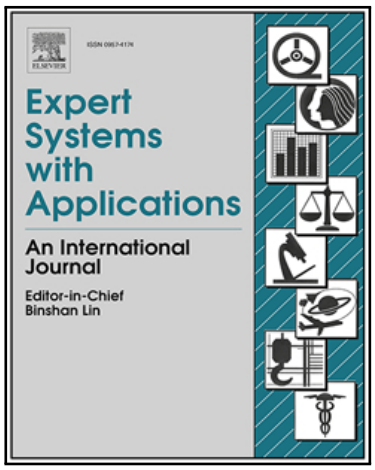


An Efficient Algorithm for High Average-Efficiency Itemset Mining over Data Streams

Gufeng Li, Shuo Chen, Xuanwei Zhang, Tao Shang

PII: S0957-4174(25)04005-9
DOI: https://doi.org/10.1016/j.eswa.2025.130390
Reference: ESWA 130390

To appear in: *Expert Systems With Applications*

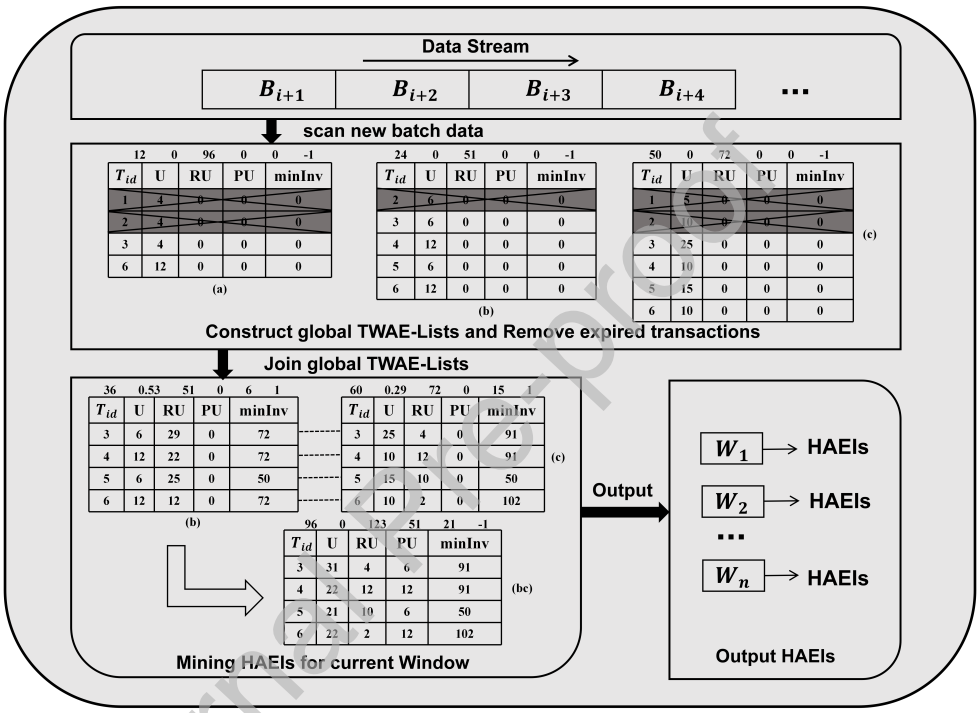
Received date: 9 July 2025
Revised date: 16 October 2025
Accepted date: 9 November 2025

Please cite this article as: Gufeng Li, Shuo Chen, Xuanwei Zhang, Tao Shang, An Efficient Algorithm for High Average-Efficiency Itemset Mining over Data Streams, *Expert Systems With Applications* (2025), doi: https://doi.org/10.1016/j.eswa.2025.130390

Graphical Abstract

An Efficient Algorithm for High Average-Efficiency Itemset Mining over Data Streams

Gufeng Li, Shuo Chen, Xuanwei Zhang, Tao Shang



Highlights

An Efficient Algorithm for High Average-Efficiency Itemset Mining over Data Streams

Gufeng Li, Shuo Chen, Xuanwei Zhang, Tao Shang

- The first high average-efficiency itemset mining algorithm applied to data streams
- A TWAE-List structure is proposed to improve window update and mining efficiency
- Two tighter average-efficiency upper bounds are introduced to reduce candidates
- The ETP strategy is designed to accelerate sliding window update processes
- Experiments show significant improvements in runtime efficiency and scalability

An Efficient Algorithm for High Average-Efficiency Itemset Mining over Data Streams

Gufeng Li[a,b,*], Shuo Chen[a], Xuanwei Zhang[b], Tao Shang[a,b]

[a]*Hangzhou Institute of Technology, Xidian University, Hangzhou, 311231, Zhejiang, China*
[b]*State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an, 710071, Shaanxi, China*

Abstract

In high utility itemset mining over data streams, the traditional total utility evaluation method tends to favor longer itemsets, while ignoring the additional cost and management complexity they may introduce. Although previous studies have proposed high average-utility and high-efficiency itemset mining methods to incorporate itemset length and cost into utility evaluation, most of these approaches are designed for static databases and are not well-suited to the frequent updates and real-time requirements of data stream environments. To address this problem, this paper proposes a high average-efficiency itemset mining algorithm over data streams, called HAEIM_Stream. The algorithm introduces a transactional-window average-efficiency list, which integrates transaction-level and window-level utility information. This structure improves the efficiency of itemset evaluation and significantly accelerates list-based join operations during mining. At the same time, a tighter upper bound is designed and an effective pruning strategy is proposed to filter out unpromising itemsets in advance and reduce the search space. In addition, an efficient outdated transaction deletion strategy is adopted to remove expired transactions during sliding window updates, ensuring the timeliness of itemset mining. Finally, experimental results in multiple sparse and dense datasets verify the effectiveness and efficiency of the proposed list structure

*Corresponding author
Email addresses: ligufeng@xidian.edu.cn (Gufeng Li), chshuo@stu.xidian.edu.cn (Shuo Chen), 24011211115@stu.xidian.edu.cn (Xuanwei Zhang), tshang@xidian.edu.cn (Tao Shang)

and pruning strategies.

Keywords:
Data mining, High average-efficiency itemsets mining, Data streams, Average-efficiency List, Sliding window

1. Introduction

High utility itemset mining (HUIM) has become an important research direction in the field of data mining (Liu and Qu, 2012; Fournier-Viger et al., 2014b; Krishnamoorthy, 2015, 2017; Duong et al., 2018; huy Duong et al., 2017; Cheng et al., 2022; Yan et al., 2024). Unlike traditional frequent pattern mining (FPM) (Agrawal and Srikant, 1998; Han et al., 2000; Djenouri et al., 2018; Uno et al., 2004), which mainly focuses on the frequency of itemset occurrences, HUIM introduces both internal utility (such as quantity and time) and external utility (such as price and weight) to achieve a quantitative evaluation of the economic value of itemsets. This method shows unique advantages in mining high utility but low-frequency patterns and has been widely applied in business decision-making, IoT monitoring, and resource allocation (Kumar and Singh, 2023). Although HUIM improves the economic sensitivity of pattern mining, its total utility measure is inherently biased toward longer itemsets, which are more likely to achieve higher utility due to cumulative effects (Kumar and Rana, 2021). In practice, the increase in itemset length is often accompanied by additional cost input or management complexity. Therefore, using total utility as the sole evaluation criterion may deviate from actual economic rationality. This issue limits the applicability of HUIM in cost-sensitive scenarios.

To address the unfairness caused by itemset length, researchers proposed the high average-utility itemset mining (HAUIM) method (Hong et al., 2011; Lin et al., 2016; Li et al., 2022). This method calculates the average-utility of an itemset by dividing its total utility by its length, thereby avoiding the inflation effect of item number on the final utility evaluation. HAUIM is able to maintain the ability to identify high utility itemsets while improving the fairness of comparison among itemsets of different sizes and enhancing its practical applicability.

However, although HAUIM achieves utility measurement based on length normalization, it still fails to consider the resource cost required to obtain the utility. In real-world scenarios, the acquisition of profit is often accompanied

by certain upfront investments and operational risks. Evaluating patterns solely based on utility or average-utility metrics cannot fully reflect economic feasibility. To address this issue, Zhang et al. proposed the high-efficiency itemset mining (HEIM) problem, which incorporates both profit and cost into a unified framework (Zhang et al., 2023). The algorithms related to HEIM improve the sensitivity to risk and the overall decision-making value. On this basis, Yildirim further proposed the high average-efficiency itemset mining (HAEIM) method, which introduces cost into the average-utility calculation, enhancing the interpretability of the mining results under practical operational constraints (Yildirim, 2024). HAEIM exhibits stronger adaptability and is applicable to complex decision-making environments that are sensitive to the relationship between efficiency and cost.

Conceptually, HUIM evaluates itemsets by total utility, which favors longer itemsets and ignores costs. HAUIM reduces this length bias by normalizing with respect to itemset size, but it still does not account for costs. HEIM aggregates utility and cost to assess overall return on investment, yet the global ratio can be skewed by a few transactions with very high cost. HAEIM extends HEIM by further normalizing by itemset size, thereby retaining cost awareness and ensuring comparability across different lengths.

With the transition from static data accumulation to dynamic data generation, traditional pattern mining methods based on static databases are facing increasing challenges (Ahmed et al., 2012a). In scenarios such as the Internet of Things and high-frequency communication, data exhibit continuous generation, rapid updates, and unstable patterns, which pose significant challenges for conventional algorithms to respond effectively in real time. Although existing studies have proposed dynamic HUIM algorithms based on sliding window models to adapt to the temporal properties of data streams (Chu et al., 2008; Dawar et al., 2017; Yun et al., 2017; Ahmed et al., 2012b), most of them still focus only on utility itself and fail to systematically integrate itemset length, mining cost, and the influence of dynamic environments on overall benefit. Therefore, it is necessary to construct a comprehensive evaluation mechanism that integrates multiple factors, including utility, length, cost, and market dynamics, to dynamically identify patterns with optimal return and support more robust and rapid analytical processes.

To address the above issues, we propose an efficient algorithm for mining high average-efficiency itemsets over data streams. Specifically, a new average-efficiency list based on the location index structure is introduced to accelerate the join operations of list structures, while an obsolete transaction

deletion strategy is proposed to ensure that expired transactions do not affect subsequent mining of high average-efficiency itemsets. In addition, a tighter upper bound and an effective pruning strategy are proposed to reduce the number of candidates and thus minimize the number of average-efficiency list joins. The main contributions of this work are summarized as follows.

