

SWIFT: Mining Representative Patterns from Large Event Streams

Yizhou Yan¹, Lei Cao², Samuel Madden² and Elke A. Rundensteiner¹

¹Department of Computer Science, Worcester Polytechnic Institute

²CSAIL, Massachusetts Institute of Technology

¹{yyan2,rundenst}@cs.wpi.edu ²{lcao,madden}@csail.mit.edu

ABSTRACT

Event streams generated by smart devices common for modern Internet of Things applications must be continuously mined to monitor the behavior of the underlying system. Given the complexity of existing stream pattern mining strategies, we now adopt pattern semantics known to lead to compact pattern encodings and adapt them for the first time to dynamic data streams, called MDL-based Representative Patterns (MRP). We then design a one-pass SWIFT algorithm that continuously mines the up-to-date MRP pattern set for each stream window upon the arrival or expiration of each individual event. 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. Our experimental evaluation demonstrates the effectiveness of our SWIFT solution in capturing representative patterns in real-world event streams and its efficiency in scaling to high velocity event streams for a rich variety of data sets – up to 2 orders of magnitude faster than the state-of-the-art approaches.

1 INTRODUCTION

Motivation. Over the past decade, smart phones, tablets, and 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, 12] is more critical than ever. As an example, Philips Lighting Research with whom we have worked closely to deploy knowledge mining technology is interested in finding the set of sequential patterns characterizing the lighting control messages exchanged between their smart lighting devices and servers in the cloud. These patterns then could be utilized to determine whether the devices perform as expected based on their configuration or perhaps are in an abnormal state.

State-of-the-Art and Limitations. Existing stream pattern mining techniques [5, 6], which detect sequential patterns from streams using traditional mining semantics [2, 18], suffer from the *pattern explosion* problem [2, 18]. That is, they tend to produce a set of frequent patterns that is huge, highly redundant, and difficult to understand. For example, in our project with Philips Lighting Research, we found that *SeqStream* [5] using the closed frequent pattern semantics [18] produces 1,082 patterns under a standard configuration of parameters, even though there are only 13 distinct message types in 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 a valid representation of the way alarms are reported in the log files. This makes the existing techniques neither effective nor efficient at supporting online applications. These techniques have quadratic or worse CPU and memory complexity, resulting in prohibitively large response time and excessive memory consumption for large data sets. Worse yet, human operators are overwhelmed when having to examine such large pattern sets.

For this reason, in [14, 17], a compact set of non-overlapping patterns is selected out of all possible patterns based on the minimum description length (MDL) principle [15] widely used in text compression. This set of patterns, if used as a dictionary, can be used to maximally compress the data into a compact pattern description. However, this work uses a calculation method for assessing the description length customized for text compression. This penalizes long patterns, i.e., it increases the possibility of missing long patterns, as we illustrate in Sec. 3. Also, the problem of selecting a set of non-overlapping patterns based on MDL is NP-hard. The solutions in [14, 17] are either too expensive because they repeatedly scan the data or yield a poor compression rate by using an over-simplified mining heuristic. Due to these issues, MDL has not previously been leveraged in the streaming context.

Proposed Approach. In this work, we propose the first approach that efficiently finds representative patterns in sliding window event streams using the MDL principle, called SWIFT¹. The key contributions include:

- New stream pattern mining semantics that leverage the MDL principle to model a compact set of representative patterns, called *MDL-based Representative Patterns (MRP)*.
- The incremental SWIFT algorithm that efficiently discovers the set of representative patterns that match the MRP semantics using an incremental strategy. SWIFT, as one-pass solution that touches each new arrival event only once, is well-suited to streaming data. The complexity of the pattern update operation triggered by each new event arrival is bounded by the ratio of the number of the distinct patterns to the number of event types. This allows SWIFT to scale to large volume event streams. We show that SWIFT selects 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

¹Sliding Window-based Frequent patterns

arrives, B-SWIFT adopts a *lazy update* strategy that refreshes the MDR pattern set only once per window. We show that B-SWIFT only conducts update operations guaranteed to cause a change in the MRP pattern set representing the current window.

- Our experimental study demonstrates the effectiveness of SWIFT at discovering representative patterns, providing a 2 order of magnitude speedup versus alternative approaches on two real world event stream datasets.

