern. Therefore, CEP addresses a 'query' problem, that is, given a query that defines one particular pattern and other constraints, find all its matches in a data stream. Our work instead focuses on a 'mining' problem, namely discovering *all* patterns that are frequent enough based on user-specified parameters.

Episode Mining in Singular Event Sequence. The episode mining problem [17, 16, 26] defines an episode as a set of events that frequently occur together in one single sequence. In [17, 16], efficient approaches were developed to count the number of occurrences for a set of candidate episodes given by the user. Therefore, they solve a *counting* problem similar to CEP, while our work focuses on a mining problem without a candidate pattern set given beforehand. The UP-Span algorithm [26] focuses on finding the high utility episodes from a sequence. The utility of each episode is measured based on the external and internal utilities of each event assigned by the users. Therefore unlike our MRP semantics, the returned episodes do not reflect their ability of succinctly covering a input sequence. Hence UP-Span is not applicable to our problem.

8. CONCLUSION

In this work we propose the SWIFT approach for the effective discovery of representative patterns from the event stream data. SWIFT features MDL-based representative pattern semantics (MRP for short) and a novel continuous pattern mining strategy that processes each new incoming event only once. Our extensive experimental evaluation with real streaming datasets demonstrates the effectiveness of MRP in succinctly summarizing the event stream, and the efficiency of SWIFT in supporting MRP.

9. REFERENCES

[1] C. C. Aggarwal and J. Han, editors. *Frequent Pattern Mining*. Springer, 2014.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14. IEEE, 1995.

[3] A. Barron, J. Rissanen, and B. Yu. The minimum description length principle in coding and modeling. *IEEE Trans. Inf. Theory*, 44(6):2743–2760, 1998.

[4] K. Beedkar, K. Berberich, R. Gemulla, and I. Miliaraki. Closing the gap: Sequence mining at scale. *ACM Trans. Database Syst.*, 40(2):8:1–8:44, June 2015.

[5] L. Chang, T. Wang, D. Yang, and H. Luan. Seqstream: Mining closed sequential patterns over stream sliding windows. In *ICDM*, pages 83–92. IEEE, 2008.

[6] H. Cheng, X. Yan, and J. Han. Incspan: incremental mining of sequential patterns in large database. In *SIGKDD*, pages 527–532. ACM, 2004.

[7] P. Fournier-Viger, C.-W. Wu, A. Gomariz, and V. S. Tseng. Vmsp: Efficient vertical mining of maximal sequential patterns. In *CAIAC*, pages 83–94, 2014.

[8] P. Fournier-Viger, C.-W. Wu, and V. S. Tseng. Mining maximal sequential patterns without candidate maintenance. In *ADMA*, pages 169–180. Springer, 2013.

[9] M. N. Garofalakis, J. Gehrke, and R. Rastogi, editors. *Data Stream Management - Processing High-Speed Data Streams*. Data-Centric Systems and Applications. Springer, 2016.

[10] A. Gomariz, M. Campos, R. Marin, and B. Goethals. Clasp: An efficient algorithm for mining frequent closed sequences. In *PAKDD*, pages 50–61. Springer, 2013.

[11] P. D. Grünwald. *The minimum description length principle*. MIT press, 2007.

[12] P. D. Grünwald, I. J. Myung, and M. A. Pitt. *Advances in minimum description length: Theory and applications*. MIT press, 2005.

[13] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1):55–86, 2007.

[14] A. Ibrahim, S. Sastry, and P. S. Sastry. Discovering compressing serial episodes from event sequences. *Knowl. Inf. Syst.*, 47(2):405–432, 2016.

[15] H. T. Lam, F. Mörchen, D. Fradkin, and T. Calders. Mining compressing sequential patterns. *Stat. Anal. Data Min.*, 7(1):34–52, Feb. 2014.

[16] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *KDD*, pages 410–419, 2007.

[17] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

[18] Y. Mei and S. Madden. Zstream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*, pages 193–206, 2009.

[19] Y. Qi, L. Cao, M. Ray, and E. A. Rundensteiner. Complex event analytics: online aggregation of stream sequence patterns. In *SIGMOD*, pages 229–240, 2014.

[20] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[21] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[22] N. Tatti and J. Vreeken. The long and the short of it: summarising event sequences with serial episodes. In *SIGKDD*, pages 462–470. ACM, 2012.

[23] D. Wang, E. A. Rundensteiner, and R. T. Ellison, III. Active complex event processing over event streams. *PVLDB*, 4(10):634–645, 2011.

[24] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *ICDE*, pages 79–90. IEEE, 2004.

[25] J. Wang, J. Han, and C. Li. Frequent closed sequence mining without candidate maintenance. *TKDE*, 19(8), 2007.

[26] C. Wu, Y. Lin, P. S. Yu, and V. S. Tseng. Mining high utility episodes in complex event sequences. In *KDD*, pages 536–544, 2013.

[27] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[28] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *SIGKDD*, pages 344–353. ACM, 2004.