- A new data structure named Transactional-Window Average-Efficiency List (TWAE-List), which integrates both transaction-level and window-level utility information. The structure maintains essential fields such as transaction utility and remaining utility, as well as aggregated utility for fast pruning. This TWAE-List effectively enhances the efficiency of itemset combination and facilitates incremental updates during window sliding.
- To further reduce the candidate search space, a relative maximum average-efficiency upper bound (MAEUB) is introduced to estimate the highest attainable average-efficiency by jointly considering the utility and investment of an itemset and its extensions. Moreover, an estimated average-efficiency upper bound (EAEUB) pruning strategy is proposed to dynamically filter unpromising itemsets during list construction. Both strategies effectively avoid unnecessary joins and reduce computational cost.
- An efficient transaction pruning (ETP) strategy is presented to accelerate window updates over data streams. This strategy significantly reduces the computational cost of maintaining list structures over sliding windows and ensures the timeliness of high average-efficiency itemset mining.
- A high average-efficiency itemset mining algorithm in data streams called HAEIM_Stream is proposed based on the TWAE-list, MAEUB, EAEUB and ETP strategies. Comprehensive experiments conducted on both sparse and dense datasets validate the efficiency of the designed data structure and the pruning methods.

The structure of this paper is as follows. Section 2 reviews related work on high average-efficiency itemset mining. Section 3 introduces the fundamental concepts. In Section 4, two new upper bounds, MAEUB and EAEUB, are proposed. Additionally, a new average-efficiency list structure and the

ETP strategy are presented. Section 5 describes the HAEIM_Stream algorithm, followed by experimental evaluations in Section 6. Finally, Section 7 concludes the paper and discusses future research directions.

2. Related work

2.1. High utility itemset mining

HUIM aims to mine itemsets with high economic or practical value from transactional databases. The Two-Phase algorithm (Liu et al., 2005) adopted a two-phase strategy and first introduces the concept of transaction weighted utility (TWU) for pruning. However, two-phase algorithms such as Two-Phase suffer from low efficiency due to frequent database scans and excessive candidate generation. To eliminate the two-phase process, the HUI-Miner algorithm (Liu and Qu, 2012) first proposed a one-phase utility-list structure. Subsequently, a series of list-based algorithms appeared, such as FHM (Fournier-Viger et al., 2014b), HUP-Miner (Krishnamoorthy, 2015), EFIM (Zida et al., 2015), and HMiner (Krishnamoorthy, 2017). These algorithms enhanced mining performance through advanced utility-list structures and improved pruning strategies. Additionally, the ULB-Miner algorithm (Duong et al., 2018) designed a utility-list buffer structure, optimizing the merging process complexity. The UBP-Miner (huy Duong et al., 2017) algorithm employs a utility bit partition bitmap to further simplify the merging process of utility lists. Meanwhile, the HUIM-SU method (Cheng et al., 2022) introduces a streamlined utility list along with a structural tree, aiming to prune the search space by leveraging extension utility and localized transaction weighted utility. Recent advances further extend HUIM to dynamic and specialized scenarios, including the join-free vertical one-phase FOTH algorithm (Yan et al., 2024), the incremental periodic IPHM algorithm (Huang et al., 2024), fuzzy high-utility mining via IF-HUPM (Chen et al., 2024), and incremental top-k HUIM (Tung et al., 2025).

2.2. High utility itemset mining over data streams

In the data stream scenario, HUIM needs to achieve real-time processing under limited storage and computational resources. The sliding window model, which keeps only the latest segment of data, has emerged as the predominant framework for processing data streams. The earliest THUI-Mine algorithm (Chu et al., 2008) extended the Two-Phase method to the sliding window environment, introducing partitioning and TWU pruning to

generate candidate itemsets, but it still followed a two-phase structure with low efficiency. To improve update performance, MHUI-BIT and MHUI-TID algorithms (Li et al., 2008) proposed new structures to store transactions and realized efficient incremental updates within the sliding window. Subsequently, the HUPMS algorithm (Ahmed et al., 2012c) constructed the HUS-tree structure to avoid itemset merging operations. The SHU-Grow algorithm (Ryang and Yun, 2016) introduced the SHU-tree data structure and incorporated two pruning methods (RGE and RLE) to effectively shrink the search space. In recent years, research has focused more on one-phase structures and incremental update mechanisms for windows. The Vert_top-k_DS algorithm (Dawar et al., 2017) designed the iList-list structure to support fast insertion and deletion of transactions. The SHUPM algorithm (Yun et al., 2017) introduced batch identifiers to improve the accuracy of merging. SOHUPDS algorithm (Jaysawal and Huang, 2020) optimized data structures to reduce redundant computations during window sliding. The HUPM-Stream algorithm (Han et al., 2023) utilizes the Ext-list structure to simplify the joining process of utility lists, and employs IRS pruning along with a hash-based result set maintenance strategy to improve efficiency within sliding window frameworks. The latest SOHUPDS+ algorithm (Jaysawal and Huang, 2024) proposes an improved utility structure IUDataListSW+, and utilizes a BitmapTransactionMerging strategy to accelerate utility computation on dense data streams. It also updates the result set based on high utility patterns from previous windows to improve runtime performance.

2.3. High average-utility itemset mining

Early HAUIM algorithms such as TPAU (Hong et al., 2011) adopted a two-phase strategy and introduced the AUUB upper bound for pruning. Subsequently, the PBAU algorithm (Lan et al., 2012) improved the efficiency of candidate generation using projection techniques. HAUI-Miner (Lin et al., 2016) introduced the AU-list data structure and employed a depth-first search approach, marking the beginning of one-phase algorithms based on utility lists. To further narrow down the search space, later algorithms such as MHAI (Yun and Kim, 2017) and HAUI-Miner+ (Lin et al., 2017b) introduced compact list structures and upper bound estimation methods, along with various pruning strategies, significantly reducing unnecessary extensions. More recent algorithms, including FHAUPM (Lin et al., 2017a), LMAHUP (Kim et al., 2021), and EMAUI (Li et al., 2022), have continued to refine upper bound estimation and enhance overall performance through structural

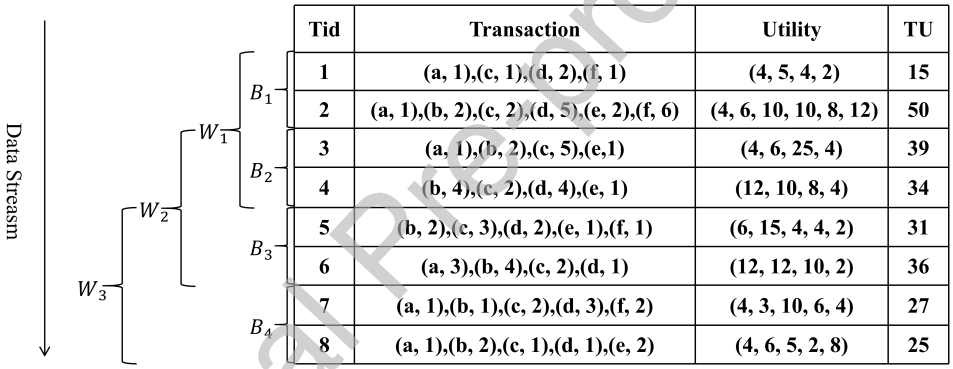
optimization and collaborative pruning strategies. In data stream environments, some studies have extended HAUIM to sliding window and time-decay models. Representative algorithms such as SHAU (Yun et al., 2016) and SHAUPM (Lee et al., 2022) introduced SHAU-tree and SHAUP-list structures to support efficient updating and pruning. The recent IIMHAUP algorithm (Kim et al., 2023) combines an indexed list structure with a dynamic realignment technique to effectively handle incremental data processing, achieving notable computational efficiency and scalability.

2.4. High-efficiency itemset mining

The core idea of HEIM is to identify more efficient patterns from databases by evaluating the ratio between profit and cost, thereby supporting more rational economic decision-making. The first HEPM (Zhang et al., 2023) algorithm introduced an efficiency metric and the EUB upper bound to optimize the two-phase mining process and effectively reduce the search space. To overcome the performance bottleneck caused by candidate generation and multiple database scans, a one-phase HEPMiner (Zhang et al., 2023) algorithm was further proposed. This algorithm introduced four pruning strategies and designed the estimated efficiency co-occurrence structure to accelerate the evaluation of candidate itemsets. The MHEI algorithm (Huynh et al., 2024) proposed tighter subtree efficiency estimations and local efficiency upper bounds to enhance pruning capability, reduce the search space, and significantly decrease runtime and memory usage. The latest EHEPM method (Yildirim, 2025a) introduces four tighter upper bound models and two new data structures, effectively enhancing the pruning capability of the search space and improving the computational efficiency of pattern effectiveness. In addition, to address the issue that HEIM does not consider itemset length, the HAEIMiner algorithm (Yildirim, 2024) incorporated average efficiency into the evaluation process and proposed the AEUB upper bound for pruning. While this approach enhances fairness and applicability, its two-phase framework still involves generating numerous candidate itemsets, which restricts the overall mining efficiency. Recent extensions address special scenarios, including the MHEINU algorithm for mining high-efficiency itemsets with negative utility values (Yildirim, 2025b). A streaming or sliding window formulation of HEIM remains largely unexplored and is a promising direction for future work.

3. Preliminaries and problem statement

Consider a transactional data stream (DS) denoted by $DS = \{T_1, T_2, \ldots, T_j\}$, and let $I = \{i_1, i_2, \ldots, i_m\}$ represent all items contained in the stream. As shown in Figure 1, the stream is partitioned into four batches $\{B_1, B_2, B_3, B_4\}$, and each sliding window spans two consecutive batches. In this example we consider three sliding windows $W_1 = \{B_1, B_2\}$, $W_2 = \{B_2, B_3\}$, and $W_3 = \{B_3, B_4\}$. The initial sliding window is W_1. When the first window reaches capacity, the window slides forward by dropping the oldest batch B_1 and adding the new batch B_3, resulting in W_2. In this sliding window framework, the algorithm processes only a fixed number of the most recent batches inside the current window.