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 is applied to specify a finite subset of the most recent events from the event streams. The size of the sliding window is denoted as w . A **window sequence** (or **sequence**) is a list of ordered events falling in the current window at time t_i , denoted as $S = \langle (e_{i-w+1}, t_{i-w+1}), (e_{i-w+2}, t_{i-w+2}), \dots, (e_i, t_i) \rangle$. The window slides when a new event (e_{i+1}, t_{i+1}) is received. The new event is inserted into the 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 also move after a certain *slide size* time interval k . The new events produced after t_i are added to the window, while the events produced before $t_{i-w+k+1}$ are removed.

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

We say a pattern $Q = \langle E'_1 E'_2 \dots E'_l \rangle$ is a **sub-pattern** of a pattern $P = \langle E_1 E_2 \dots E_m \rangle$ ($l \leq m$), denoted $Q \subseteq P$, if integers $1 \leq i_1 < i_2 < \dots < i_l \leq m$ exist such that $E'_1 = E_{i_1}, E'_2 = E_{i_2}, \dots, E'_l = E_{i_l}$. 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 [15] is widely used [3, 10, 11] in data compression as a measure to select the encoding model that best compresses the data. 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 [14, 17], 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. This pattern set is considered as the best, because it lets analysts understand the key features of the sequence dataset with a minimum amount of information.

For example, we have a sequence segment $S = \langle (s, 1)(l, 2)(a, 3)(s, 4)(l, 5)(a, 6)(s, 7)(l, 8)(s, 9)(l, 10) \rangle$ extracted from the lighting application shown in Fig. 1. Let us use the traditional pattern semantics [2] to mine the frequent patterns that consider a pattern as


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.

frequent if its frequency is larger than a predefined *support* threshold. When the support threshold is set to 2, it 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 (1,2), (a,3), (l,5), (a,6) are used in the occurrences of both P_1 and P_2 . Or put differently, the occurrences of P_1 and P_2 overlap with each other.

Then we could construct the pattern set $\mathbb{P}_1 = \{P_1 : SLA, P_2 : SL\}$. Each pattern in \mathbb{P}_1 uses 2 occurrences and these occurrences do not overlap with each other. Or we could construct a pattern set $\mathbb{P}_2 = \{P_2 : SL\}$ that contains only one pattern $\langle SL \rangle$ but with 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 ease of presentation.

Based on the calculation rule of 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 streams based on the MDL principle.

Adapting MDL. In MDL, the description length of the *dictionary* $L(\mathbb{P}_i)$, which is part of the overall description length $L(M_i)$, 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 $\langle SL \rangle$ outweighs \mathbb{P}_1 that contains two patterns $\langle SLA \rangle$ and $\langle SL \rangle$, because the description length of \mathbb{P}_1 ($L(\mathbb{P}_1) = 7$) is much larger than the description length of \mathbb{P}_2 ($L(\mathbb{P}_2) = 3$). This causes it to miss the longer $\langle SLA \rangle$ pattern corresponding to the behavior of the lighting devices' alarm reporting. However, summarizing system behavior with the longest possible repeating patterns is intuitively preferable to using shorter patterns, because long patterns can model complex system behaviors.

For this reason, we now slightly alter MDL to use the *number of patterns* in the dictionary \mathbb{P}_i as $L(\mathbb{P}_i)$. With the revised MDL principle, now $L(\mathbb{P}_1) + L(S'_1) = 2+4 = 6$, while $L(\mathbb{P}_2) + L(S'_2) = 1+6 = 7$. Therefore, \mathbb{P}_1 is now selected as expected.

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

- (1) Given $\forall O_{P_i}^S$ of $P_i \in \mathbb{P}$ and $\forall O_{P_j}^S$ of $P_j \in \mathbb{P} (j \neq i)$, if event $e \in O_{P_i}^S$, then $e \notin O_{P_j}^S$;
- (2) $\forall P_i \in \mathbb{P}, L(P_i) > 1 \ \& \ num(P_i, S) > 1$;

(3) Given a pattern $P_i \in \mathbb{P}$, \forall adjacent events (e_x, t_x) and $(e_y, t_y) \in O_{P_i}^S$, $t_y - t_x - 1 \leq \text{eventGap}$.

Def. 3.1, Condition (1) requires that occurrences of the patterns in \mathbb{P} do not overlap with each other. Condition (2) requires that each pattern P_i in \mathbb{P} has at least two occurrences ($\text{num}(P_i, S) > 1$) and the length of P_i is at least 2 ($L(P_i) > 1$). This excludes trivial patterns from \mathbb{P} such as the singleton patterns with only one event type or a pattern that corresponds to the whole sequence S itself. Condition (3) corresponds to the widely adopted event gap constraint [4]. That is, the gap between any two adjacent events in a pattern occurrence cannot be larger than the *eventGap* threshold.

