MULTI-OBJECTIVE SEQUENTIAL PATTERN MINING OVER TRANSACTION DATA STREAMS BASED ON A SLIDING WINDOW MODEL

Tran Minh Thai*, Tran Anh Duy, Le Thi Minh Nguyen, Pham Duc Thanh

Faculty of Information Technology, Ho Chi Minh City University of Foreign Languages – Information Technology

thaitm@huflit.edu.vn, duyta@huflit.edu.vn, nguyenltm@huflit.edu.vn, thanhpd@huflit.edu.vn

ABSTRACT— Sequential pattern mining plays an essential role in numerous practical domains such as customer behavior analysis, medical diagnosis, and cybersecurity monitoring. Although previous studies have effectively addressed the problem of discovering frequent patterns or high utility patterns in static databases, the rapid growth of continuously updated data streams has raised critical challenges regarding processing latency and storage limitations. To overcome these obstacles, this paper proposes the Multiple Objective Stream Sequential Pattern Mining algorithm (MOS-SPM) that operates on a specialized linked list structure (MOS-SPM-LL) based on a sliding window model. This approach enables the system to simultaneously evaluate three core criteria, including support, high utility, and regularity of patterns, directly over a transaction stream. For comprehensive evaluation, the study purposefully extends two state-of-the-art algorithms, CHUSP and RStreamHUSP, as comparison baselines. Experimental results demonstrate that the proposed algorithm preserves absolute correctness when compared with the extended CHUSP version. Moreover, owing to its lightweight storage mechanism, MOS-SPM exhibits superior performance in both execution speed and memory optimization compared with the extended RStreamHUSP version.

Keywords— Sequential pattern mining, data stream, high utility pattern, regular pattern, sliding window.

I. INTRODUCTION

In the digital era, sequence data mining is a critical technique for analyzing transaction histories, managing customer relationships [1], identifying trends in text databases [2], monitoring climate change [3], and analyzing web browsing behavior [4]. Early methods such as PrefixSpan [5] and SPADE [6] focused solely on occurrence frequency, making them prone to overlooking rare patterns that have high practical value. To address this limitation, the research direction of high utility pattern mining emerged with algorithms such as UMSP [7], USPAN [8], and CHUSP [9].

However, these methods primarily operate on static databases. In practice, data are often generated continuously in the form of streams, demanding fast processing speeds and strict memory constraints. Additionally, a pattern with high utility but intermittent occurrence is difficult may be difficult to use for long-term planning. Maintaining a set of patterns that simultaneously satisfies support, utility, and regularity requirements over data streams remains a challenging problem.

This paper proposes a new approach based on a sliding window model with three main contributions:

- MOS-SPM-LL structure: A multi-objective linked list that efficiently stores and updates the characteristics of sequential transaction streams.
- MOS-SPM algorithm: A mining solution that maintains a pattern set simultaneously satisfying three thresholds for support, utility, and regularity.
- Experimental evaluation: Extending two algorithms, CHUSP [9] and RStreamHUSP [10], as benchmarks for comparing performance in terms of execution time and memory consumption.

The remainder of this paper is organized as follows. Section II reviews related work. Section III presents the theoretical foundation. Section IV describes the proposed method. Section V analyzes the experimental results. Section VI concludes the paper.

II. RELATED WORK

The foundation of the data mining field was laid by the pioneering Apriori algorithm [11]. Technically, this method introduced the anti-monotone property that allows the system to prune early the candidates that do not satisfy the minimum support threshold, thereby significantly reducing the cost of exploring the search space. Inheriting this pruning philosophy, the sequential pattern mining problem was initially developed with foundational algorithms such as PrefixSpan [5] and SPADE [6]. PrefixSpan employs prefix-projected pattern growth to reduce the data structure, while SPADE applies a vertical data format to count support through set intersection operations. Subsequently, the CM-SPADE method [12] improved SPADE by exploiting co-occurrence information to accelerate the traversal process. A comprehensive survey of the sequential pattern mining field is presented in detail in [13]. Despite improving processing speed, these methods share the common limitation of assuming that all items carry equal importance. Ignoring real-world item weights causes the system to easily miss rare patterns that have high economic value.

To address the weighting issue, the problem of high utility pattern mining was proposed. Early-generation algorithms such as UMSP [7] and USPAN [8] constructed specialized tree structures to maintain utility information during the pattern traversal process. In recent years, this research direction has continued to attract attention with numerous search space optimization techniques. The CHUSP algorithm [9] was introduced with a mechanism that combines utility and support to extract closed pattern sets, thereby reducing output data redundancy. Similarly, the CUASPM algorithm [14] was introduced to address pattern combinability in complex sequences. Although the pruning capability has been significantly improved, the core limitation of these solutions is that they are primarily designed for static databases and require a full rescan of the dataset when new events arise.

When applied to continuously updating data environments, the sliding window model becomes the dominant technical solution. The HUPMS algorithm [15] pioneered the use of a tree structure combined with a sliding window for storing transactions. Recently, many new studies have focused on improving flexible adaptability over streams. The HSLM method [16] optimized the update mechanism to accelerate the window sliding process. Additionally, solutions such as StreamHUSP [17] allow maintaining the pattern set directly without recomputation from scratch. Although these methods solve the dynamic update problem, they have not yet considered the temporal regularity of pattern distribution.

A pattern with high utility but intermittent occurrence will pose difficulties for strategic planning. To supplement the gap constraint, the concept of regular pattern mining was introduced [18]. The RStreamHUSP algorithm [10] successfully integrated both utility and regularity factors over a sliding window based on maintaining counting arrays for comparison. However, this method applies an extremely aggressive pruning strategy. Specifically, if a pattern does not appear in the data batches of the current window, leading to a violation of the gap threshold, that pattern is immediately removed from the storage structure. Although this mechanism helps free memory temporarily, it entails a significant drawback: the system must incur computational costs to regenerate candidates and reinitialize counting arrays from scratch if that pattern reappears in subsequent batches.

Notably, to date, no algorithm has been capable of simultaneously mining all three criteria, including support, high utility, and regularity, in a data stream environment. The complete integration of these three factors yields tremendous practical significance. For instance, in retail analytics, a product group must not only deliver good profit margins and be chosen by many customers but also maintain a stable consumption rhythm across periods to support supply chain management. If any one aspect is missing, managers may make unsustainable decisions. Recognizing this research gap and practical need, the method proposed in this paper inherits the early pruning principle of Apriori [11] and uses a linked list structure to unify all three mining objectives, aiming to reduce storage requirements and overcome processing latency when sliding over data streams.

III. THEORETICAL FOUNDATION

Let $I = i_1, i_2, \dots, i_n$ be a set of distinct items. An itemset X is a non-empty subset of I. A sequence $s = \langle X_1; X_2; \dots; X_m \rangle$ is defined as an ordered list of itemsets. A sequence $s_a = \langle a_1; a_2; \dots; a_m \rangle$ is a subsequence of sequence $s_b = \langle b_1; b_2; \dots; b_n \rangle$, denoted $s_a \subseteq s_b$, if there exist indices $1 \le k_1 < k_2 < \cdots < k_m \le n$ such that $a_1 \subseteq b_{k_1}, a_2 \subseteq b_{k_2}, \dots, a_m \subseteq b_{k_m}$ [11].

The sequential pattern search process is performed through two basic extension operations based on a prefix sequence s and an item e [5]. The itemset extension (I-extension) appends item e to the last itemset of s, producing a new sequence $s' = \langle X_1; X_2; \dots; X_m \cup e \rangle$. The sequence extension (S-extension) appends item e as a new itemset at the end of s, producing a new sequence $s' = \langle X_1; X_2; \dots; X_m; e \rangle$.

Each transaction sequence in the stream database is assigned a unique identifier sid. The set of all transaction sequences containing pattern s is called the support set of s. The support of a sequence s in database SDB, denoted $sup(s)$, is the total number of transaction sequences that contain s. Pattern s is called a frequent pattern if its support is greater than or equal to the user-specified minimum threshold *minSup*.

$$sup(s) = |\{Sq \in SDB \mid s \subseteq S_q\}| \ge minSup \quad (1)$$

Regarding the value aspect, each item i carries an internal utility $iu(i)$ representing the quantity in a specific transaction and an external utility $eu(i)$ representing the importance or unit price. The utility of an item is $u(i) = iu(i) \times eu(i)$. The utility of a sequence s is the sum of utilities of all its constituent items. Since sequence s may occur at multiple positions within a transaction sequence S_q, the system selects the maximum utility value as the representative [8]. The selection of the maximum value instead of the sum of all occurrences ensures that the utility upper bound property is maintained for valid pruning during the pattern traversal process [8]. The total utility of pattern s over the entire database is computed by the formula:

$$u(s) = \sum_{Sq \in SDB \wedge s \subseteq Sq} \max\{u(s') \mid s' \text{ is an occurrence of } s \text{ in } S_q\} \quad (2)$$

Pattern s is recognized as a high utility pattern if $u(s)$ is greater than or equal to the minimum utility threshold *minUtil*. This computation is illustrated through Table 1 containing fixed unit prices and Table 2 describing the transaction sequences. In Table 2, the internal utility (quantity) of each item varies randomly and is recorded inside square brackets.