Tid	Transaction	Utility	TU
1	(a, 1),(c, 1),(d, 2),(f, 1)	(4, 5, 4, 2)	15
2	(a, 1),(b, 2),(c, 2),(d, 5),(e, 2),(f, 6)	(4, 6, 10, 10, 8, 12)	50
3	(a, 1),(b, 2),(c, 5),(e,1)	(4, 6, 25, 4)	39
4	(b, 4),(c, 2),(d, 4),(e, 1)	(12, 10, 8, 4)	34
5	(b, 2),(c, 3),(d, 2),(e, 1),(f, 1)	(6, 15, 4, 4, 2)	31
6	(a, 3),(b, 4),(c, 2),(d, 1)	(12, 12, 10, 2)	36
7	(a, 1),(b, 1),(c, 2),(d, 3),(f, 2)	(4, 3, 10, 6, 4)	27
8	(a, 1),(b, 2),(c, 1),(d, 1),(e, 2)	(4, 6, 5, 2, 8)	25

Figure 1: Transactional data streams

Definition 1 (Internal Utility)**.** The internal utility of an item i_l within a transaction T_j is denoted by $qu(i_l, T_j)$.

For example, in Figure 1, $qu(c, T_1) = 1$.

Definition 2 (External utility)**.** For each item $i_l \in I$, its external utility is a positive value denoted by $eu(i_l)$.

As illustrated in Table 1, $eu(c) = 5$.

Definition 3 (Utility)**.** Let $X = \{i_1, \ldots, i_k\}$ be a k-itemset with $1 \leq k \leq m$. For any transaction T_j containing X, the item-level utility is $u(i_l, T_j) =$

Table 1: External utility table

i_l	a	b	c	d	e	f
$eu(i_l)$	4	3	5	2	4	2

$eu(i_l) \times qu(i_l, T_j)$ and the itemset utility is $u(X, T_j) = \sum_{i \in X} u(i, T_j)$. For a batch B_r, $u(X, B_r) = \sum_{T_j \in B_r,\, X \subseteq T_j} u(X, T_j)$. For a sliding window W_n, $u(X, W_n) = \sum_{T_j \in W_n,\, X \subseteq T_j} u(X, T_j)$.

For example, $u(c, T_1) = 1 \times 5 = 5$, $u(c, B_1) = 1 \times 5 + 2 \times 5 = 15$, $u(c, W_1) = 15 + 35 = 50$. Similarly, $u(\{ac\}, T_1) = 1 \times 4 + 1 \times 5 = 9$, $u(\{ac\}, B_1) = 9 + 14 = 23$, and $u(\{ac\}, W_1) = 23 + 29 = 52$.

Definition 4 (Remaining utility)**.** Let $\succ$ be a total order (e.g., alphabetical order) on items from I, X be an itemset, and T/X denote the set of all items appearing after X in a transaction. The remaining utility of X in a transaction T_j is defined as $ru(X, T_j) = \sum_{i_l \in (T_j/X)} u(i_l, T_j)$.

For example, $T_1/\{ac\} = \{d, f\}$, $ru(\{a, b\}, T_1) = u(\{d, f\}, T_1) = 2 \times 2 + 1 \times 2 = 6$.

Definition 5 (Investment)**.** Each item i_l has a unit investment $cu(i_l)$ (e.g., time, energy, money). In a transaction T_j, the investment is $inv(i_l, T_j) = cu(i_l)\, qu(i_l, T_j)$. Over the stream DS, the total investment is $inv(i_l) = \sum_{T_j \in DS} inv(i_l, T_j)$. Table 2 provides the investment values for each item.

For example, Since $qu(e, T_2) = 2$ and $cu(e) = 13$, the investment of item e in transaction T_2 is calculated as $inv(e, T_2) = 2 \times 13 = 26$. The total investment of item e across all transactions is $inv(e) = inv(e, T_2) + inv(e, T_3) + inv(e, T_4) + inv(e, T_5) + inv(e, T_8) = 26 + 13 + 13 + 13 + 26 = 91$. Similarly, the total investment of item f is $inv(f) = inv(f, T_1) + inv(f, T_2) + inv(f, T_5) + inv(f, T_7) = 5 + 30 + 5 + 10 = 50$. Therefore, the investment of itemset $\{e, f\}$ is computed as $inv(\{e, f\}) = inv(e) + inv(f) = 91 + 50 = 141$.

Table 2: Investment of items

i_l	a	b	c	d	e	f
$cu(i_l)$	12	3	4	9	13	5
$inv(i_l)$	96	51	72	162	91	50

Definition 6 (Efficiency)**.** The efficiency of an itemset X within a batch B_r is defined as $E(X, B_r) = \frac{u(X,B_r)}{inv(X)}$, while its efficiency within a sliding window W_n is given by $E(X, W_n) = \frac{u(X,W_n)}{inv(X)}$.

For instance, the efficiency of the itemset $\{ac\}$ within batch B_1 is computed as $E(\{ac\}, B_1) = \frac{u(\{ac\},T_1)+u(\{ac\},T_2)}{inv(ac)} = \frac{9+14}{96+72} = 0.137$. Similarly, the efficiency of $\{ac\}$ in the sliding window W_1 is calculated by summing its utilities over batches B_1 and B_2 divided by the total investment. Hence $E(\{ac\}, W_1) = \frac{u(\{ac\},B_1)+u(\{ac\},B_2)}{inv(ac)} = \frac{23+29}{96+72} = 0.31$.

Definition 7 (Average-efficiency)**.** The average-efficiency of an itemset X in batch B_r is expressed as $AE(X, B_r) = \frac{u(X,B_r)}{inv(X)\times|X|}$, while the average-efficiency of X within the sliding window W_n is defined as $AE(X, W_n) = \frac{u(X,W_n)}{inv(X)\times|X|}$.

For example, the average-efficiency of the itemset $\{ac\}$ in batch B_1 is computed as $AE(\{ac\}, B_1) = \frac{u(\{ac\},T_1)+u(\{ac\},T_2)}{inv(ac)\times|ac|} = \frac{9+14}{(96+72)\times2} = 0.068$. The average-efficiency of the itemset $\{ac\}$ in window W_1 is computed as $AE(\{ac\}, W_1) = \frac{u(\{ac\},B_1)+u(\{ac\},B_2)}{inv(ac)\times|ac|} = \frac{23+29}{(96+72)\times2} = 0.155$.

Definition 8 (Maximum utility)**.** The maximum utility of a transaction T_j is defined as $mu(T_j) = \max\{u(i_l, T_j) \mid i_l \in T_j\}$.

For example, the maximum utility of transaction T_2 is computed as $mu(T_2) = \max\{u(i_l, T_2) \mid i_l \in T_2\} = 12$.

Definition 9 (Average-efficiency upper bound, AEUB)**.** The average-efficiency upper bound (AEUB) of an itemset X within a transaction T_j is defined as $AEUB(X, T_j) = \frac{mu(T_j)}{inv(X)}$. For a batch B_r, the AEUB of X is given by $AEUB(X, B_r) = \sum_{T_j \in B_r,\, X \subseteq T_j} AEUB(X, T_j)$, and similarly, within a sliding window W_n, it is defined as $AEUB(X, W_n) = \sum_{T_j \in W_n,\, X \subseteq T_j} AEUB(X, T_j)$.

For example, The AEUB of item a in transaction T_1 is $AEUB(a, T_1) = \frac{5}{96} = 0.052$, and the AEUB of item a in batch B_1 is $AEUB(a, B_1) = AEUB(a, T_1) + AEUB(a, T_2) = 0.052 + 0.125 = 0.177$. Finally, the AEUB of item a in window W_1 is $AEUB(a, W_1) = AEUB(a, B_1) + AEUB(a, B_2) = 0.177 + 0.26 = 0.437$.

Definition 10 (High average-efficiency itemset, HAEI)**.** In a transaction data stream DS, let $minae$ be the minimum average-efficiency threshold. An itemset X is called a high average-efficiency itemset in a sliding window W_n if its average-efficiency satisfies $AE(X, W_n) \geq minae$.

Consider the data streams shown in Figure 1 with $minae = 0.1$. For itemset $\{a, c\}$ in window W_1, we have $AE(\{a, c\}, W_1) = 0.155 \geq minae = 0.1$. Thus, $\{a, c\}$ is a HAEI in W_1.

Lemma 1. The average-efficiency of an itemset X in window W_n is always less than or equal to its average-efficiency upper bound, i.e., $AE(X, W_n) \leq AEUB(X, W_n)$.

Proof. Since the maximum utility $mu(T_j)$ satisfies $mu(T_j) \geq \frac{u(X,T_j)}{|X|}$, it follows that $AEUB(X, T_j) = \frac{mu(T_j)}{inv(X)} \geq \frac{u(X,T_j)}{|X| \times inv(X)} = AE(X, T_j)$. Thus, by summing over all relevant transactions, we have $AE(X, W_n) \leq AEUB(X, W_n)$. □

Lemma 2. For an itemset X and any of its superset X^S, it holds that $AEUB(X, W_n) \geq AEUB(X^S, W_n)$.

Proof. Since $X \subseteq X^S$, we have $inv(X) < inv(X^S)$ and $|X| < |X^S|$. Within window W_n, the number of transactions containing X is no less than that containing X^S. This implies $\sum_{T_j \in W_n \wedge X \subseteq T_j} mu(T_j) \geq \sum_{T_j \in W_n \wedge X^S \subseteq T_j} mu(T_j)$. Therefore, $AEUB(X, W_n) = \frac{\sum_{T_j \in W_n \wedge X \subseteq T_j} mu(T_j)}{inv(X)} \geq \frac{\sum_{T_j \in W_n \wedge X^S \subseteq T_j} mu(T_j)}{inv(X)} > \frac{\sum_{T_j \in W_n \wedge X^S \subseteq T_j} mu(T_j)}{inv(X^S)} = AEUB(X^S, W_n)$. □

Property 1 (AEUB Pruning)**.** If $AEUB(X, W_n) < minae$, then neither X nor any of its supersets can be HAEIs. These itemsets can be safely pruned from the search space without rescanning the database to compute their actual average-efficiency.

Definition 11 (Prefix utility)**.** For an itemset X and its extension $E(X)$ with $y \in E(X)$, the prefix utility of the extended itemset Xy in transaction T_j is given by $pu(Xy, T_j) = u(X, T_j)$.

For instance, considering the data streams depicted in Figure 1, with items ordered lexicographically as $a \prec b \prec c \prec d \prec e \prec f$. Let $X = \{a, d\}$ and $E(X) = \{e, f\}$ in transaction T_2. The prefix utility of the extended itemset $Xy = \{a, d, e\}$ is $pu(\{a, d, e\}) = u(\{a, d\}, T_2) = 14$.

Definition 12 (Prefix investment)**.** For an itemset X and its extension $E(X)$ with $y \in E(X)$, the prefix investment of the extended itemset Xy is defined as $pi(Xy) = inv(X)$.