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

Definition 3.2. Stream MRP Mining Semantics. Given a event streams S , continuously mine the MRP (Def. 3.1) from the sequence in the current window of S .

Problem Complexity. To discover the optimal set of MDL-based representative patterns from a given input sequence, we have to (1) first extract all patterns with at least two occurrences, and (2) then identify a subset of these patterns that best compresses the sequence. The first step solves the sequential pattern mining problem, and has been shown to be NP-hard [20]. The second step solves the Data Compression by Textual Substitution (DCTS) problem, and is also NP-hard [16]. Therefore, the MRP mining problem is NP-hard. Hence, what is required is an efficient heuristic strategy designed to meet the response time requirements of online applications.

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

4 SWIFT: MINING MDL-BASED REPRESENTATIVE PATTERN SET

In this work, instead of using a two-step approach, our SWIFT approach directly mines the set of MRPs *in one step*. Further, SWIFT is a one-pass strategy that processes each incoming event only once upon its arrival. We prove that SWIFT, while lightweight, always chooses the pattern update operation for each incoming event that most reduces the description length of the sequence in the current window.

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 (MG)	Singletons or patterns that could be merged to form new patterns
Match Candidate (MC)	The 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 a new event arrives,

SWIFT first checks if the current window is full. The earliest event in the current window will be removed from the current window. This then triggers the *expire* operation. The *insert* operation is triggered to process the new event. Tab. 1 summarizes the meta-data structures maintained by SWIFT for the window of the event stream. They are used to accelerate both *insert* and *expire* operations. The details 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. Given an incoming event, the *Insert* operation updates the pattern set. Insert classifies the update into two categories, namely “merge” and “match” based on whether a new pattern will be produced. This allows us to design distinct 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 4.3.

Given a new event e_i of type E_i , e_i is handled in one of two ways. The first case is that it 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 case, e_i could create *match* to one of the *current patterns* after merging with some previous singleton events or pattern occurrence 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, i.e., the one that most reduces the minimum description length of the current window sequence.

Next, we introduce the two key processes of insert operation, namely *merge pair identification* and *match pattern identification*. Customized strategies are designed to *efficiently* support them.

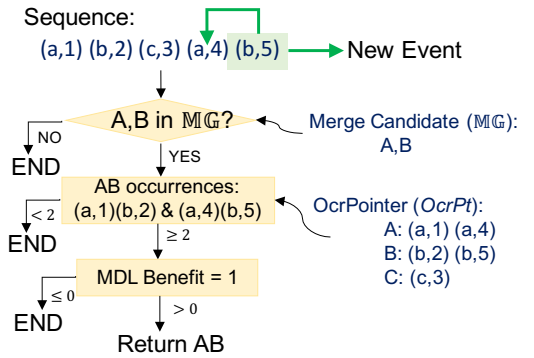


Figure 2: Merge Pair Identification

4.1.1 Merge Pair Identification

Given an incoming event e_i of type E_i , the *element* E_j (either a pattern or a singleton event type) that can be merged with E_i is selected with the assistance of the *merge candidate* (MG) and *occurrence pointer* (ocrPt) meta data structures.

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 1 Merge Pair Identification.

```

1: function FINDMERGE(ELEMENT  $e_i$ , METADATA meta)
2:   mergePair.benefit =  $-\infty$ 
3:   if  $ocrPt.get(e_i).size > 1$  then
4:     for  $e_j \in \mathbb{MG}$  do
5:       if  $e_j - e_i - 1 > eventGap$  then
6:         return
7:       benefit = COMPUTEMDL( $e_i, e_j, meta$ )
8:       if benefit  $> 0$  & benefit  $> mergePair.benefit$  then
9:         mergePair.pair = ( $e_i, e_j$ )
10:        mergePair.benefit = benefit
11:    $\mathbb{MG}.add(e_i)$ 
12:   return mergePair

```

Alg. 1 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. 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 $(b, 5)$ arrives. Since $(b, 5)$ can merge with the event $(a, 4)$ 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 $(a, 1)$ and $(a, 4)$. Event type B has two occurrences $(b, 2)$ and $(b, 5)$. 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 $(a, 1)(b, 2)$ and $(a, 4)(b, 5)$. 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 $(b, 5)$ has the opportunity to form a valid occurrence of any pattern with $(b, 5)$. Therefore no more merge pair can be produced. $\langle A, B \rangle$ is selected as the *best* merge pair.