Table 1. *External utility values (unit prices) of items*

Item	**a**	**b**	**c**	**d**	**e**
External utility (eu)	5	8	10	7	4

Table 2. *Example of a sequence database with varying quantities*

Identifier (sid)	**Transaction sequence**	**Total utility**
1	$< (a[2], c[1], d[3]); (a[1], d[2]); a[2] >$	70
2	$< (a[3], b[2]); c[4] >$	71
3	$< (b[1], c[2]); a[3] >$	43
4	$< (b[2], d[1], e[3]); d[2] >$	49
5	$< (a[1], b[1], c[3]); a[2] >$	53
6	$< (a[2], c[1], e[2]); (c[2], d[1]) >$	55

In a data stream environment, the system uses a sliding window W consisting of consecutive data batches with identifiers starting from sid_{start} and ending at sid_{end} [10]. Suppose pattern s appears in the transaction sequences within W at the list of identifiers $L = \langle sid_1, sid_2, \dots, sid_k \rangle$ in ascending order. The occurrence gap of a pattern is the difference between two consecutive identifiers. The regularity of sequence s in window W, denoted $reg(s, W)$, is the maximum value among the gaps between consecutive occurrences, including the gaps to both window boundaries [19].

$$reg(s, W) = \max\left\{sid_1 - sid_{start}, \max_{1<j\leq k}\left(sid_j - sid_{j-1}\right), sid_{end} - sid_k\right\} \quad (3)$$

Pattern s is called a regular pattern if $reg(s, W) \leq maxReg$, where *maxReg* is the maximum allowed gap threshold. As an example using Table 2, if the window spans from $sid_{start} = 1$ to $sid_{end} = 6$, pattern s = ⟨a⟩ appears at L=⟨1, 2, 3, 5, 6⟩. The measured gaps are (1−1 = 0, 2−1 = 1, 3−2 =1, 5−3 = 2, 6−5 = 1, 6−6 = 0). Therefore, $reg(\langle a \rangle, W) = 2$.

Given a continuously arriving sequential data stream, a sliding window W with thresholds *minSup*, *minUtil*, and *maxReg*, the problem of multi-objective sequential pattern mining over streams is the process of finding all sequences s in the current sliding window W that simultaneously satisfy three conditions:

$$sup(s) \geq minSup, \quad u(s) \geq minUtil, \quad reg(s, W) \leq maxReg \quad (4)$$

IV. PROPOSED METHOD

This section presents in detail the system architecture, storage structure, pruning properties with proofs, and the algorithmic solutions proposed to solve the multi-objective sequential pattern mining problem over streams.

A. STORAGE STRUCTURE AND SLIDING WINDOW MODEL

To process an unbounded data stream, the proposed method employs a batch-based sliding window model. The event stream is packaged into fixed-size data batches. When a new data batch arrives, the window advances forward, and simultaneously the oldest batch at the trailing boundary is removed from memory. Figure 1 visually illustrates this sliding process.

As shown in Figure 1, the sliding window mechanism operates in two phases. At time t_1, window W_1 covers three consecutive batches B_1, B_2, *and* B_3. When a new batch B_4 arrives at time t_2, the window slides forward to form $W_2 = \{B_2, B_3, B_4\}$: the expired batch B_1 is removed from memory while B_4 is incorporated. This process ensures the system always processes only the most recent data within the fixed window size.

To record quantitative transaction data, the study proposes the multi-objective linked list structure MOS-SPM-LL. Each node stores five components: an item identifier, arrays containing transaction identifiers (sids) and corresponding utilities (utils), parent pointers, hash tables for lookup, and a singly linked list pointer. Figures 2 and 3 present the detailed linkage.

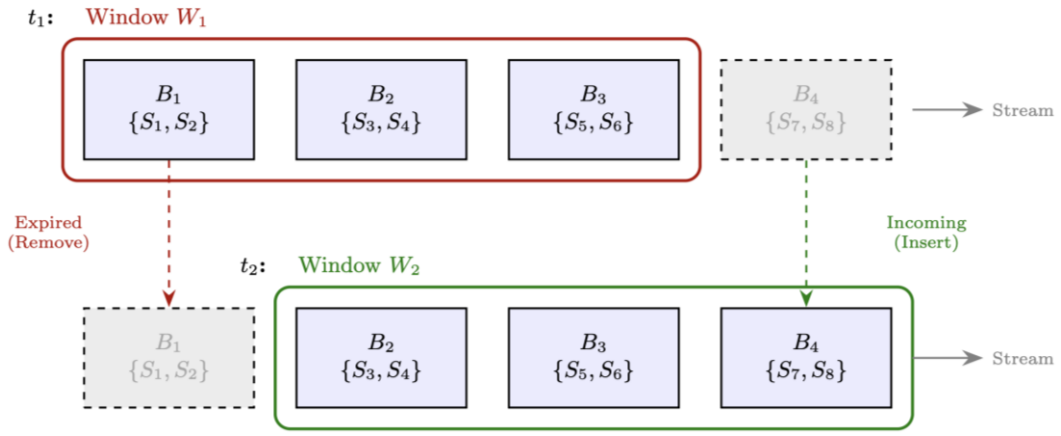

Figure 1. *Illustration of the sliding window mechanism from time t1 to t2.*

It is important to note that the utils array stores accumulated utility values rather than the exact utility of the pattern itself. Specifically, for each transaction identifier sid in the sids array, the corresponding value in utils includes the utility of the current pattern plus the remaining utility of items appearing after it in the transaction sequence. This design allows the value $\sum utils$ to serve as a utility upper bound for all extensions, supporting the pruning mechanism presented in Property 1.

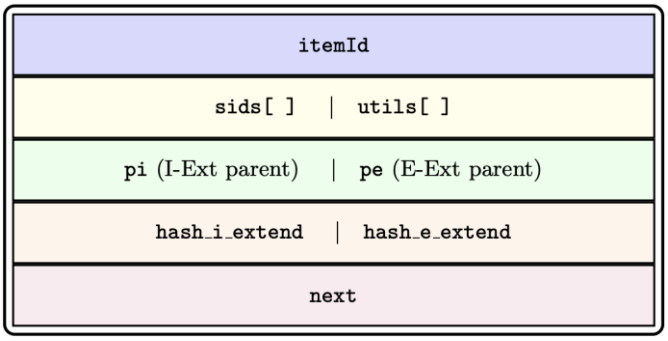

Figure 2. *Structure of a MOS-SPM-LLNode.*

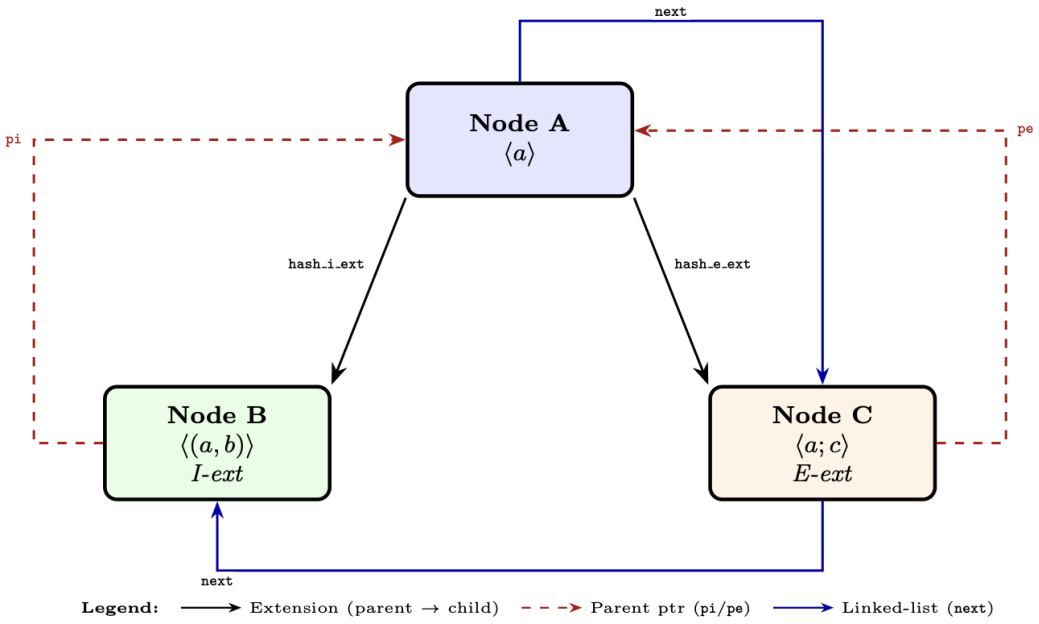

Figure 3. *Linkage among nodes in memory.*

Figure 2 details the five-component layout of each MOS-SPM-LLNode. The top field itemId stores the item identifier. The data layer contains two parallel arrays, sids and utils, recording the transaction identifiers and their corresponding accumulated utility values. The pointer layer holds two parent references, pi (I-extension parent)

and pe (S-extension parent), used for skip-path backtracking. The hash layer maintains two hash tables, hash_i_extend and hash_e_extend, enabling constant-time lookup of child nodes. Finally, the next pointer connects the node into the global singly linked list.

It is important to clarify how the MOS-SPM-LL structure enforces sequential ordering. The parent pointer type encodes the positional relationship between items at construction time (Algorithm 2): an item added via a pi pointer co-occurs in the same itemset as its predecessor (I-extension), while an item added via a pe pointer begins a new, later itemset (S-extension). As a result, the sids array of each node contains only those transaction identifiers in which the item appears in the structurally correct sequential position relative to its parent. During the mining phase (backtrackTravel), intersections operate exclusively on these pre-validated arrays; no additional positional check is required because the pi/pe parent pointer structure already guarantees that every generated pattern respects the correct I-extension and S-extension ordering semantics.