For example, referring to the data streams in Figure 1, with items arranged in lexicographic ascending order $a \prec b \prec c \prec d \prec e \prec f$. Given $X = \{b, c\}$ and $E(X) = \{d, e, f\}$, the prefix investment of the extended itemset $Xy = \{b, c, d\}$ is $pi(\{b, c, d\}) = inv(\{b, c\}) = 123$.

Problem Statement. Given a transactional data stream DS, a minimum average-efficiency threshold *minae*, a sliding window size *winSize*, and a batch size *batchSize*, the problem of high average-efficiency itemset mining is to discover all itemsets within the current sliding window whose average-efficiency meets or exceeds *minae*, i.e., $AE(X, W_n) \geq minae$.

4. Data structure and key strategies

4.1. The relative maximum average-efficiency upper bound

To reduce the search space during average-efficiency evaluation, this section proposes the relative maximum average-efficiency upper bound (MAEUB), which is a tighter upper bound. Unlike the AEUB, which only relies on the maximum utility in a transaction, the MAEUB estimates the highest achievable average-efficiency of an item by combining it with co-occurring items while considering both their utilities and investments. This approach enables a more accurate upper bound estimation and effectively reduces unnecessary computational overhead.

Definition 13 (Maximum average-efficiency upper bound, MAEUB)**.** Given an item i_l in transaction T_j, let T_r denote the list of items in T_j, sorted in descending order according to their utilities. The relative maximum average-efficiency upper bound, denoted as $MAEUB(i_l, T_j)$, estimates the maximum average-efficiency achievable by extending i_l with additional items in T_j. It is defined as follows.

$$ae_n = \frac{u(i_l, T_j) + \sum_{k=1}^{n} u(i_k, T_r)}{(1+n) \times (inv(i_l) + \sum_{k=1}^{n} inv(i_k))}$$

$$ae_{\max} \leftarrow \max(ae_{\max}, ae_n), (u(i_{n+1}, T_r) < \frac{u(i_l, T_j) + \sum_{k=1}^{n} u(i_k, T_r)}{1+n})$$

$$MAEUB(i_l, T_j) = ae_{\max}$$

For example, after reordering the items in T_1 by descending utility, we obtain $T_r = \{(c, 5), (a, 4), (d, 4), (f, 2)\}$. We now compute MAEUB for item

$\{a\}$ in T_1. First, we compute $ae_0 = \frac{u(a,T_1)}{inv(a)\times 1} = \frac{4}{96} = 0.042$. Since $u(i_1, T_r) = 5 > \frac{u(a,T_1)}{1} = 4$, combining item a with i_1 may yield a higher average-efficiency. We then compute $ae_1 = \frac{u(a,T_1)+u(i_1,T_r)}{(inv(a)+inv(i_1))\times 2} = \frac{4+5}{(96+72)\times 2} = 0.027$. Since $ae_1 < ae_0$, the current maximum value remains ae_0. Next, we check whether further combinations can improve the average-efficiency. Since $u(i_2, T_1) = 4 < \frac{u(a,T_1)+u(i_1,T_r)}{2} = 4.5$, further extensions are not promising. Therefore, the final result is $MAEUB(a, T_1) = ae_0 = 0.042$.

Lemma 3. The MAEUB value of item i_l in transaction T_j is the maximum average-efficiency upper bound achievable by i_l and any of its supersets in T_j.

Proof. For all $0 \le j \le n$, we have $MAEUB(i_l, T_j) = ae_{\max} \ge ae_j$, indicating that $ae_{\max}$ is the maximum estimated average-efficiency among i_l and all its $(j+1)$-itemset supersets in T_j. Now consider extending i_l further with item i_{n+1}. The next estimate is

$$ae_{n+1} = \frac{u(i_l, T_j) + \sum_{k=1}^{n+1} u(i_k, T_r)}{(inv(i_l) + \sum_{k=1}^{n+1} inv(i_k)) \times (1 + n + 1)}$$

Since $u(i_{n+1}, T_r) < \frac{u(i_l,T_j)+\sum_{k=1}^{n} u(i_k,T_r)}{1+n}$, we have $inv(i_l) + \sum_{k=1}^{n+1} inv_r(i_k) > inv(i_l) + \sum_{k=1}^{n} inv_r(i_k)$, and also $\frac{u(i_l,T_j)+\sum_{k=1}^{n+1} u(i_k,T_r)}{1+n+1} < \frac{u(i_l,T_j)+\sum_{k=1}^{n} u(i_k,T_r)}{1+n}$. Thus, both the numerator decreases and the denominator increases, so $ae_{n+1} < ae_n < ae_{\max}$. Similarly, $ae_{n+2} < ae_{n+1}$, and so on. Therefore, $ae_{\max}$ is the maximum average-efficiency estimate that can be obtained by i_l and any of its supersets in T_j, which completes the proof. □

Lemma 4. The MAEUB provides a tighter upper bound compared to AEUB. Specifically, for any item i_l in transaction T_j, it holds that $MAEUB(i_l, T_j) \le AEUB(i_l, T_j)$.

Proof. If $MAEUB(i_l, T_j) \ne ae_0$, then $inv(i_l) < inv(i_l) + \sum_{k=1}^{n} inv(i_k)$, and also $\frac{\max\{u(i_l,T_j)|i_l\in T_j\}}{1} \ge \frac{u(i_l,T_j)+\sum_{k=1}^{n} u(i_k,T_r)}{1+n}$. Therefore, $AEUB(i_l, T_j) = \frac{mu(i_l,T_j)}{inv(i_l)} > ae_n$. Thus, $AEUB(i_l, T_j) > MAEUB(i_l, T_j)$.

In the special case where the maximum utility in T_j is from item i_l, and no further extension occurs, we have $MAEUB(i_l, T_j) = ae_0$, and thus $AEUB(i_l, T_j) = \frac{u(i_l,T_j)}{inv(i_l)} = ae_0 = MAEUB(i_l, T_j)$.

Hence, in all cases, $MAEUB(i_l, T_j) \le AEUB(i_l, T_j)$, which completes the proof. □

Lemma 5. The $AE(i_l, T_j)$ is always less than or equal to $MAEUB(i_l, T_j)$, it holds that $AE(i_l, T_j) \leq MAEUB(i_l, T_j)$.

Proof. Since $ae_0 = AE(i_l, T_j)$, $MAEUB(i_l, T_j) = ae_{\max} \geq ae_0$. Therefore, $MAEUB(i_l, T_j) \geq AE(i_l, T_j)$. □

Definition 14 (MAEUB in Batch and Window)**.** The MAEUB of an itemset X within a batch B_r is formally defined as $MAEUB(X, B_r) = \sum_{T_j \in B_r, X \subseteq T_j} MAEUB(X, T_j)$. Correspondingly, the MAEUB of X in a sliding window W_n is expressed as $MAEUB(X, W_n) = \sum_{B_r \in W_n} MAEUB(X, B_r)$.

For instance, Table 3 presents the MAEUB and AEUB values of items within the sliding window W_1 .

Table 3: The MAEUB and AEUB of items in W_1

i_l	a	b	c	d	e	f
$MAEUB(i_l, W_1)$	0.183	0.479	0.694	0.136	0.233	0.280
$AEUB(i_l, W_1)$	0.563	1.060	0.750	0.333	0.593	1.080

Property 2 (MAEUB Pruning)**.** Within the sliding window W_n, if an itemset X satisfies $MAEUB(X, W_n) < minae$, then none of its supersets can be considered high average-efficiency itemsets.

Proof. By definition, $MAEUB(X, T_j)$ is the maximum average-efficiency upper bound that X and its supersets can achieve in transaction T_j. Since $MAEUB(X, W_n) = \sum_{T_j \in W_n} MAEUB(X, T_j)$, this value represents the maximum average-efficiency upper bound that X and all its supersets can achieve within window W_n. If $MAEUB(X, W_n) < minae$, then for any superset X^S, we have $AE(X^S, W_n) \leq MAEUB(X^S, W_n) < MAEUB(X, W_n) < minae$. Therefore, none of the supersets X^S are high average-efficiency itemsets, and can be safely pruned. □

4.2. The estimated average-efficiency upper bound

In the list-based one-phase mining algorithms, frequent utility list join operations need to be performed. However, not all join operations are necessary, as some resulting itemsets cannot become high average-efficiency itemsets. To address this problem, this section introduces a maximum remaining average-efficiency estimation model that computes an upper bound for the average-efficiency of potential merged itemsets. This allows for safe pruning of unpromising candidates and avoids unnecessary join operations.

Definition 15 (Minimum Remaining Investment)**.** In transaction T_j, the minimum remaining investment of an itemset X is defined as $minInv(X, T_j) = \min\{inv(i_1), inv(i_2), \ldots, inv(i_l)\}$, where $i_l \in T_j/X$.

For example, in T_2, for the itemset $X = \{a, c\}$, assuming the items are ordered lexicographically, $minInv(X, T_2) = \min\{inv(d), inv(e), inv(f)\} = \min\{72, 91, 50\} = 50$.

Definition 16 (Estimated average-efficiency upper bound, EAEUB)**.** Let $X \subseteq T_j$. Denote by $u(X, T_j)$ the utility of X in T_j, by $ru(X, T_j)$ the remaining utility in T_j (under the chosen extension order), by $inv(X)$ the investment of X in T_j, and by $minInv(X, T_j)$ the minimal additional investment to extend X by one item in T_j. The per-transaction bound is

$$EAEUB(X, T_j) = \begin{cases} \frac{u(X,T_j)+ru(X,T_j)}{(inv(X)+minInv(X,T_j))\times(|X|+1)}, & ru(X, T_j) > 0 \\ 0, & ru(X, T_j) = 0 \end{cases}$$

Aggregating over a batch B_r and a sliding window W_n yields

$$EAEUB(X, B_r) = \sum_{T_j \in B_r} EAEUB(X, T_j)$$

$$EAEUB(X, W_n) = \sum_{B_r \in W_n} EAEUB(X, B_r)$$

For instance, consider the itemset $X = \{a, c\}$ in transaction T_1, where items are ordered lexicographically. The minimum investment for X in T_1 is calculated as $minInv(\{a, c\}, T_1) = \min\{inv(d), inv(f)\} = \min\{162, 50\} = 50$. Therefore, $EAEUB(\{a, c\}, T_1) = \frac{u(\{a,c\},T_1)+ru(\{a,c\},T_1)}{(inv(\{a,c\})+minInv(\{a,c\},T_1))\times(|\{a,c\}|+1)} = \frac{9+6}{(168+50)\times 3} = 0.023$. For the itemset $X = \{a, c\}$ in W_1, $EAEUB(\{a, c\}, W_1) = \sum_{T_j \in W_1} EAEUB(\{a, c\}, T_j) = 0.023 + 0.067 + 0.042 = 0.132$.