Similarly, after the event $(c, 6)$ arrives, a merge pair $\langle P_1, C \rangle$ is generated, where $P_1 = \langle AB \rangle$. The benefit is 2. After merging C and P_1 , we get a new pattern $P_2 = \langle ABC \rangle$.

4.1.2 Match Pattern Identification

Given an incoming event e_i , e_i can form matches with some existing patterns by merging. Intuitively, this can be done using a method similar to the merge process described above. That is, given an incoming event e_i , we first augment e_i with the item that is directly in front of e_i in the encoded sequence to form an occurrence Op_t of a temporary pattern P_t . Then we examine whether P_t matches with any existing pattern. If it does, Op_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 Op_t . This process stops if Op_t does not match any pattern. Clearly, this process is expensive because of the potentially large number of pattern match operations.

To reduce the computation cost, we propose a match strategy that efficiently identifies the best match for the incoming event. Given an incoming 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 can form valid matches with the current patterns. 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, $[B, P_2(ABC), 1, [7]]$ 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. “[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 an incoming 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 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 table. Using this table, given an event e_i , the existing patterns with E_i as prefix can be obtained in constant time.

Update Match Candidate. Given an incoming event e_i , the match candidates that are expecting event type E_i will transition to a new

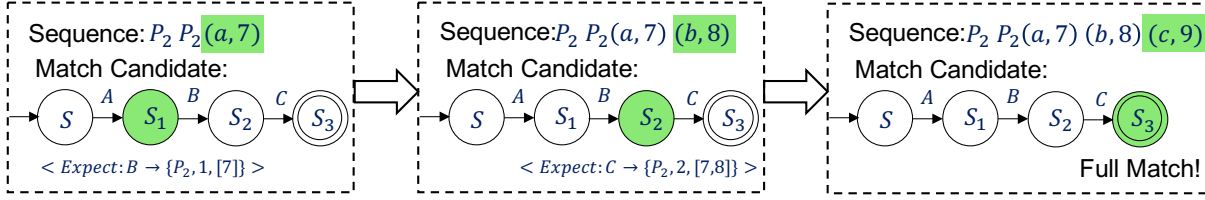


Figure 3: Example of Matching Pattern $\{P_1 : ABC\}$.

state. Specifically, we update the *expected-event* to the event type next to E_i in *match-pattern*. We then insert the position of e_i into the *timestamp* field. For example, given a match candidate MC_i $\{B, P_2(ABC), 1, [7]\}$, after event $(b, 8)$ arrives, MC_i transitions to $\{C, P_2(ABC), 2, [7, 8]\}$. 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 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.

Example 4.2. Fig. 3 shows the match process of pattern $P_2 : \langle ABC \rangle$. 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 $\{B, P_2, 1, [7]\}$ 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 automata. With the arrival of event $(a, 7)$, the automaton transitions from state S to state S_1 . Then event $(b, 8)$ comes next. It transitions the match candidate from state S_1 to state S_2 , which is $\{C, P_2, 2, [7, 8]\}$. No new match candidate is generated in this 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. In this case, it is $2 (\text{len}(P_2) - 1)$.

Avoiding Recursive Updates. Intuitively, to further reduce the description length, the new pattern occurrence e generated by a merge or match update 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 4.3. *Given a new pattern occurrence e produced by insert, at most one additional update operation can be triggered.*

Proof. We prove Lemma 4.3 by showing: (1) if e is produced by *merge*, no further update is required; (2) if e is produced by *match*, e can only trigger at most one additional *merge* update.

Proof of (1): by proving (a) e cannot produce any new match; (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 ???. 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 is 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, because E_j had already been merged to produce either a new pattern or a match with existing pattern. This contradicts the fact that E_j does exist and has been merged with E_i . Therefore, the new element P cannot merge with any other event type E_x again.

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

The Optimality of Insert.

LEMMA 4.4. *Given an incoming event e_i , insert always chose the update operation for e_i that most reduces the MDL score of the current sequence.*

Proof. We prove Lemma 4.4 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 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 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 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 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 finally proves Lemma 4.4. ■

Time Complexity Analysis of Insert. we analyze the amortized time complexity of the insert operation per event. Its complexity is mainly determined by the two processes, namely *merge pair identification* and *match pattern identification*. In the following analysis, let N be the number of elements in current window. Each element represents either one pattern occurrence or a singleton event. Let $|P|$ denote the number of patterns found in current window and l denotes the average length of each pattern. Let $|E|$ denote the number

of event types. We assume each event type arrives independently with the same probability.