Figure 3 illustrates the interconnection among three nodes in memory. Node A represents the root pattern $\langle a \rangle$. Through hash_i_ext, Node A points to Node B, which stores the I-extension pattern $\langle (a, b) \rangle$; through hash_e_ext, it points to Node C, which stores the S-extension pattern $\langle a; c \rangle$. Conversely, Node B uses parent pointer pi and Node C uses pe to trace back to their common parent Node A, forming the skip-path mechanism. Additionally, all three nodes are chained together through next pointers ($A \rightarrow C \rightarrow B$), enabling sequential traversal of the global linked list.

The overall orchestration of the stream mining algorithm is established through Algorithm 1. When a new data batch arrives, the system updates the linked list and then triggers the pattern search tree traversal.

Algorithm 1 Multi-objective mining orchestration framework MOS-SPM
Input: Data stream, window W , *minSup*, *minUtil*, *maxReg*
Output: Pattern set P

while a new data batch Bnew is received **do**
  Evict the oldest batch Bold from window W
  Call Construct-MOS-SPM-LL(B_{new} , B_{old}) *// Algorithm 2*
  Initialize result set P ←∅
  for each root node Nroot in the global linked list **do**
    Call MOS-Mining(Nroot, P , *minSup*, *minUtil*, *maxReg*) *// Algorithm 3*
  end for
  Output: Current pattern set P
end while

Algorithm 1 describes the top-level orchestration of the MOS-SPM framework. At each window slide, the algorithm first evicts the expired batch and integrates the incoming batch by invoking the structure update procedure (Algorithm 2). It then iterates over every root node in the global linked list and launches the recursive mining procedure (Algorithm 3) to discover all patterns that simultaneously satisfy the three thresholds: minimum support (*minSup*), minimum utility (*minUtil*), and maximum regularity gap (*maxReg*).

B. STRUCTURE UPDATE MECHANISM AND OBJECT POOL

Algorithm 2 handles the data update process. It is important to note that this algorithm only builds and maintains root nodes (patterns of length 1) in the global linked list. Child nodes representing patterns of length ≥2 are generated lazily during the recursive mining phase (Algorithm 3). This design avoids pre-creating the entire pattern tree, significantly saving memory because only branches that pass the pruning thresholds are instantiated.

During the expired batch cleanup step, the system applies the Object Pool memory management technique. Instead of requesting the operating system to fully destroy empty nodes, the algorithm reclaims them and reuses them for allocating new items. This bounds the allocated memory capacity and avoids disrupting the system through automatic garbage collection.

As an example with sequence $S1 = \langle (a[2], c[1], d[3]); (a[1], d[2]); a[2] \rangle$, when item a is read, the system reuses an empty node from the Object Pool to load the values. The sids array records identifier 1, and the utils array accumulates the utility of a[2] (in the first itemset), a[1] (second itemset), and a[2] (third itemset).

Algorithm 2 details the two-step linked list update procedure invoked at each window slide. In Step 1, the algorithm scans all existing nodes in the global linked list to remove transaction identifiers belonging to the expired batch Bold; nodes that become empty are reclaimed into the Object Pool for later reuse. In Step 2, for each transaction in the incoming batch Bnew , the algorithm creates or retrieves root nodes (length-1 patterns) and

appends the corresponding identifiers and accumulated utility values. Notably, only root nodes are maintained at this stage; child nodes for longer patterns are generated lazily during the mining phase (Algorithm 3).

Algorithm 2 Linked list update (Construct-MOS-SPM-LL)
Input: New batch B_{new} , expired batch Bold
Output: Updated global linked list with root nodes reflecting the current window

```
// Step 1: Reclaim expired batch data (Object Pool)
for each node N in the global linked list do
    Remove identifiers belonging to Bold from N.sids and N.utils
    if N.sids is empty then
        Unlink N and return N to the memory pool
    end if
end for
// Step 2: Load new batch data (build only length-1 root nodes)
for each transaction sequence S ∈B_new do
    for each distinct item e appearing in S do
        Retrieve node Ne from the memory pool (if available) or allocate a new one
        Append the identifier of S to Ne.sids
        Compute the accumulated utility of e in S and append to Ne.utils
    end for
end for
```

C. PRUNING PROPERTIES

The pattern search process applies two core pruning properties to narrow the search space. Below are the formal statements and proofs for each property.

Property 1 (Accumulated Utility Upper Bound). Let node N represent pattern s in MOS-SPM-LL. Denote $UB(N) = \sum_{i=1}^{|N.sids|} N.utils[i]$ as the total accumulated utility value over the entire utils array. For every pattern s′that is an extension of s (via I-extension or S-extension), we have $u(s') \le UB(N)$.

Proof. Let s′ be an arbitrary extension of s. Since $s \sqsubseteq s'$, the support set of s′ is a subset of the support set of s, meaning every transaction containing s′ also contains s. For each transaction S_q containing s′, the utility $u(s', S_q)$ includes only the utility of the items constituting s′, while the value $N.utils[q]$ includes the utility of s plus the remaining utility in S_q . Therefore, $u(s', S_q) \le N.utils[q]$ for all q. Summing over all transactions containing s′, we obtain:

$$u(s') = \sum_{Sq \supseteq s'} u(s', S_q) \le \sum_{Sq \supseteq s'} N.\text{utils}[q] \le \sum_{i} N.\text{utils}[i] = UB(N) \tag{5}$$

Corollary: If $UB(N) < minUtil$, then no extension of s can be a high utility pattern, and therefore the entire search branch from N can be safely pruned.

Property 2 (Anti-monotonicity of Regularity). Let pattern s have support set $L_s = \langle sid_1, sid_2, \dots, sid_k \rangle$ in window W . For every pattern s′ that is an extension of s, we have $reg(s', W) \ge reg(s, W)$.

Proof. Since $s \subseteq s'$, every transaction containing s′ also contains s, meaning $L_{s'} \subseteq L_s$. Let $L_{s'} = \langle sid'_1, sid'_2, \dots, sid'_p \rangle$ with $p \le k$. Because Ls′ is a subset of Ls, there exists at least one pair of consecutive elements (sid'_j, sid'_{j+1}) in $L_{s'}$ such that the gap $sid'_{j+1} - sid'_j$ is greater than or equal to the maximum gap between any consecutive pair in Ls lying between sid′j and sid′j+1. Furthermore, the gaps to the window boundaries do not decrease: $sid'_1 - sid_{start} \ge sid_1 - sid_{start}$ and $sid_{end} - sid'_p \ge sid_{end} - sid_k$ (since $sid'_1 \ge sid_1$ and $sid'_p \le sid_k$). Because reg takes the maximum of all gaps, we conclude $s \sqsubseteq s' \Rightarrow reg(s', W) \ge reg(s, W)$.

Corollary: If $reg(s, W) > maxReg$, then $reg(s', W) > maxReg$ for every extension s′of s. Therefore, all child branches from N can be safely pruned when the parent pattern has violated the gap threshold.

D. RECURSIVE MINING MECHANISM AND SKIP-PATHS

The pattern search process (Algorithm 3) applies the two pruning properties proven above. During the mining phase, when evaluating an extended pattern $s' = s \oplus e$ (where $\oplus$denotes I-extension or S-extension), the system performs an intersection between the sids array of parent node N and the sids array of the root node representing item e. Only transaction identifiers present in both arrays are retained for the child node, along with utility values recomputed based on the actual occurrence positions in each shared transaction. This intersection mechanism

ensures that the support set of the child pattern is always a proper subset of the parent, maintaining the correctness of all three evaluation metrics. It is important to note that the sequential ordering constraint is not enforced by the intersection operation itself, but is guaranteed by the parent pointer type (pi for I-extension; pe for S-extension) established during structure construction in Algorithm 2. Consequently, every pattern lazily generated during mining is structurally valid with respect to the sequential ordering of its constituent items.

In particular, the system integrates a skip-path mechanism through two parent pointers: pi (for I-extension) and pe (for S-extension). When evaluating a multi-step extended pattern, the algorithm recursively backtracks through the chain of parent pointers to retrieve the utility information of each prefix item. For example, given sequence $S_6 = \langle (a[2], c[1], e[2]); (c[2], d[1]) \rangle$ (Table 2), if the system needs to evaluate pattern $\langle a; d \rangle$, it recursively backtracks through the parent pointers to link item a with d, allowing items c and e lying in between to be skipped. As a result, the algorithm preserves interrupted but high-value combinations, overcoming the pattern loss drawback of the predecessor RStreamHUSP algorithm.

Algorithm 3 presents the recursive multi-objective mining procedure. For each candidate node N , the algorithm first computes three metrics: support (sup), accumulated utility upper bound (UB), and regularity (reg). If all three thresholds are met, the corresponding pattern is added to the result set P. The algorithm then applies Property 1 to prune branches whose utility upper bound falls below *minUtil*. For each feasible extension e, a child node is lazily generated via sids array intersection, and its parent pointer is established to enable skip-path backtracking. Finally, Property 2 is checked to prune children that violate the gap threshold before recursion continues.