Lemma 6. For any itemset X and its extensions X^S, $EAEUB(X^S, W_n) \leq EAEUB(X, W_n)$.

Proof. For any transaction T_j, we have $u(X^S, T_j) + ru(X^S, T_j) \leq u(X, T_j) + ru(X, T_j)$. Since $|X^S| \geq |X| + 1$ and $inv(X^S) \leq inv(X) + minInv(X, T_j)$, it follows that $EAEUB(X^S, T_j) \leq EAEUB(X, T_j)$. Finally, summing over transactions and noting that $\{ T : X^S \subseteq T \} \subseteq \{ T : X \subseteq T \}$, we obtain $EAEUB(X^S, W_n) \leq EAEUB(X, W_n)$. □

Property 3 (EAEUB Pruning)**.** In window W_n, if the estimated average-efficiency upper bound of an itemset X satisfies $EAEUB(X, W_n) < minae$, then all supersets X^S are not HAEIs.

Proof. For any transaction T_j, we have $u(X^S, T_j) \leq u(X, T_j) + ru(X, T_j)$, with equality if and only if $X^S = X \cup (T_j/X)$. Since $|X^S| \geq |X| + 1$, with equality only when $X^S = X \cup \{i\}$ for some $i \in T_j/X$, and $inv(X^S) \geq inv(X) + minInv(X, T_j)$, it follows that the average-efficiency of any superset X^S is bounded by $AE(X^S, W_n) \leq EAEUB(X, W_n)$. If $EAEUB(X, W_n) < minae$, then $AE(X^S, W_n) < minae$, and thus no superset X^S can be a high average-efficiency itemset. □

4.3. Constructing TWAE-List

For the effective mining of high average-efficiency itemsets within sliding window data streams, we propose a novel list-based data structure called the TWAE-List. As shown in Figure 2, this structure integrates both window-level aggregated utility information and transaction-level utility information, enabling both efficient evaluation of high average-efficiency itemsets and rapid updates during window sliding.

Definition 17 (Utility of Old Batches)**.** In a sliding window W_n, the utility of an itemset X over the expired batches is defined as $OldU(X, W_n) = \sum_{T_j \in OldBatch \wedge X \subseteq T_j} u(X, T_j)$.

Definition 18 (Maximum Index of Old Batches in Tidlist)**.** In the TWAE-Lists constructed from a sliding window W_n , each itemset X maintains its efficiency-related information through an associated tidlist. The maximum index of transactions from old batches within this tidlist is denoted as $OldIndex(X, W_n)$.

Definition 19 (TWAE-List structure)**.** The TWAE-List of an itemset X is denoted as $AEL(X)$. As shown in Figure 2, the window-level utility information includes six key fields: $U(X, W_n)$, $EAEUB(X, W_n)$, $Inv(X)$, $PI(X)$, $OldU(X, W_n)$, and $OldIndex(X, W_n)$. These fields facilitate efficient transaction deletion during window sliding.

The transaction-level utility information is maintained in a structure called *tidlist*. This structure comprises multiple tuples $\langle T_j, U(X, T_j), RU(X, T_j), PU(X, T_j), minInv(X, T_j)\rangle$, which are utilized to record the utility, remaining utility, prefix utility, and minimum remaining investment of itemset X in each transaction.

$U(X, W_n)$	$EAEUB(X, W_n)$	$Inv(X)$	$PI(X)$	$OldU(X, W_n)$	$OldIndex(X, W_n)$
T_j	$U(X, T_j)$	$RU(X, T_j)$		$PU(X, T_j)$	$minInv(X, T_j)$

Figure 2: The TWAE-List structure of the itemset X

4.3.1. The construction of TWAE-List for 1-itemset

During the construction process of TWAE-Lists for 1-itemsets, the algorithm maintains a global list structure $listOfAELs$, which stores the TWAE-List corresponding to each item in the current sliding window. After the first batch arrives, the algorithm scans its transactions, computes MAEUB for each 1-itemset, and appends to the TWAE-List a tuple $\langle T_j, U(i_\ell, T_j), 0, 0, 0\rangle$ that records its transaction information. Once all transaction batches within the current window have arrived, the algorithm proceeds to update the TWAE-Lists. It first selects the TWAE-Lists corresponding to items with potentially high average-efficiency, then sorts them in ascending order according to the MAEUB values of the items. Subsequently, the algorithm traverses the sorted TWAE-Lists in reverse order, updating transaction and investment information in its corresponding list. The detailed procedure is presented in Algorithm 1.

Algorithm 1 describes the assignment process for 1-itemset TWAE-Lists that satisfy the minimum average-efficiency threshold condition in the current sliding window. Initially, the algorithm filters out $AEL(X)$ from the global TWAE-Lists *listOfAELs* where $MAEUB(X, W_n) \geq minae$ and adds them to *AELRecursion* (Lines 1-5). Subsequently, auxiliary arrays RU and INV are initialized to store the remaining utility and minimum investment values, with array lengths of $MaxTid+1$ (Lines 6-7). Then, the algorithm computes the boundary identifiers of the expired batch based on the values of *winNumber* and *batchSize*, resulting in *maxOldBatchTid* and *minOldBatchTid* (Lines 8-9). Next, the algorithm traverses each TWAE-List in *AELRecursion* in reverse order and iterates over each tuple *ex* in the transaction identifier list *tidlist* (Lines 10-12). If the transaction belongs to the old batch (i.e., $ex.tid \leq maxOldBatchTid$), it further checks whether it exceeds the current window range (i.e., $ex.tid \leq minOldBatchTid$). If it exceeds the window range, the tuple is removed from *tidlist* and the utility value $U(X, W_n)$ of $AEL(X)$ is updated (Lines 13-17). Otherwise, the tuple is marked as an old batch transaction, its index position *index* is recorded, and its utility value *OldU* is accumulated (Lines 18-21). Then, for each tuple, the remain-

Algorithm 1 Construct TWAE-List

Input: $listOfAELs$: global TWAE-Lists, $winNumber$: the index of current sliding window, $batchSize$: number of transactions per batch, $MaxTid$: the maximum transaction ID in the window

Output: $AELRecursion$: the set of $AEL(X)$ satisfying $MAEUB(X, W_n) \geq minae$

1: **for** each $AEL(X) \in listOfAELs$ **do**
2: **if** $MAEUB(X, W_n) \geq minae$ **then**
3: $AELRecursion \leftarrow AELRecursion \cup AEL(X)$
4: **end if**
5: **end for**
6: $RU \leftarrow$ array of zeros of length $MaxTid + 1$
7: $INV \leftarrow$ array of -1 of length $MaxTid + 1$
8: $maxOldBatchTid \leftarrow winNumber \times batchSize$
9: $minOldBatchTid \leftarrow maxOldBatchTid - batchSize$
10: **for** each $AEL(X) \in AELRecursion$ in reverse order **do**
11: $index \leftarrow 0$
12: **for** each tuple $ex \in AEL(X).tidlist$ **do**
13: **if** $ex.tid \leq maxOldBatchTid$ **then**
14: **if** $ex.tid \leq minOldBatchTid$ **then** // **Expired batch**
15: $AEL(X).U \leftarrow AEL(X).U - ex.U$
16: remove ex from $AEL(X).tidlist$
17: **end if**
18: **else**
19: $AEL(X).OldIndex \leftarrow index + 1$
20: $AEL(X).OldU \leftarrow AEL(X).OldU + ex.U$
21: **end if**
22: $ex.RU \leftarrow RU[ex.tid]$
23: **if** $INV[ex.tid] = -1$ **then**
24: $ex.minInv \leftarrow -1$
25: $INV[ex.tid] \leftarrow AEL(X).Inv$
26: **else**
27: $ex.minInv \leftarrow INV[ex.tid]$
28: **if** $ex.RU \neq 0$ **then**
29: $AEL(X).EAEUB \leftarrow AEL(X).EAEUB + \frac{ex.U+ex.RU}{(AEL(X).Inv+INV[ex.tid])\times(|X|+1)}$
30: **end if**
31: $INV[ex.tid] \leftarrow \min\{INV[ex.tid], AEL(X).Inv\}$
32: **end if**
33: $RU[ex.tid] \leftarrow RU[ex.tid] + ex.U$
34: **end for**
35: **end for**

ing utility RU is calculated (Line 22). The algorithm then checks whether the minimum remaining investment array INV has been initialized. If so, the corresponding $EAEUB$ value for ex is added to $AEL(X).EAEUB$, and the minimum remaining investment is subsequently updated (Lines 23-32). Finally, the array RU is updated (Line 33).

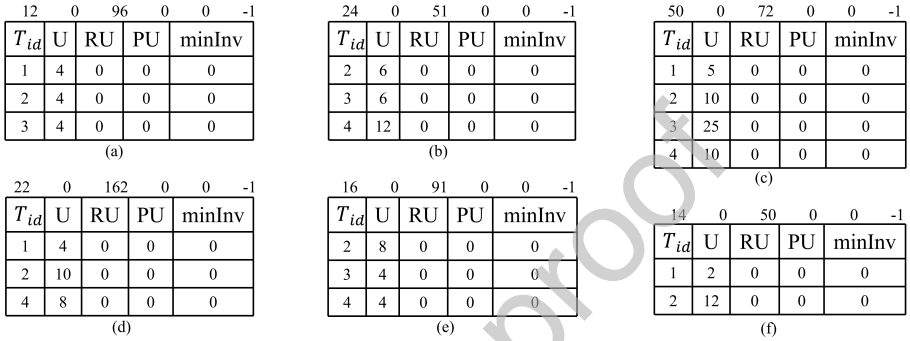


Figure 3: Global TWAE-Lists in W_1

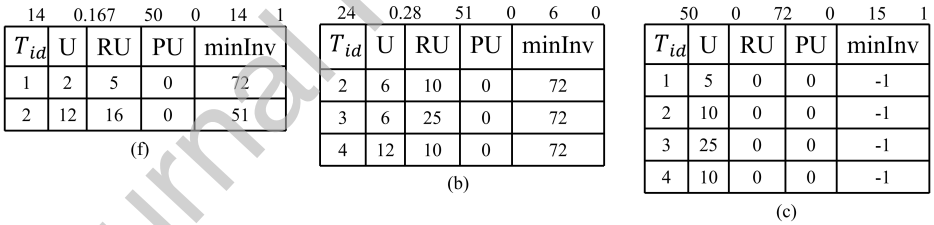


Figure 4: TWAE-Lists of possible items in W_1

For example, assume the minimum average-efficiency threshold $minae = 0.25$. According to the MAEUB pruning strategy, construct all TWAE-Lists in W_1 that satisfy $MAEUB(X, W_1) \geq minae$, and obtain $d \prec a \prec e \prec f \prec b \prec c$ in ascending order of the MAEUB. Figure 3 shows the globally initialized TWAE-List constructed during the initialization phase based on the transactions in W_1. Subsequently, the global TWAE-List is assigned according to the procedure in Algorithm 1. First, TWAE-Lists corresponding to items that satisfy the minimum average-efficiency threshold condition are filtered out. Since $MAEUB(a, W_1) = 0.183 < minae$, $MAEUB(d, W_1) =$

$0.136 < minae$, $MAEUB(e, W_1) = 0.233 < minae$, further construction of the TWAE-Lists for {a, d, e} is not required. Then, update the TWAE-Lists corresponding to the remaining items {f, b, c}. The TWAE-Lists after the final assignment operation are shown in Figure 4.