Merge Pair Identification. In order to identify the best merge pair for the incoming event, we have to traverse all occurrences of each possible merge pair. Therefore, the time complexity of the merge pair identification depends on two aspects, namely the number of possible merge pairs and the number of occurrences for each merge pair. In the worst case, each event could find at most τ merge pairs ($\tau = \text{eventGap} + 1$). Each element has equal probability to be involved in the merge pair. Then the amortized time complexity of merge pair identification per event is $O(\frac{2 \times N \times \tau}{|E|})$. It is proportional to N and inversely proportional to $|E|$.

Match Pattern Identification. The match pattern identification is composed of two processes, namely the match candidate update and match candidate initialization. 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 initialization process creates a new match candidate for each pattern that starts with E_i . The number of updates and initializations is no more than $|P| \times l$ for $|E|$ event types. Since each event type arrives independently with the same probability, the amortized time complexity of match pattern identification per event is $O(\frac{|P| \times l}{|E|})$. It is proportional to $|P|$ and inversely proportional to $|E|$.

Overall the time complexity of *insert* is bounded by the ratio of N and $|P|$ to $|E|$. Since typically N , $|P|$, and $|E|$ increase together when the data volume increases, the ratio tends to be small. Therefore, SWIFT is scalable to high volume stream data.

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 some pattern.

Expiring a singleton event is straightforward. Since it is not involved in any pattern, it can be simply removed from the meta data structures including the “encoded sequence”, “merge candidate” MR, “match pattern” MC, and “occurrence pointers” *ocrPt*.

Expiring an event used by an occurrence O of some 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 the existing patterns.

The idea of the expire operation is to treat singleton events produced due to event expiration as new events. Expire events are processed one by one, leveraging the insert operation (Sec. 4.1). Unlike the real incoming events which are always 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 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. This is in contrast to the merge process triggered by

the new arrival which searches for merge pair from the early arrivals in a reverse order.

Second, the modification of the *match process* is mainly on the match candidates MC meta data. Originally 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 predict the candidates that could match the existing patterns after merging with e_i . 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 is processed, the patterns with E_i as prefix are inserted into the candidate set. Then when processing the next event e_{i+1} , the candidates which are expecting E_{i+1} are updated to expect the next event type. New match candidates are then constructed corresponding to the patterns with E_{i+1} as prefix. After all these events are processed, this temporary match candidate set is discarded.

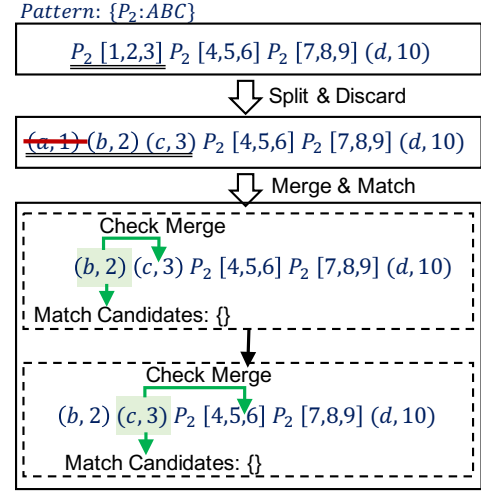


Figure 4: Pattern Expiration.

Example 4.5. In our running example, the sequence is encoded into $\langle P_2[1,2,3]P_2[4,5,6]P_2[7,8,9](d, 10) \rangle$ 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. Since the expiring event $(a, 1)$ is involved in one of the occurrences of pattern P_2 , this occurrence is split into singleton events $(b, 2)$ and $(c, 3)$ which are then treated as new events. Here no match or merge can be formed on the two singletons. Therefore they are simply inserted back to the sequence at their original positions. After expiring the first event, 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 possible in streaming data analytics. In such cases, it is not necessary to update results every time a new event arrives. Instead, results can be 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.

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 results in the split of the other occurrence of P , because P is no longer frequent. However, 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 – only temporarily infrequent. Therefore, it is not necessary to split the other occurrence of P .

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).