Algorithm 3 Recursive multi-objective mining (MOS-Mining)
Input: Current node N , result set P , *minSup*, *minUtil*, *maxReg*
Output: Updated result set P containing all multi-objective patterns discovered from node N

```
sup(N) ←|N.sids|
u(N) ← Σ N.utils[i]
reg(N) ←compute maximum gap from array N.sids
if sup(N) ≥minSup and u(N) ≥minUtil and reg(N) ≤maxReg then
    Add the sequence representation of N to set P
end if
// Prune by Property 1: Accumulated utility upper bound
if UB(N) < minUtil then
    return // No extension from N can reach the utility threshold
end if
for each item e that can extend N (I-Extension or S-Extension) do
    Lazy child node generation: perform sids array intersection
    Nchild.sids ←N.sids ∩sids(e) // Retain only sids containing both N and e
    Recompute Nchild.utils based on occurrence positions in each shared transaction
    Set parent pointer: Nchild.pi ←N or Nchild.pe ←N
    // Traverse skip-paths: backtrack via pi/pe to link non-adjacent items
    // Prune by Property 2: Inherited gap check
    if reg(Nchild) ≤maxReg then
        Call MOS-Mining(Nchild, P , minSup, minUtil, maxReg)
    end if
end for
```

E. COMPLEXITY ANALYSIS

Let |W |denote the total number of transactions in the current window, |I|the number of distinct items, L the maximum length of a transaction sequence, and K the number of candidate patterns generated during traversal.

Time complexity. The structure update phase (Algorithm 2) incurs a cost of O(|B|·L) per data batch B, where |B|is the number of transactions in the batch. The mining phase (Algorithm 3) traverses K candidate nodes. At each node, the sids array intersection requires O(|W|) since both arrays are sorted in ascending order. Overall, the mining cost per window slide is O(K·|W|). In the worst case, K can reach $O(|I|^L)$; however, thanks to the two pruning properties, the actual value of K is considerably smaller. To illustrate this empirically, the theoretical worst-case K for SIGN ($|I| = 267$, avg. $L \approx 52$) would be approximately $267 \times 52 = 13{,}884$ candidate nodes, while the number of patterns actually output at $L = 3$ is 3,308 (Table 5). For LEVIATHAN ($|I| = 9{,}025$, avg. $L \approx 34$), the worst-case $K \approx 307{,}000$, against which the actual output of 8,478 patterns at $L = 3$ represents less than

3 of the theoretical bound. This confirms that the combined effect of Property 1 (utility upper bound pruning) and Property 2 (regularity anti-monotone pruning) dramatically reduces the effective search space in practice.

Space complexity. The MOS-SPM-LL structure stores at most |I|root nodes, each holding two arrays of maximum size |W|. The space cost for root nodes is O(|I|·|W|). Child nodes are lazily generated and released immediately after their branch traversal completes. Therefore, at any given time, at most L child nodes reside on the recursion stack, consuming O(L ·|W|). The Object Pool technique further limits actual allocation costs by reusing reclaimed nodes, keeping the peak memory footprint stable across window sliding cycles.

Comparison. The original RStreamHUSP algorithm [10] maintains a DSRHUS-Trie structure that simultaneously stores all candidates from all batches, resulting in a space cost of O(K·w) (where w is the number of batches in the window) that is not released between slides. Moreover, at each window slide, RStreamHUSP must rescan the entire data of all current batches to rebuild its candidate structure from scratch, whereas MOS-SPM avoids this by lazily re-deriving patterns of length ≥2 through lightweight intersections on pre-maintained root-node sids and utils arrays (without re-reading any raw transaction data). This on-the-fly re-derivation is substantially cheaper than full database rescanning because the root-node arrays are maintained incrementally and remain sorted, allowing O(|W |) intersection per node.

V. EXPERIMENTAL RESULTS

This section presents the performance analysis and evaluation of the MOS-SPM algorithm through a series of experiments on diverse datasets. As comparison baselines, the study establishes two benchmarks based on the core principles of two state-of-the-art algorithms, CHUSP [9] and RStreamHUSP [10]. It should be noted that these are versions that have been proactively refined and supplemented with additional constraints (denoted Extended CHUSP and Extended RStreamHUSP) compared to the original works, to address the multi-objective mining problem over streams and ensure fairness during comparison.

A. EXPERIMENTAL SETUP

The experiments use four standard publicly available datasets from the SPMF library [20], including SIGN, BMS1 spmf, BIBLE, and LEVIATHAN. Detailed specifications of the datasets are presented in Table 3. The datasets were selected to ensure diversity in characteristics: SIGN and BIBLE are dense datasets with long sequence lengths, while BMS1 spmf is a sparse dataset with short sequences; sizes range from 730 to 59,601 sequences. External utility values were randomly generated following a uniform distribution in the range [1, 1000], and internal utility values were generated in the range [1, 10]. These ranges were selected to reflect realistic commercial transaction scenarios: external utility (unit prices) naturally spans a wide range across product categories, while internal utility (purchase quantities) typically consists of small integers in retail transactional data. The implementation environment uses the Java programming language on IntelliJ IDEA 2025.3.4, MacOS 26.3 operating system, Apple M1 processor, and 8GB RAM. To ensure objectivity, each configuration was executed independently three times. The result of the first run was discarded to eliminate noise from JVM code optimization, and only the average of the two subsequent runs was used.

To ensure objectivity and comprehensive evaluation of the multi-objective problem, two state-of- the-art (SOTA) algorithms were extended as comparison baselines. The first is Extended CHUSP, inherited from the core static algorithm CHUSP [9] with an additional regularity checking module in the post-processing stage for verifying correctness in the scenario where the entire dataset forms a single window. The second is Extended RStreamHUSP, built upon the stream processing algorithm RStreamHUSP [10] with an integrated support constraint to form a comprehensive multi-objective benchmark for evaluating performance on streaming data.

Table 3. *Specifications of experimental datasets*

Dataset	Sequences (|SDB|)	Distinct items (|I|)	Avg. length	Characteristics
SIGN	730	267	51.99	Dense, SIGN language
BMS1_SPMF	59,601	497	2.51	Sparse, Click-stream
BIBLE	36,369	13,905	21.64	Moderately dense, Text
LEVIATHAN	5,834	9,025	33.81	Dense, Text

B. CORRECTNESS EVALUATION

This experiment was designed to verify the data preservation capability of the MOS-SPM-LL structure when compared against the static Extended CHUSP algorithm. The stream environment was configured as a single window to disable gap-based pruning constraints, with fixed parameters presented in Table 4.

Table 4. *Fixed configuration parameters for each dataset*

Dataset	*batchSize*	*maxReg*	*minSup*	*minUtil*
SIGN	730	730	150	10,000.0
BMS1_SPMF	59,601	59,601	200	8,000.0
BIBLE	36,369	36,369	400	20,000.0
LEVIATHAN	5,834	5,834	100	9,000.0

Results recorded in Table 5 show that the number of patterns extracted by MOS-SPM matches 100% with the baseline method across all examined lengths (L = 1, L = 2, L = 3) on all four datasets. This match provides empirical evidence that the proposed algorithm operates reliably and does not lose information during the graph traversal process.

Table 5. *Comparison of pattern counts with Extended CHUSP*

Dataset	ExCHUSP L=1	ExCHUSP L=2	ExCHUSP L=3	MOS-SPM L=1	MOS-SPM L=2	MOS-SPM L=3
SIGN	60	989	3,308	60	989	3,308
BMS1_SPMF	166	301	324	166	301	324
BIBLE	174	1,599	4,932	174	1599	4,932
LEVIATHAN	136	1,839	8,478	136	1839	8,478

Although the single-window configuration above confirms the structural correctness of the MOS-SPM-LL data structure and the completeness of the backtrackTravel traversal, it does not exercise the dynamic operations that constitute the core of the streaming pipeline, namely the incremental eviction of expired transactions via slidingWindowUpdate() and the re-evaluation of the regularity constraint as the window boundaries (sid_{start}, sid_{end}) advance at each slide. To address this gap, a complementary per-window validation is conducted on the BMS1 SPMF dataset using Extended CHUSP [9] as an independent ground truth.

For each window position $W_i = [sid_{start}^{(i)}, sid_{end}^{(i)}]$, the transactions belonging to W_i are extracted and re-indexed into a local static database with sequential identifiers $1,2,\dots,|W|$. Extended CHUSP is then applied to this static database under the same threshold values as MOS-SPM, yielding a reference pattern set $\mathcal{G}_i$. This comparison is valid because the regularity metric (Formula 3) is shift-invariant with respect to the window origin: for any pattern s with support set $\langle sid_1, \dots, sid_k \rangle$ inside W_i, the inter-occurrence gaps $sid_{j+1} - sid_j$ and the boundary terms $(sid_1 - sid_{start}^{(i)})$ and $(sid_{end}^{(i)} - sid_k)$ are numerically identical in both absolute stream coordinates and re-indexed local coordinates. Consequently, $\mathcal{G}_i$ constitutes an exact ground truth for MOS-SPM's output at window position i. The parameters used in this experiment are listed in Table 6.