4.3.2. The construction of TWAE-List for m-itemset

Given two (m-1)-itemsets Xa and Xb, and according to the itemset order where $a \prec b$, the TWAE-List $AEL(\{Xab\})$ of the m-itemset $\{Xab\}$ can be generated by merging $AEL(Xa)$ and $AEL(Xb)$. When T_c represents the common transactions of Xa and Xb, The merged $AEL(\{Xab\})$ has the following relationships with $AEL(Xa)$ and $AEL(Xb)$.

$$AEL(\{Xab\}).Inv = AEL(Xa).Inv + AEL(Xb).Inv - AEL(Xa).PI$$

$$u(\{Xab\}, T_c) = u(\{Xa\}, T_c) + u(\{Xb\}, T_c) - PU(\{Xa\}, T_c)$$

$$ru(\{Xab\}, T_c) = ru(\{Xb\}, T_c)$$

$$PU(\{Xab\}, T_c) = u(\{Xa\}, T_c)$$

$$minInv(\{Xab\}, T_c) = minInv(\{Xb\}, T_c)$$

$$AEL(\{Xab\}).EAEUB = \sum_{T_c \in W_n} \frac{u(\{Xab\},T_c)+ru(\{Xab\},T_c)}{(AEL(\{Xab\}).Inv+minInv(\{Xb\},T_c))\times(|\{Xab\}|+1)}$$

For example, assume the minimum average-efficiency threshold $minae = 0.25$. The merging and construction processes of the TWAE-Lists for the $\{f, b\}$, $\{f, c\}$ and $\{f, b, c\}$ are shown in Figure 5.

Property 4 (LAEUB Pruning)**.** Given two itemsets X and Y in W_n, if $LAEUB(P_{xy}) = EAEUB(X, W_n) - \sum_{T_j \in W_n, X \in T_j \wedge Y \notin T_j} EAEUB(X, T_j) < minae$, then the supersets $\{XY\}^S$ of XY are not HAEIs.

Proof. Let T_{XY} denote the set of transactions where XY appears, and T_X denote the set of transactions where X appears, then $T_{XY} \subseteq T_X$. According to Property 3, we have $AE(\{XY\}^S, W_n) \leq EAEUB(XY, W_n)$.

$$AE(\{XY\}^S, W_n) \leq EAEUB(XY, W_n)$$

$$= \sum_{T_j \subseteq T_{XY}} \frac{u(XY, T_j) + ru(XY, T_j)}{(inv(XY) + minInv) \times (|XY| + 1)}$$

$$\leq \sum_{T_j \subseteq T_{XY}} \frac{u(X, T_j) + ru(X, T_j)}{(inv(X) + minInv) \times (|X| + 1)}$$

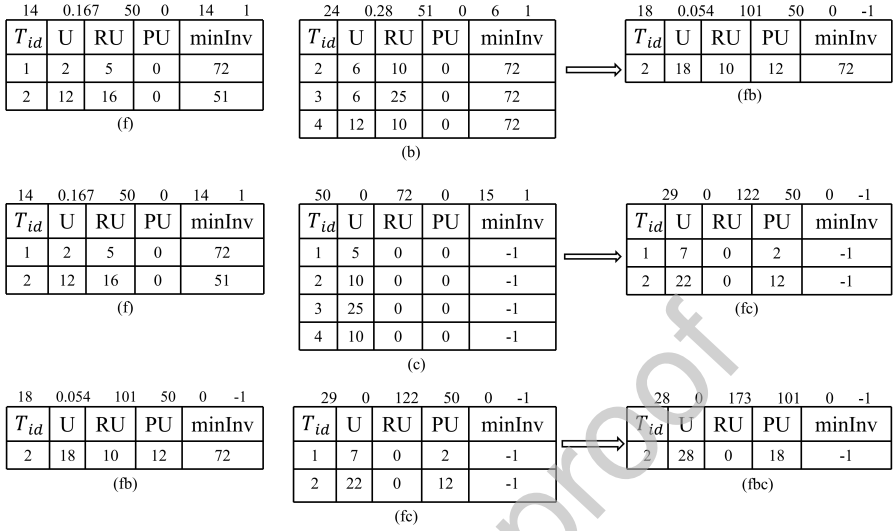


Figure 5: Example of m-itemset TWAE-list construction

$$= EAEUB(X, W_n) - \sum_{T_j \in W_n, X \in T_j \wedge Y \notin T_j} EAEUB(X, T_j) < minae$$

□

4.4. The efficient transaction pruning strategy

In sliding window-based high average-efficiency itemset mining, the performance of the algorithm heavily depends on how efficiently the TWAE-Lists are updated following each window slide. When the window slides, the TWAE-Lists from the previous window contain transactions from both the expired batch and the common batch. For transactions in the common batch, since they remain valid in the next window, the initialized 1-itemset TWAE-Lists can be directly reused. However, for transactions in the expired batch, deletion operations are required to prevent them from affecting the subsequent high average-efficiency itemset mining process.

Property 5 (The efficient transaction pruning strategy, ETP)**.** In the construction of the TWAE-List, it stores the maximum index position of the expired batch transactions (*OldIndex*) in the window-level aggregated utility information, along with the utility sum of those transactions (*OldU*).

When updating the TWAE-List, if it contains transactions from the expired batch, all entries in the window-level aggregated information from index 0 to *OldIndex* are directly removed, and *OldU* is subtracted from the total utility stored in the TWAE-List.

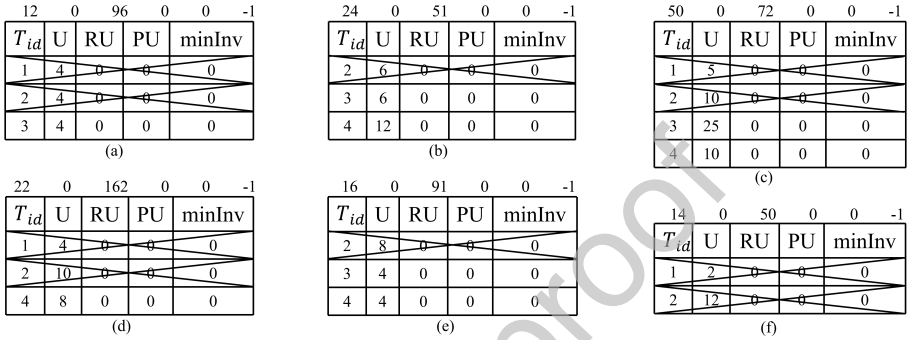


Figure 6: The TWAE-Lists after deleting the expired batch transactions

For example, when the window slides, the expired batch B_1 containing transactions T_1 and T_2 will be removed from the TWAE-Lists. The TWAE-Lists after deleting the expired batch transactions are shown in Figure 6. Taking $AEL(\{a\})$ as an example, since $OldIndex = 1$, all transactions in the *tidlist* with indices from 0 to 1 (corresponding to T_1 and T_2 from the expired batch) will be deleted. Meanwhile, we update $AEL(\{a\}).U = AEL(\{a\}).U - AEL(\{a\}).OldU = 12 - 8 = 4$, and then reset both $OldIndex$ and $OldU$ to -1 and 0 respectively.

5. Proposed algorithm

5.1. HAEIM_Stream algorithm

To address the problem of mining high average-efficiency itemsets from data streams and improve the practical applicability of the results, this paper introduces an algorithm called HAEIM_Stream. The algorithm maintains a TWAE-List structure for fast updates, employs tighter upper bounds composed of the MAEUB and EAEUB, with a decrement LAEUB for early stopping inside joins, and applies an ETP strategy that removes information from expired transactions when the window slides. These components jointly reduce candidates and list joins, yielding better runtime and memory usage. Figure 7 outlines the workflow: as batches arrive, the algorithm

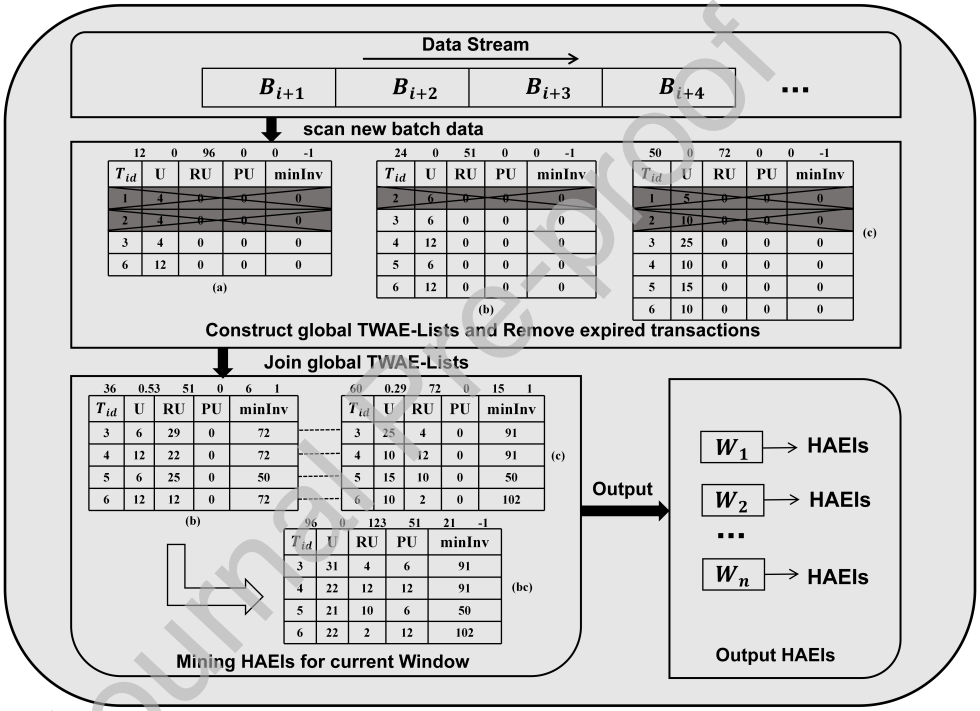


Figure 7: The overall process of HAEIM_Stream algorithm

updates global TWAE-Lists and removes expired tuples, then filters singletons by MAEUB, and mines high average-efficiency itemsets using EAEUB, LAEUB and ETP. The process covers batch processing, sliding-window maintenance, candidate generation with pruning, and finally outputs the mined high average-efficiency itemsets.