5.1 Batch Expire Operation

Instead of expiring each event independently, the batch expire operation (*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 part of 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 part of 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 new incoming batch such that eventually P_s might still be valid. These *to-be-split* patterns will not be handled until processing the new incoming batch.

Example 5.1. Fig. 5 demonstrates the batch expire process of our running example. As shown at the left of Fig. 5, after processing the first 10 events the sequence is encoded as $S = \langle P_2[1,2] P_1[3,4,5] P_1[6,7,8] P_2[9,10] \rangle$, where two patterns $\{P_1 : ACD, P_2 : AB\}$ are 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 occurrence split operations caused by batch expiration by prioritizing the orders of the operations conducted on the new batch.

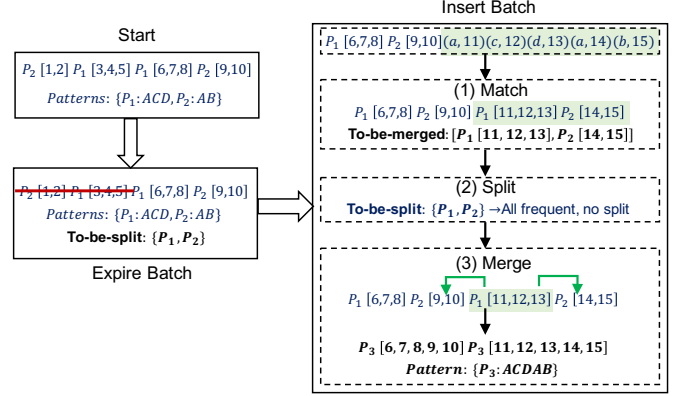


Figure 5: Expire & Insert a Batch.

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 til 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 in the match processes.

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 in the 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 in their arrival order using the merge process described in Sec. 4.1.

In the running example, since there is no singleton 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

We now describe our evaluation of SWIFT.

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/high velocity event streams.

Real Datasets: A *log file dataset* exacted from a mobile application that tracks driver behavior. 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.

A *lighting dataset* produced by our commercial lighting application. 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, which 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 utilized in the literature [5, 6] 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 noises, that is, the events not belonging to any pattern to mimic the real-world data.

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 (Intel(R) Xeon(R) CPU E5-2690 v4), 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 five different systems. We adapt three static methods, namely *SeqKrimp*, *GoKrimp* and *SQS* to the streaming context. (1) *SeqKrimp*: the two-step static pattern mining method of [14]; (2) *GoKrimp*: the one-step static pattern mining method of [14] that directly mines representative patterns; (3) *SQS*: the static pattern mining method of [17] that mines the representative patterns

by recursively scanning the data. More details of the above three algorithms are described in related work; (4) SWIFT: our incremental streaming representative pattern mining approach (Sec. 4); (5) B-SWIFT: enhanced SWIFT to support batch update (Sec. 5). Each time the window slides, the patterns in the new window are mined again. 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, we evaluate their effectiveness by measuring the *average compression rate (ACR)*. 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 each window sequence. *ACR* measures how succinctly the patterns summarize the input sequence data. The lower the rate is, the better the set of representative patterns is. Furthermore, we measure the *processing time* on each window sequence for efficiency evaluation.

6.2 Evaluation of Effectiveness

To demonstrate the effectiveness of SWIFT (Sec. 4) and B-SWIFT (Sec. 5), we evaluate the average compression rate (ACR). Overall, SWIFT and B-SWIFT outperforms the state-of-the-art at ACR 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, namely 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 consequently results in 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 valid representative patterns. 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 forming patterns based on the frequency of each singleton event type. Among the state-of-the-art methods, SQS achieves the best ACR (35%), although still worse than our SWIFT. Its good ACR is achieved via a complex search strategy that requires the iterative scan of the data, resulting in prohibitive execution time for SQS as shown in Sec. 6.3.

Furthermore, as shown in Fig. 6(a), as the window size goes up, 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

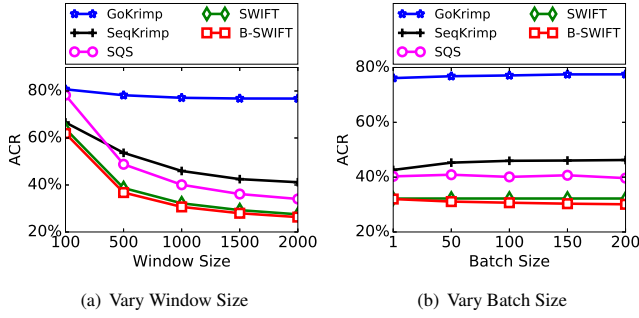


Figure 6: Average Compression Rate on Log File Data.