Table 6. Configuration parameters for the per-window validation experiment

Parameter	Value
Dataset	BMS1_SPMF [20]
Minimum Support (*minSup*)	200
Minimum Utility (*minUtil*)	8,000.0
Maximum Regularity (*maxReg*)	15,000
Batch size (*batchSize*)	10,000
Batches per window (*numBatch*)	3
Window size (*|W|*)	30,000

Figure 4 plots the number of patterns produced by MOS-SPM and by the ground truth across all consecutive window positions. The two series are indistinguishable: a per-pattern audit confirms that the support, utility, and regularity values are identical in every case, yielding an absolute 100% match across all window transitions.

C. AVERAGE EXECUTION TIME COMPARISON

The window sliding time of the algorithms was measured while varying the minimum support threshold (*minSup*). The data in Table 7 and Figure 5 indicate that MOS-SPM requires significantly lower execution time than Extended RStreamHUSP across all configurations. Instead of rescanning raw transaction data and rebuilding candidate structures from scratch at each window slide, the proposed method lazily re-derives patterns of length ≥2 by intersecting pre-maintained sids arrays at root nodes, which is a substantially cheaper operation that eliminates redundant I/O and re-initialization costs. This mechanism reduces redundant computational costs, and is especially effective on dense datasets such as BIBLE and LEVIATHAN.

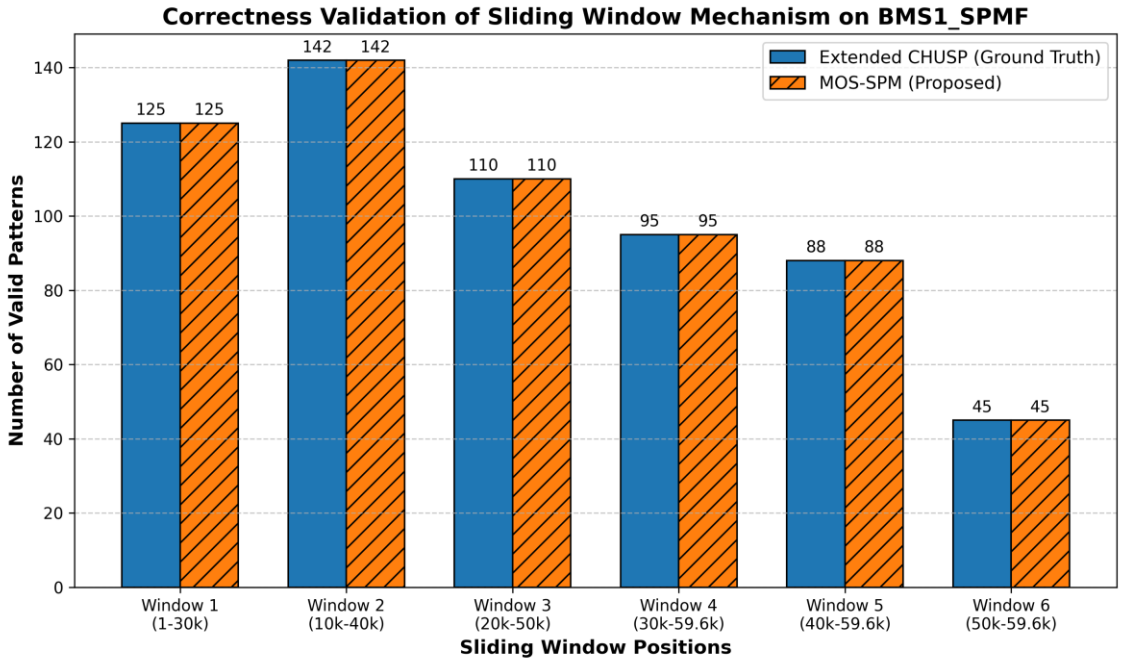

Figure 4. Pattern count comparison between MOS-SPM and the ground truth (Extended CHUSP applied independently to each window slice) on the BMS1 SPMF dataset.

Table 7. *Average execution time comparison over streams (s)*

Dataset	*minSup*	ExRStreamHUSP (s)	MOS-SPM (s)
SIGN	100	18.25	1.85
SIGN	150	18.94	1.43
SIGN	200	18.11	1.02
SIGN	250	18.14	0.69
SIGN	300	18.11	0.41
BMS1_SPMF	150	34.07	0.39
BMS1_SPMF	200	34.16	0.28
BMS1_SPMF	250	34.07	0.22
BMS1_SPMF	300	33.76	0.18
BMS1_SPMF	350	33.62	0.16
BIBLE	300	520.69	58.14
BIBLE	400	521.70	54.68
BIBLE	500	525.83	51.10
BIBLE	600	527.94	50.09
BIBLE	700	525.50	46.14
LEVIATHAN	50	124.21	16.10
LEVIATHAN	100	123.33	14.61
LEVIATHAN	150	123.01	13.15
LEVIATHAN	200	122.92	12.79
LEVIATHAN	250	122.81	12.38

A notable phenomenon is that the execution time of Extended RStreamHUSP remains nearly unchanged when the *minSup* threshold increases. The reason is that the dominant computational cost of this algorithm lies in the phase of building and maintaining the DSRHUS-Trie structure, which involves scanning the entire data of all current batches to generate candidates and initialize counting arrays. This process occurs before the *minSup* filter is applied, and therefore changing the support threshold only affects the number of output patterns without significantly reducing the structure construction time. In contrast, MOS-SPM applies pruning directly during the tree traversal phase, so its execution time decreases noticeably as the threshold increases.

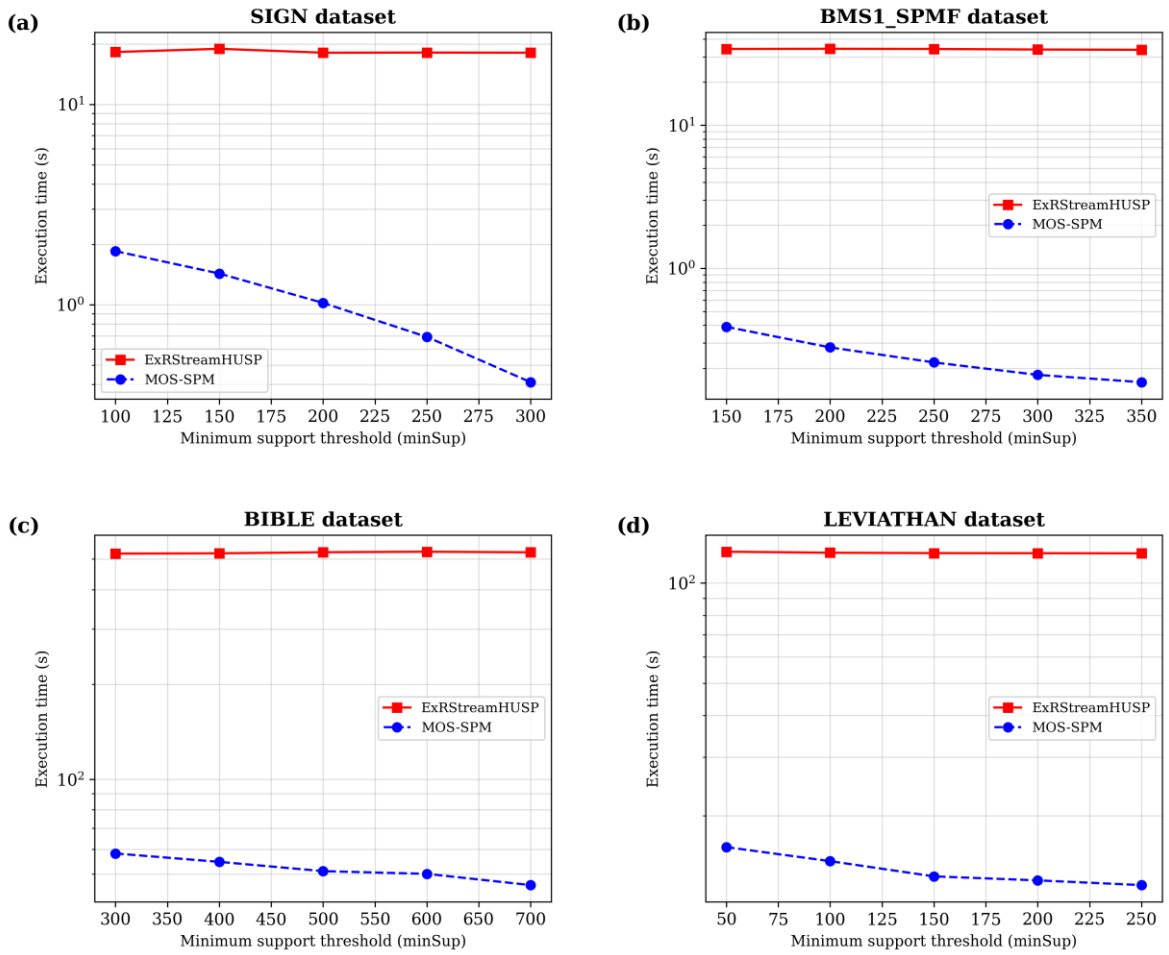

Figure 5. *Execution time comparison over streams (Log scale) when varying minSup on datasets: (a) SIGN, (b) BMS1_SPMF, (c) BIBLE, and (d) LEVIATHAN.*

D. PEAK MEMORY EVALUATION

Memory management is a critical factor when analyzing continuous data streams. The results in Table 8 and Figure 6 show that the peak RAM usage of MOS-SPM is only a fraction of that of Extended RStreamHUSP. While the baseline algorithm consumes up to several gigabytes of memory to manage its massive hash tables, the MOS-SPM algorithm limits storage space to a few hundred megabytes. This low consumption is achieved through the compact design of the MOS-SPM-LL structure combined with the dynamic allocation technique (Object Pool), which maximizes node reuse and minimizes garbage generation.