Algorithm 2 HAEIM_Stream Algorithm

Input: DS: Transaction stream, $batchSize$: Number of transactions in each batch, $winSize$: Number of batches in each window, $minae$: Minimum average-efficiency threshold

Output: $HAEIs$: All high average-efficiency itemsets in each window

1: Initialize an empty global utility list to store 1-itemset TWAE-Lists
2: Let $batchNumber$ represent the number of batches reached in the window
3: Let $batchTransaction$ represent the set of transactions already reached in the batch
4: **for all** $T_j \in DS$ **do**
5: **if** $|batchTransaction| < batchSize$ **then**
6: $batchTransaction = batchTransaction \cup T_j$
7: **end if**
8: **if** $|batchTransaction| = batchSize$ **then**
9: $batchNumber = batchNumber + 1$
10: **if** $batchNumber > winSize$ **then**
11: Call **ProcessOldBatch**()
12: **end if**
13: Call **ProcessNewBatch**($batchTransaction, batchNumber, winSize, minae$)
14: **if** $batchNumber \geq winSize$ **then**
15: Output all $HAEIs$ in the current sliding window
16: **end if**
17: $batchTransaction \leftarrow \emptyset$
18: **end if**
19: **end for**

Algorithm 2 presents the overall procedure of HAEIM_Stream over data streams. The algorithm first initializes the global repository of 1-itemset TWAE-Lists (Line 1). It then sets the batch counter and the in-batch transaction buffer (Lines 2–3). Next, it scans the stream DS and collects transactions into the buffer according to the batch size $batchSize$ (Lines 4–7). When

the buffer reaches $batchSize$, the batch counter is increased (Lines 8–9). If the number of batches exceeds the size of current window, Algorithm 3 is called to process the expired batch using the ETP strategy and update the window-level information of the TWAE-Lists (Lines 10–12). The algorithm then calls Algorithm 4 to insert the transactions of the new batch into the lists (Line 13). Once all batches in the window have arrived, the mining process starts and the HAEIs of the current window are output (Lines 14–16). Finally, the in-batch buffer is cleared and the collection of the next batch begins (Lines 17–19).

Algorithm 3 ProcessOldBatch

Input: $listOfAELs$: the list of global TWAE-Lists

1: **for all** $AEL(i_l) \in listOfAELs$ **do**
2: **if** $AEL(i_l).OldIndex \neq -1$ **then**
3: Remove all tuples from $AEL(i_l).tidlist$ with indices from 0 to $AEL(i_l).OldIndex$
4: $AEL(i_l).OldIndex = -1$
5: $AEL(i_l).U- = AEL(i_l).OldU$
6: $AEL(i_l).OldU = 0$
7: **end if**
8: **end for**

Algorithm 3 maintains the sliding window by applying the ETP strategy to the expired transactions of each 1-itemset TWAE-Lists. The transaction list $AEL(i_l).tidlist$ is stored in chronological order, the $AEL(i_l).OldIndex$ records the last position of the transaction in the expired batch, and the $AEL(i_l).OldU$ caches the total utility of those tuples. The procedure scans all TWAE-Lists (Line 1). If $AEL(i_l).OldIndex \neq -1$, it removes the expired transaction tuples from 0 to $AEL(i_l).OldIndex$ (Lines 2–3), resets $AEL(i_l).OldIndex$ (Line 4), subtracts $AEL(i_l).OldU$ from the window-level utility $AEL(i_l).U$ (Line 5), and clears $AEL(i_l)OldU$ (Lines 6–8). Since investment is maintained at the global level, no cost update is required here.

Algorithm 4 updates 1-itemset TWAE-Lists with the newly arrived batch and, once the window is complete, launches mining. Let $winNumber$ denote the numble of the current sliding window(Line 1). For each transaction T_j in the batch (Line 2), the algorithm computes MAEUB (Line 3), then initializes or updates $AEL(i_l)$ for every item $i_l \in T_j$ (Line 4), and appends a per-transaction tuple to its TWAE-List (Line 5). If the window is complete

(line 7), the window count is incremented(line 8). The algorithm then retains only the 1-itemsets whose MAEUB satisfies *minae* and calls algorithm 1 to build their TWAE-Lists (Lines 9–10). Finally, the depth-first algorithm 5 is called to mine all high average-efficiency itemsets in the current window (line 11).

Algorithm 4 ProcessNewBatch

Input: $listOfAELs$: the lists of global TWAE-Lists, $batchTransaction$: all transactions in the batch, $batchNumber$: the number of batches reached in the window, $winSize$: the number of batches in each window, $minae$: the minimum average-efficiency threshold

Output: $HAEIs$: all high average-efficiency itemsets in each window

1: Let $winNumber$ represent the current window number in the data streams
2: **for all** $T_j \in batchTransaction$ **do**
3: Calculate the MAEUB for each item in T_j
4: Initialize the AEL(i_l) for each item i_l and store it in $listOfAELs$
5: Add $\langle T_j, U(i_l, T_j), 0, 0, 0\rangle$ to $AEL(i_l).tidlist$
6: **end for**
7: **if** $batchNumber \geq winSize$ **then**
8: $winNumber+ = 1$
9: Let $AELRecursion$ represent the set of AEL(i_l) for each item i_l where $MAEUB(i_l, W_n) \geq minae$
10: $AELRecursion =$ **ConstructAEL**(listOfAELs, winNumber, batchSize, TID)
11: $HAEIs =$ **Mining**$(\emptyset, 0, AELRecursion, minae)$
12: **end if**

Algorithm 5 performs depth-first mining based on the prefix, prefix length, the extension list of TWAE-Lists, and the threshold *minae*. It iterates over each $AEL(X)$ in $AELs$ (Line 1) and lets $X.item$ denote its item (Line 2). If the average-efficiency meets $minae$(Line 3), the pattern $prefix \cup \{X.item\}$ is added to $HAEIs$ (Lines 4–5). If $AEL(X).EAEUB \geq minae$, the algorithm generates extensions in ascending MAEUB order (Line 6). It considers each $AEL(Y)$ whose index is greater than that of $AEL(X)$ (Lines 7-9), calls Algorithm 6 to construct $AEL(XY)$ (Line 10), and when a non-NULL list is returned it adds it to $exAELs$ (Lines 11–13). Then, $X.item$ is appended to $prefix$ and the procedure recurses on $exAELs$ with $prefixLength + 1$

(Lines 15–16). The process uses average-efficiency to evaluate the itemset and uses the tighter EAEUB to decide whether to continue expansion along the global order.

Algorithm 5 Mining

Input: $prefix$: the prefix itemset, $prefixLength$: the length of $prefix$, $AELs$: the TWAE-Lists for the prefix to extend, $minae$: the threshold
Output: $HAEIs$: the set of high average-efficiency itemsets with prefix

1: **for all** $AEL(X) \in AELs$ **do**
2: Let $X.item$ be the item represented by X
3: **if** $\frac{AEL(X).U}{AEL(X).Inv \cdot (prefixLength+1)} \geq minae$ **then**
4: $HAEIs \leftarrow HAEIs \cup \{X\}$
5: **end if**
6: **if** $AEL(X).EAEUB \geq minae$ **then**
7: $k \leftarrow$ index of $AEL(X)$ in $AELs$
8: $exAELs \leftarrow \emptyset$
9: **for all** $AEL(Y) \in AELs$ with index $> k$ **do**
10: $AEL(XY) \leftarrow$ **Join**$(AEL(X), AEL(Y), minae, prefixLength)$
11: **if** $AEL(XY) \neq$ NULL **then**
12: $exAELs \leftarrow exAELs \cup \{AEL(XY)\}$
13: **end if**
14: **end for**
15: $prefix[prefixLength] \leftarrow X.item$
16: **Mining**($prefix$, $prefixLength$+1, $exAELs$, $minae$)
17: **end if**
18: **end for**

Algorithm 6 joins two AELs that share the same prefix to construct the TWAE-List of XY. After initializing $AEL(XY)$ and setting $AEL(XY).item$, the procedure records the prefix investment and obtains the global investment of XY (Lines 1–3). The variable LAEUB is initialized by $AEL(X).EAEUB$ (Line 4). Subsequently, the algorithm traverses the tidlists of $AEL(X)$ and $AEL(Y)$ using a dual-pointer approach(Lines 5–7). The variables i and j are defined to serve as indices for traversing the tidlists of $AEL(X)$ and $AEL(Y)$, respectively (Line 8). If the current transaction identifiers ($tids$) are equal, compute the transaction information of the new tuple eXY and append it to $AEL(XY).tidlist$ (Lines 8–16). Then, pointers i and j are advanced si-

Algorithm 6 Join

Input: $AEL(X)$: the average-efficiency list of X, $AEL(Y)$: the average-efficiency list of Y, $minae$: the minimum average-efficiency threshold, $prefixLength$: the length of the prefix
Output: $AEL(XY)$: the average-efficiency list of XY

Initialize $AEL(XY)$
$AEL(XY).item \leftarrow Y.item$, $AEL(XY).PI \leftarrow AEL(X).Inv$
$AEL(XY).Inv \leftarrow AEL(X).Inv + AEL(Y).Inv - AEL(X).PI$
$LAEUB \leftarrow AEL(X).EAEUB$
$i \leftarrow 0, j \leftarrow 0$
while $i < |AEL(X).tidlist|$ and $j < |AEL(Y).tidlist|$ **do**
    $ex \leftarrow AEL(X).tidlist[i], ey \leftarrow AEL(Y).tidlist[j]$
    if $ex.tid = ey.tid$ **then**
        $exy.tid \leftarrow ex.tid$
        $exy.U \leftarrow ex.U + ey.U - ex.PU$
        $exy.RU \leftarrow ey.RU$
        $exy.minInv \leftarrow ey.minInv$
        Add exy to $AEL(XY).tidlist$
        if $ey.minInv \neq -1$ **then**
            $AEL(XY).EAEUB \leftarrow AEL(XY).EAEUB + \frac{exy.U+exy.RU}{(AEL(XY).Inv+ey.minInv)\times(prefixLength+3)}$
        end if
        $i \leftarrow i + 1, j \leftarrow j + 1$
    else
        if $ex.tid < ey.tid$ **then**
            if $ex.minInv \neq -1$ **then**
                $LAEUB \leftarrow LAEUB - \frac{ex.U+ex.RU}{(AEL(X).Inv+ex.minInv)\times(prefixLength+2)}$
                if $LAEUB < minae$ **then**
                    return NULL
                end if
            end if
            $i \leftarrow i + 1$
        else
            $j \leftarrow j + 1$
        end if
    end if
end while
return $AEL(XY)$

multaneously (line 17). If $ex.tid < ey.tid$, the variable LAEUB is updated (Lines 19–21). If the value of LAEUB is less than $minae$, then XY is not a HAEI, and NULL is returned immediately (Lines 22–25). Then, pointer i is moved forward (Line 26). If $ex.tid > ey.tid$, only pointer j is moved forward (Line 28). After the traversal is complete, the TWAE-List of XY is returned (Line 32).