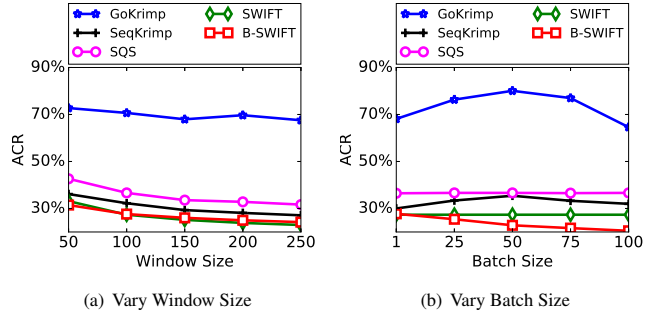


Figure 7: Average Compression Rate on Lighting Data.

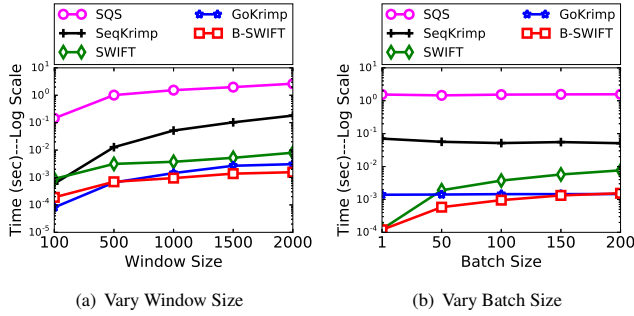


Figure 8: Processing Time on Log File Dataset.

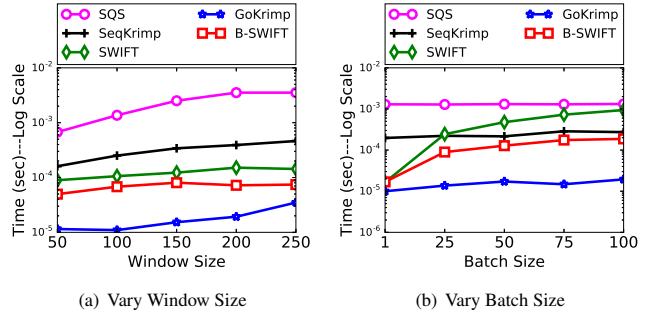


Figure 9: Processing Time on Lighting Dataset.

on all events in the batch. When the batch size is large, the decision made by B-SWIFT is closer to the global optimal solution.

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 other alternatives in 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.

6.3 Evaluation of Efficiency

We investigate the processing time of our SWIFT using the two real datasets by varying the *windowSize* and *batchSize*. Overall, B-SWIFT consistently outperforms SWIFT and the state-of-art approaches by up to 4 orders of magnitude.

Log File Dataset. The parameters are set as *WindowSize*=1000 and *BatchSize*=100 except when they are under variation. As shown in Fig. 8, B-SWIFT consistently outperforms SWIFT, SeqKrimp, 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 search strategy. On the other hand, SQS is much slower than any other method because of its complex iterative search strategy, although it has relatively good ACR compared to SeqKrimp and GoKrimp.

Fig. 8(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 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. 8(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, 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. 8(b), even in the worst case, B-SWIFT is still 30 times faster than SeqKrimp 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 average processing time. This confirms the effectiveness of the *lazy update* strategy of B-SWIFT in eliminating the unnecessary update operations.

Lighting Dataset. Fig. 9 shows the average processing time on the lighting dataset. The parameters are set to *WindowSize* = 100,

$BatchSize = 10$ by default. As shown in Fig. 9, B-SWIFT is consistently faster than other methods except GoKrimp by up to 70x under all circumstances. Although GoKrimp is slightly faster than B-SWIFT on this relatively simple dataset, GoKrimp is not effective in capturing the representative patterns due to its over-simplified search heuristic as demonstrated in Sec. 6.2.

Fig. 9(a) and 9(b) show the processing time as varying window and batch sizes. The trends are similar to those for the log file dataset. In the worst case when the batch size is as large as the window size, B-SWIFT is still 2x and 7x faster than SeqKrimp and SQS.

6.4 Efficiency Evaluation on Synthetic datasets

We use synthetic datasets to evaluate how SWIFT and B-SWIFT perform on data streams with large window and batch sizes. 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*. The results are shown in Fig. 10.