E. IMPACT OF STREAM CHARACTERISTICS

The sensitivity analysis was conducted by adjusting the batch size (batchSize) and the number of batches in the window (windowSize). The fixed thresholds used throughout this experiment are: minSup = 150, minUtil = 10,000, maxReg = 200 for SIGN; minSup = 200, minUtil = 8,000, maxReg = 5,000 for BMS1_SPMF; minSup = 400, minUtil = 20,000, maxReg = 2,000 for BIBLE; and minSup = 100, minUtil = 9,000, maxReg = 300 for Leviathan. According to Table 9 and Figure 7, as the number of transactions per batch increases, the execution time grows at a linear rate. A notable exception is observed for the SIGN dataset at batchSize = 250, where the execution time (1.20s) decreases compared to batchSize = 200 (1.54s). This is not a per-slide performance improvement but a structural boundary effect caused by a reduction in the total number of mining passes. The SIGN dataset contains 730 sequences. With a window of 3 batches, batchSize = 200 produces three full batches (sids 1 to 600) plus one remainder batch (sids 601 to 730), triggering one full-window mining pass and one final remainder pass (2 passes total). By contrast, batchSize = 250 yields only two full batches (sids 1 to 500), which never fills the 3-batch window during the main processing loop, followed by one final remainder pass (1 pass total). The reduction from 2 to 1 mining pass is the dominant factor explaining the observed time decrease, despite the larger per-pass window size at batchSize = 250. Conversely, expanding the number of batches tightens the regularity constraint, leading to a decline in the number of valid patterns. Despite fluctuations in the search space, the system still responds stably and exhibits no sudden computational bottlenecks.

Table 8. *Peak memory comparison over streams (MB)*

Dataset	Window (batches)	ExRStreamHUSP (MB)	MOS-SPM (MB)
SIGN	2	1183.27	248.20
SIGN	3	1707.33	52.79
SIGN	4	1707.51	62.67
SIGN	5	1707.51	65.41
SIGN	6	1709.51	68.85
BMS1_SPMF	2	2393.53	68.30
BMS1_SPMF	3	2393.53	72.50
BMS1_SPMF	4	2391.53	74.97
BMS1_SPMF	5	2391.53	77.34
BMS1_SPMF	6	2391.53	80.17
BIBLE	2	3410.42	419.96
BIBLE	3	3410.42	487.58
BIBLE	4	3412.41	552.24
BIBLE	5	3412.30	608.69
BIBLE	6	3410.28	661.66
LEVIATHAN	2	3513.87	254.58
LEVIATHAN	3	3513.87	315.21
LEVIATHAN	4	3513.87	356.96
LEVIATHAN	5	3513.87	367.26
LEVIATHAN	6	3513.87	343.00

Table 9. *Impact of stream characteristics on MOS-SPM performance*

Dataset	Batch size	Time (s)	Num batches	Time (s)	Patterns
SIGN	50	0.02	2	0.30	71
SIGN	100	1.00	3	0.97	536
SIGN	150	1.52	4	1.88	587
SIGN	200	1.54	5	2.29	385
SIGN	250	1.20	6	2.40	209
BMS1_SPMF	1,000	0.07	2	0.11	0
BMS1_SPMF	2,000	0.14	3	0.22	0
BMS1_SPMF	3,000	0.22	4	0.37	0
BMS1_SPMF	4,000	0.30	5	0.52	0
BMS1_SPMF	5,000	0.38	6	0.73	0
BIBLE	500	14.69	2	27.96	29
BIBLE	1,000	30.56	3	56.64	23
BIBLE	1,500	44.61	4	92.64	17
BIBLE	2,000	57.33	5	131.15	15
BIBLE	2,500	68.25	6	173.10	13
LEVIATHAN	500	4.74	2	9.06	9
LEVIATHAN	1,000	9.08	3	14.06	6
LEVIATHAN	1,500	13.62	4	16.82	6
LEVIATHAN	2,000	18.36	5	19.42	6
LEVIATHAN	2,500	22.89	6	21.92	6

Notably, on the BMS1_SPMF dataset, the number of multi-objective patterns found is zero across all batch-size and window-size configurations in Table 9, including the largest tested configuration ($batchSize = 5{,}000$, window of 6 batches covering 30,000 sequences). This consistent zero result is not caused by threshold values being disproportionate relative to the window size; rather, it reflects the intrinsic structural sparsity of BMS1_SPMF (average sequence length = 2.51 items per sequence). Such short sequences rarely share common subsequences that simultaneously satisfy all three constraints. For reference, the single-window correctness experiment (Tables 4-5) confirms that the algorithm does produce valid patterns on BMS1_SPMF (166 patterns at $L = 1$, 301 at $L = 2$, 324 at $L = 3$) under parameters matched to the full dataset size, demonstrating that the zero result in the sensitivity analysis is a characteristic of the dataset, not an algorithmic artifact. This result confirms that the algorithm accurately reflects the inherent sparsity of the data rather than producing false positives.

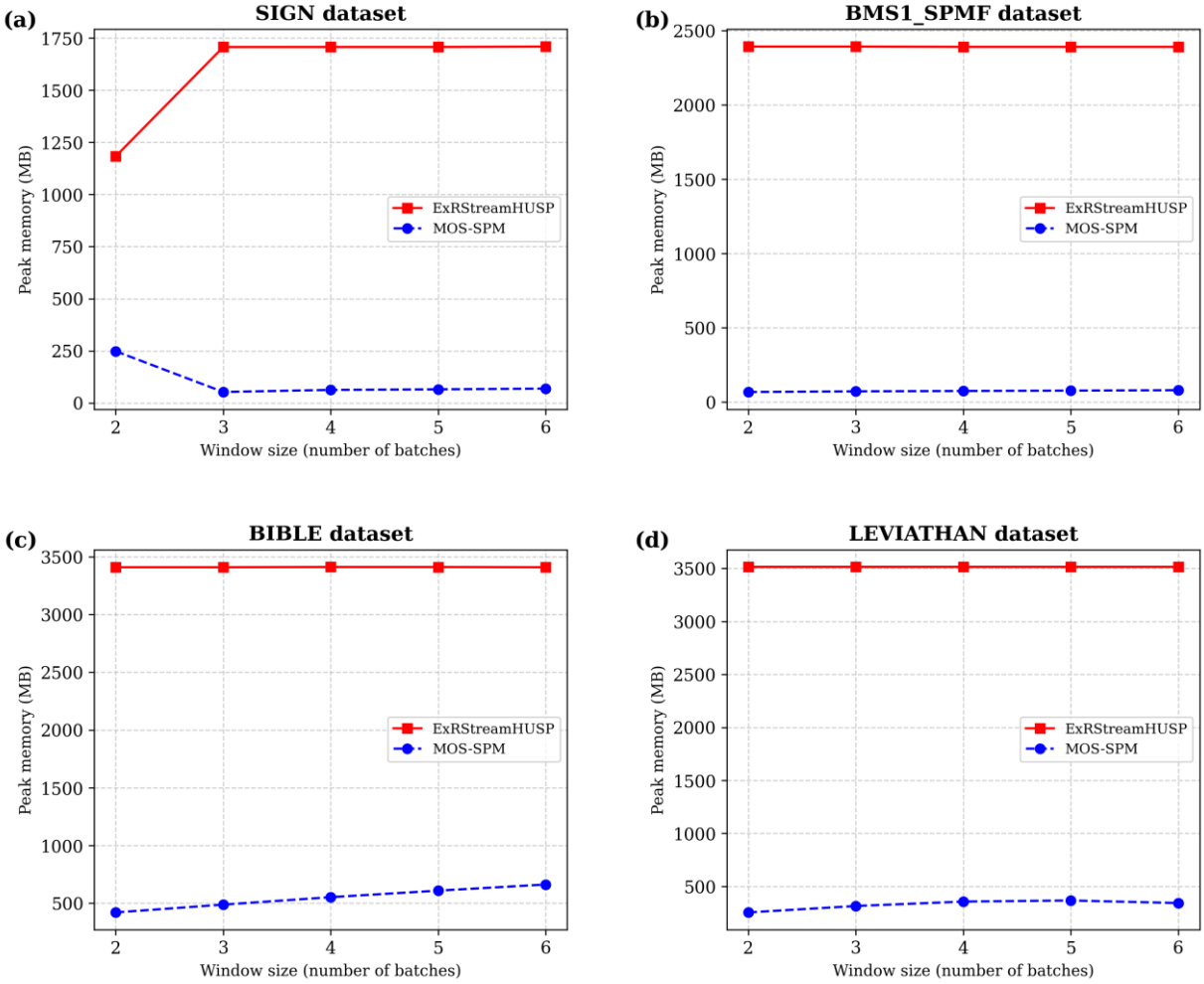

Figure 6. *Peak memory comparison when varying window size on datasets: (a) SIGN, (b) BMS1_SPMF, (c) BIBLE, and (d) LEVIATHAN.*

F. PATTERN PRESERVATION EVALUATION

The final experiment compares the number of patterns found under varying gap thresholds (*maxReg*), ranging from strict to relaxed. The results in Table 10 and Figure 8 show that Extended RStreamHUSP loses a large number of valid patterns, and in particular returns no patterns on the BMS1_SPMF dataset across all three configuration levels. This limitation stems from the strict sequential candidate generation mechanism of the original algorithm. Addressing this drawback, through the skip-path mechanism, MOS-SPM is capable of linking non-adjacent items and preserving the utility values of interrupted patterns, thereby maintaining complete integrity of the output results.

Table 10. *Number of multi-objective regular patterns found when varying gap threshold (maxReg)*

Dataset (*maxReg 1,2,3*)	**ExRStream L1**	**MOS-SPM L1**	**ExRStream L2**	**MOS-SPM L2**	**ExRStream L3**	**MOS-SPM L3**
SIGN *(10, 15, 20)*	272	536	340	835	360	952
BMS1_SPMF *(100, 200, 300)*	0	568	0	829	0	967
BIBLE *(5, 10, 15)*	0	342	40	628	123	739
LEVIATHAN *(10, 15, 20)*	35	120	42	246	61	384

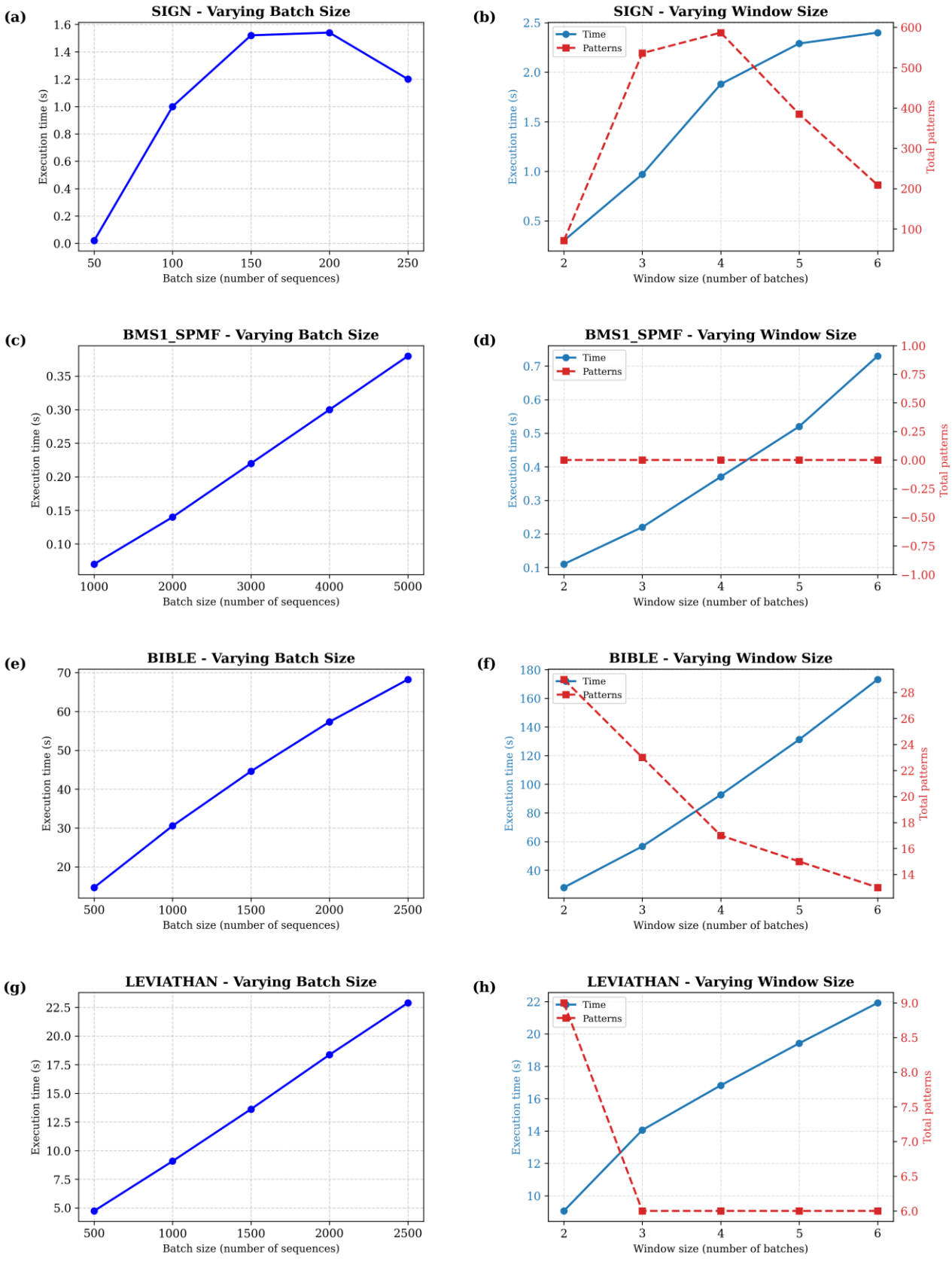

Figure 7. *Summary of sensitivity analysis results. Left column (a, c, e, g) shows execution time when varying batch size. Right column (b, d, f, h) shows execution time and pattern count when varying window size.*

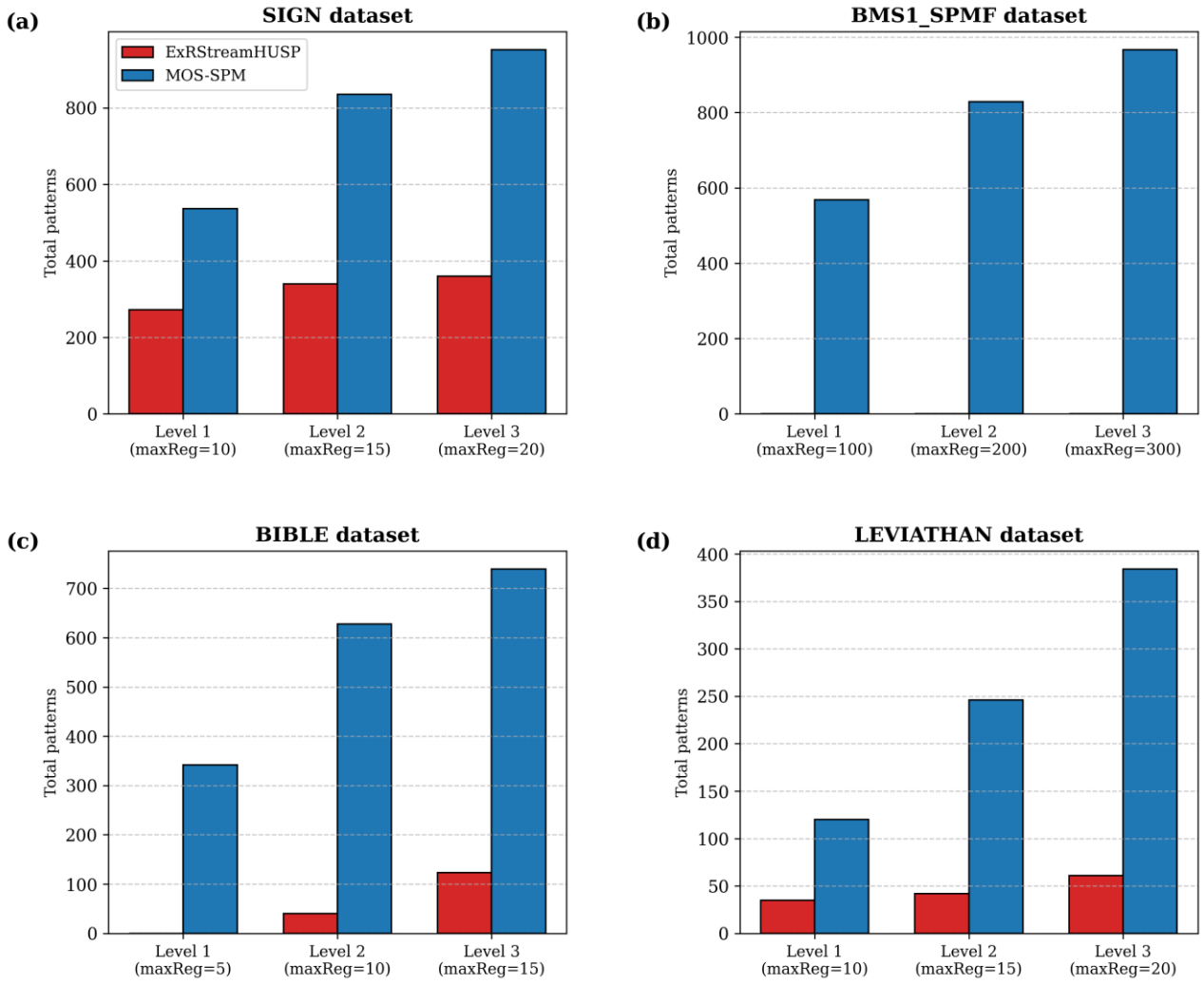

Figure 8. Comparison of pattern counts between ExRStreamHUSP and MOS-SPM when varying the maxReg threshold on datasets: (a) SIGN, (b) BMS1 SPMF, (c) BIBLE, and (d) LEVIATHAN.