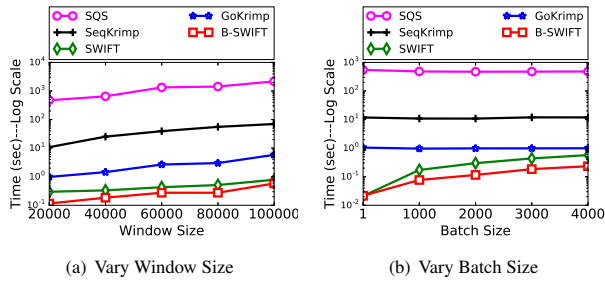


Figure 10: Processing Time on Synthetic Datasets.

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 outperforms 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, the processing time is within one second.

Fig. 10(b) shows the processing time when varying 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 which is confirmed to fail in generating representative patterns, 51x faster than SeqKrimp, and 3 orders of magnitude faster than SQS. This confirms that our SWIFT approaches are scalable to the batch size representing the velocity of the event stream.

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 the pattern explosion problem generating too many redundant patterns. To alleviate this problem, variations of the basic pattern mining semantics [7–9, 19] were proposed. The *closed frequent pattern* semantics [9, 19] 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 *maximum frequent pattern* semantics [7, 8] keeps only the longest frequent patterns. It discards all sub-patterns of P when P is frequent. Therefore, it only handles the redundancy among a pattern and its sub-patterns. However, it blindly discards all sub-patterns of a frequent pattern P . Thus it might miss some representative patterns, since sometimes P and its sub-patterns might both be representative. Therefore, the redundant pattern problem cannot be solved by simply applying these hard-coded rules.

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 semi-frequent patterns 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. In [13], *Zips* was proposed to incrementally mine patterns from event stream. However, it overlooks the event expiration problem – essential for stream pattern mining. Each time when a new batch of events arrives, *Zips* recursively selects one pattern from the existing pattern set that best covers the remaining events in the new batch. This pattern is then extended by appending the next singleton event right after it. *Zips* has severe drawbacks. First, it relies on a continuously maintained pattern set to process new batch. Therefore, supporting a sliding window stream by repeatedly applying *Zips* on each new window starting from an empty pattern set is impractical. Second, it does not evict any pattern until hitting the memory limit. Therefore, it tends to generate a large number of patterns and hence cannot solve the pattern explosion problem.

MDL-based Frequent Pattern Mining in Static Datasets. In [14, 17], the minimum description length (MDL) principle was applied to mine frequent patterns from *static* sequence data. These approaches solve a different problem from ours, i.e. using MDL to produce a compact compression such that the original sequence data can be exactly recovered from it. As shown in Sec. 3 this is not ideal in finding representative patterns.

Pattern mining algorithms were also designed in these works. SeqKrimp [14] 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 minimize 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 further 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 (Sec. 6).

Unlike SeqKrimp, GoKrimp [14] 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 the oversimplified search heuristic. Similar to GoKrimp, SQS [17] also directly mines the representative patterns from the static sequence dataset. The patterns are constructed iteratively. In each iteration one 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 that it is not as effective as SWIFT.

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. On the contrary, our SWIFT naturally fit continuously evolving event streams as it only processes each incoming event once.

8 CONCLUSION

In this work we propose the SWIFT approach for the effective discovery of representative patterns from 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.

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] 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.
- [10] P. D. Grünwald. *The minimum description length principle*. MIT press, 2007.
- [11] P. D. Grünwald, I. J. Myung, and M. A. Pitt. *Advances in minimum description length: Theory and applications*. MIT press, 2005.
- [12] 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.
- [13] H. T. Lam, T. Calders, J. Yang, F. Mörchén, and D. Fradkin. Zips: mining compressing sequential patterns in streams. In *IDEA, Chicago, Illinois, USA, August 11, 2013*, pages 54–62.
- [14] H. T. Lam, F. Mörchén, D. Fradkin, and T. Calders. Mining compressing sequential patterns. *Stat. Anal. Data Min.*, 7(1):34–52, Feb. 2014.
- [15] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.
- [16] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.
- [17] 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.
- [18] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *ICDE*, pages 79–90. IEEE, 2004.
- [19] J. Wang, J. Han, and C. Li. Frequent closed sequence mining without candidate maintenance. *TKDE*, 19(8), 2007.
- [20] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *SIGKDD*, pages 344–353. ACM, 2004.