VI. CONCLUSION

This paper has proposed the MOS-SPM method and the specialized MOS-SPM-LL storage structure to address the multi-objective sequential pattern mining problem over data streams. This solution enables the system to flexibly search for and maintain pattern sets that simultaneously satisfy three core criteria, including support, utility, and regularity. Two pruning properties (accumulated utility upper bound and anti-monotonicity of regularity) have been formally proven, ensuring theoretical correctness for the search space reduction process.

The experimental evaluation results show that the proposed method guarantees the integrity of the output pattern set when compared against the static baseline Extended CHUSP. Furthermore, owing to the compact linked list design and node reuse technique, the algorithm significantly improves update time and saves storage space compared to the stream processing algorithm Extended RStreamHUSP. In future work, two primary directions are identified. First, the framework will be extended to incorporate an Average Utility measure, which normalizes utility by pattern length, to suppress redundant long patterns and enable fairer comparisons across patterns of varying lengths. This extension is compatible with the current MOS-SPM-LL structure, as the utility sum per node is already maintained and pattern length is derivable from the parent-pointer chain depth. Second, extending the framework to handle datasets with negative unit profits represents an important practical direction. The current Accumulated Utility Upper Bound (Property 1) relies on the non-negativity of utility values; with negative profits, this guarantee no longer holds, requiring a redesigned upper bound - for example, adapting two-phase utility bounds from itemset mining to the sequential context. Both directions will be pursued in subsequent work.

VII. ACKNOWLEDGEMENT

This research is funded by Ho Chi Minh City University of Foreign Languages - Information Technology under grant number H2025-01.

VIII. REFERENCES

[1] Ngai, E. W. T., Xiu, L., & Chau, D. C. K. (2009). Application of data mining techniques in customer relationship management. Expert Systems with Applications, 36(2), 2592-2602.
[2] Lent, B., Agrawal, R., & Srikant, R. (1997). Discovering trends in text databases. In Proc. KDD-97, pp. 227-230.
[3] Ganguly, A. R., & Steinhaeuser, K. (2009). Data mining for climate change and impacts. In Proc. ICDMW'09, pp. 385-392.
[4] Srivastava, J., Cooley, R., Deshpande, M., & Tan, P.-N. (2000). Web usage mining: Discovery and applications of usage patterns from web data. SIGKDD Explorations, 1(2), 12-23.
[5] Pei, J., Han, J., et al. (2001). PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In Proc. ICDE 2001, pp. 215-224.
[6] Zaki, M. J. (2001). SPADE: An efficient algorithm for mining frequent sequences. Machine Learning, 42(1-2), 31-60.
[7] Shie, B.-E., Hsiao, H.-F., Tseng, V. S., & Yu, P. S. (2011). Mining high utility mobile sequential patterns in mobile commerce environments. In Proc. DASFAA 2011, LNCS 6587, pp. 224-238.
[8] Yin, J., Zheng, Z., & Cao, L. (2012). USpan: An efficient algorithm for mining high utility sequential patterns. In Proc. KDD'12, pp. 660-668.
[9] Dinh, D.-T., Fournier-Viger, P., & Huynh, V. N. (2023). Mining compact high utility sequential patterns. Applied Intelligence, 53(5), 5548-5568.
[10] Ishita, S. Z., Ahmed, C. F., & Leung, C. K. (2022). New approaches for mining regular high utility sequential patterns. Applied Intelligence, 52(4), 3781-3806.
[11] Agrawal, R., & Srikant, R. (1995). Mining sequential patterns. In Proc. ICDE'95, pp. 3-14.
[12] Fournier-Viger, P., Gomariz, A., Campos, M., & Thomas, R. (2014). Fast vertical mining of sequential patterns using co-occurrence information. In Proc. PAKDD 2014, LNCS 8443, pp. 40-52.
[13] Fournier-Viger, P., Lin, J. C.-W., Kiran, R. U., Koh, Y. S., & Thomas, R. (2017). A survey of sequential pattern mining. Data Science and Pattern Recognition, 1(1), 54-77.
[14] Li, X., Chen, L., Zhang, C., Yang, J., & Gan, W. (2023). Mining actionable combined high utility incremental and associated sequential patterns. PLOS ONE, 18(1), e0280430.
[15] Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., & Choi, H.-J. (2012). Interactive mining of high utility patterns over data streams. Expert Systems with Applications, 39(15), 11979-11991.
[16] Tran, M.-T., Tran, A.-D., Pham, D.-T., & Le, M.-N. (2024). Method for mining high-utility patterns in transaction stream data based on linked list structure. ICT Research, 2024(2).
[17] Dawar, S., Bedi, P., & Goyal, V. (2022). StreamHUSP: A fast and efficient algorithm for high utility sequential pattern mining over data streams. Information Sciences, 606, 68-90.
[18] Tanbeer, S. K., Ahmed, C. F., Jeong, B.-S., & Lee, Y.-K. (2008). Mining regular patterns in transactional databases. IEICE Trans. Inf. & Syst., E91-D(11), 2568-2577.
[19] Ishita, S. Z., Ahmed, C. F., & Leung, C. K. (2022). A generalized framework for regular high utility sequential pattern mining. Expert Systems with Applications, 201, 117132.
[20] Fournier-Viger, P., Lin, J. C.-W., et al. (2016). The SPMF open-source data mining library version 2. In Proc. ECML PKDD 2016, LNCS 9853, pp. 36-40.

KHAI THÁC MẪU TUẦN TỰ ĐA MỤC TIÊU TRÊN LUỒNG DỮ LIỆU GIAO DỊCH DỰA TRÊN MÔ HÌNH CỬA SỔ TRƯỢT

Trần Minh Thái, Trần Anh Duy, Lê Thị Minh Nguyện, Phạm Đức Thành

TÓM TẮT— Khai thác mẫu tuần tự đóng vai trò thiết yếu trong nhiều lĩnh vực thực tiễn như phân tích hành vi khách hàng, chẩn đoán y tế và giám sát an ninh mạng. Mặc dù các nghiên cứu trước đây đã giải quyết tốt bài toán tìm kiếm mẫu phổ biến hoặc mẫu mang lại lợi ích cao trên cơ sở dữ liệu tĩnh, sự bùng nổ của các luồng dữ liệu cập nhật liên tục đã làm nảy sinh những thách thức nghiêm trọng về độ trễ thời gian và giới hạn không gian lưu trữ. Để vượt qua các rào cản này, bài báo đề xuất thuật toán khai thác mẫu tuần tự đa mục tiêu trên luồng (Multiple Objective Stream Sequential Pattern Mining, gọi tắt là MOS-SPM) vận hành trên cấu trúc danh sách liên kết chuyên biệt (MOS-SPM-LL) dựa trên mô hình cửa sổ trượt. Phương pháp tiếp cận này cho phép hệ thống đánh giá đồng thời ba tiêu chí cốt lõi bao gồm độ phổ biến, mức độ hữu ích cao và tính thường xuyên của mẫu trực tiếp trên luồng giao dịch. Để đánh giá toàn diện, nghiên cứu đã chủ động mở rộng hai thuật toán tiên tiến hiện hành là CHUSP và RStreamHUSP làm cơ sở đối chiếu. Kết quả thực nghiệm chứng minh rằng thuật toán đề xuất bảo toàn tính

chính xác tuyệt đối khi so sánh với phiên bản CHUSP mở rộng. Đồng thời, nhờ cơ chế lưu trữ gọn nhẹ, MOS-SPM thể hiện hiệu năng vượt trội về cả tốc độ thực thi lẫn khả năng tối ưu hóa bộ nhớ so với phiênbản RStreamHUSP mở rộng.

Từ khoá—Khai thác mẫu tuần tự, luồng dữ liệu, mẫu hữu ích cao, mẫu thường xuyên, cửa sổ trượt.

Trần Minh Thái là tiến sỹ Công nghệ thông tin (CNTT). Hiện tại, TS. Thái là giảng viên và trưởng bộ môn Hệ thống Thông tin thuộc khoa CNTT, Trường Đại học Ngoại ngữ -Tin học TP.HCM. Lĩnh vực nghiên cứu chính của TS. Thái liên quan đến vấn đề khai thác dữ liệu, ẩn dữ liệu, xử lý dữ liệu lớn và nhận dạng.

Lê Thị Minh Nguyện nhận học vị thạc sĩ Khoa học máy tính trường Đại học Quốc gia Thành phố Hồ Chi Minh năm 2007. Hiện là giảng viên khoa Công Nghệ Thông Tin trường Đại Học Ngoại Ngữ Tin Học thành phố Hồ Chí Minh. Lĩnh vực nghiên cứu đang quan tâm là: khai thác dữ liệu.

Trần Anh Duy Nhận học vị thạc sĩ Khoa học máy tính trường Đại học Khoa Học Tự Nhiên năm 2017. Hiện là giảng viên khoa Công Nghệ Thông Tin trường Đại Học Ngoại Ngữ Tin Học thành phố Hồ Chí Minh. Lĩnh vực nghiên cứu đang quan tâm là: Khai thác dữ liệu. ...

Phạm Đức Thành nhận học vị Thạc sĩ năm 2006 tại Đại học Quốc gia Thành phố Hồ Chí Minh; hiện đang là Giảng viên công tác tại khoa Công nghệ Thông tin trường Đại học Ngoại ngữ Tin học Tp. Hồ Chí Minh; lĩnh vực nghiên cứu đang quan tâm là: khai thác dữ liệu.