5.2. Complexity analysis

This section examines the time complexity of the HAEIM_Stream algorithm. The entire process can be broken down into three stages: building TWAE-List for 1-itemset, constructing TWAE-List for m-itemset, and performing the recursive search. Let n_w denote the number of transactions within the sliding window, n_b denote the number of transactions per batch, and T_a denote the average transaction length. Additionally, let k represent the total number of distinct items in the sliding window. Without loss of generality, it is assumed that all items are promising. While constructing the 1-itemset TWAE-Lists, the algorithm scans every transaction in the initial sliding window to calculate the MAEUB value for each item. For subsequent sliding windows, the algorithm only scans the transactions in the newly arrived batch. Because calculating MAEUB involves sorting items in each transaction by descending utility, the time complexity for this phase is $O(n_w \times T_a \log T_a + n_w \times T_a)$ for the initial window, or $O(n_b \times T_a \log T_a + n_b \times T_a)$ for subsequent updates. Then, the algorithm sorts the candidate items based on their MAEUB values in ascending order, with a time complexity of $O(k \log k)$. Subsequently, the sorted items, together with their utility and $minInv$ values, are inserted in reverse order into the respective TWAE-Lists, and the EAEUB for each 1-itemset is calculated. This part takes $O(n_w \times T_a)$ or $O(n_b \times T_a)$ time. From the above analysis, the worst-case time complexity for building 1-itemset TWAE-Lists is $O(n_w \times T_a \log T_a + n_w \times T_a + k \log k + n_w \times T_a)$, which simplifies to $O(n_w \times T_a \log T_a + k \log k)$. In the case of batch-wise incremental updates, the time complexity becomes $O(n_b \times T_a \log T_a + n_b \times T_a + k \log k + n_b \times T_a)$, which can be simplified as $O(n_b \times T_a \log T_a + k \log k)$. In the second part, during the construction of m-itemset TWAE-Lists, the algorithm needs to find and merge transactions with common tids from the (m-1)-itemset TWAE-Lists. Assuming each (m-1)-itemset TWAE-List contains p transaction entries in its tidlist, the worst-case scenario is when p reaches the maximum value of n_w. Since TWAE-Lists utilize positional indexing, the merge process only

needs to traverse the transaction tuples in the TWAE-Lists, without relying on time-consuming binary search. Consequently, the merging operation has a worst-case time complexity of $O(n_w)$. In the final part, assume the total number of itemsets considered during recursive search is q, with a worst-case upper bound of $2^k - 1$. Given that building each itemset requires $O(n_w)$ time, the recursive search process has a time complexity of $O(q \times n_w)$. Therefore, the total time complexity of the HAEIM_Stream algorithm for each sliding window is $O(n_w \times T_a \log T_a + k \log k + q \times n_w)$. Given that q may reach an exponential scale, the time complexity per sliding window is approximately $O(q \times n_w)$.

6. Performance evaluation

This section provides an experimental evaluation of the performance of the HAEIM_Stream algorithm on various datasets. The experiments utilize four real-world datasets, including two sparse datasets (Retail and Chainstore) and two dense datasets (Chess and Accidents), all available from the SPMF repository (Fournier-Viger et al., 2014a). Regarding investment values, because many real-world phenomena are often modeled by a Gaussian distribution, we use a normal distribution $\mathcal{N}(10{,}000,\ 10{,}000^2)$ to generate these values (Zhang et al., 2023). The large variance increases the diversity of the data. If a generated investment value is negative, we regenerate it using a smaller Gaussian distribution $\mathcal{N}(100,\ 25^2)$ to ensure positivity (Zhang et al., 2023). Table 4 summarizes the properties of the datasets. Here, $|DS|$ represents the total number of transactions, $|I|$ denotes the count of distinct items, *AvgLen* indicates the average length of transactions, and *Density* is computed as the ratio of *AvgLen* to $|I|$. *Type* denotes the type of the dataset and is classified into sparse and dense according to the *Density*. All experiments were performed on a system featuring an AMD Ryzen 9 HX 370 CPU running at a base frequency of 2.0 GHz, with 32 GB of RAM, operating on Windows 11. The algorithms were implemented in Java using JDK 21.

In this section, we include a comparative baseline called HAEIMiner-DS. This variant adapts the static two-phase HAEIMiner (Yildirim, 2024) to the streaming setting and uses AEUB as the only upper bound. HAEIMiner-DS maintains no incremental state across windows, does not use MAEUB or EAEUB, and performs no expired-transaction pruning, so it rebuilds candidates and rescans the window data every time. This baseline isolates the gains brought by the MAEUB, EAEUB, and the ETP strategy. For

Table 4: Characteristics of the datasets

Dataset	$\|DS\|$	$\|I\|$	*AvgLen*	*Density*	*Type*
Retail	88,162	16,470	10.30	0.06%	sparse
Chess	3,196	75	37.00	49.33%	dense
Accidents	340,183	468	33.80	7.22%	dense
Chainstore	1,112,949	46,086	7.23	0.02%	sparse

Table 5: Strategies adopted by each algorithm variant

Algorithm	**AEUB**	**MAEUB**	**EAEUB**	**ETP**
HAEIM_Stream-1	✓			
HAEIM_Stream-2		✓		
HAEIM_Stream-3		✓	✓	
HAEIM_Stream		✓	✓	✓

our method, we ablate the proposed components to form three variants: HAEIM_Stream-1 uses AEUB as the sole upper bound, HAEIM_Stream-2 replaces AEUB with the tighter MAEUB, HAEIM_Stream-3 combines MAEUB with the EAEUB pruning, and HAEIM_Stream is the full model that further incorporates the ETP strategy. Table 5 summarizes the strategies used by each variant. This section examines the behavior of HAEIM_Stream under different minimum average efficiency thresholds, window sizes, batch sizes, and pruning configurations. The baseline and variants are evaluated in terms of runtime, memory consumption, and the number of generated candidates.

6.1. Performance evaluations with various minimum average-efficiency thresholds

To investigate the impact of the minimum average-efficiency threshold on algorithm performance, experiments were conducted under different *minae* settings while keeping the *winSize* and *batchSize* fixed. Because datasets differ in characteristics such as the number of items and average transaction length, using certain window and batch sizes can yield either too few or too many high average-efficiency itemsets, which would confound a fair comparison. Following prior work (Zhang et al., 2023; Huynh et al., 2024; Han et al., 2023), we selected window and batch sizes appropriate for the datasets to ensure comparable output results. Table 6 shows the *winSize* and *batchSize*

Table 6: *winSize* and *batchSize* settings when varying *minae*

Dataset	*winSize*	*batchSize*
Chess	3	500
Accidents	5	20,000
Retail	5	10,000
Chainstore	8	20,000

settings for each dataset. In addition, we also need to choose a suitable *minae*, so we first conducted a parameter adjustment experiment. We ran the HAEIM_Stream and HAEIMiner-DS algorithms on each dataset while decreasing the *minae* threshold until the algorithm outputted a certain number of high average-efficiency itemsets, became too slow, ran out of memory, or a clear winner emerged. Table 7 reports the parameter adjustment results on the Chess dataset. As *minae* decreases the baseline HAEIMiner-DS shows a rapid growth in runtime, memory, and candidates, while HAEIM_Stream remains stable. With $minae = 500$ the output contains only four HAEIs, which is too small for a meaningful comparison. With $minae = 50$ the output grows to 377 HAEIs and the baseline exhibits a candidate explosion, whereas HAEIM_Stream still finishes within 0.2 s and uses about 36 MB. The region $minae \in [50, 100]$ therefore offers a good balance between a non-trivial number of HAEIs and manageable cost. In the subsequent experiments we select appropriate parameters for each dataset following this principle, so that each setting yields comparable and tractable outputs.

Figure 8, Figure 9, and Figure 10 present respectively the effects of the minimum average-efficiency threshold on runtime, memory, and the number of generated candidates. When *minae* is assigned a relatively high value, the experimental outcomes may not clearly reflect the benefits of the proposed strategies. This is reasonable because a higher threshold imposes a stricter constraint. Therefore, we retuned the parameters and conducted additional experiments with smaller threshold settings. It is worth noting that setting *minae* to a very low value leads to a significant increase in the runtime of HAEIMiner-DS. To maintain visualization clarity and comparability, we terminated any runs that exceeded 12 hours and did not include those results in the figures.

As shown in Figure 8, with the continuous increase of the minimum average-efficiency threshold, both the number of high average-efficiency itemsets and the number of candidate itemsets obtained during the mining process

Table 7: Parameter adjustment results for different *minae* values on the Chess dataset

minae	HAEIs	Algorithm	Runtime (s)	Memory (MB)	Candidates
1000	0	HAEIMiner-DS	0.091	27.76	15
		HAEIM_Stream	0.077	23.36	0
500	4	HAEIMiner-DS	0.833	63.01	2875
		HAEIM_Stream	0.139	23.36	4
200	21	HAEIMiner-DS	45	340	168122
		HAEIM_Stream	0.153	29.05	179
100	83	HAEIMiner-DS	110	320	295836
		HAEIM_Stream	0.166	35.53	674
50	377	HAEIMiner-DS	885	527	1536846
		HAEIM_Stream	0.173	35.62	1547

gradually decrease, leading to a reduction in the overall runtime of the algorithms. This is primarily due to the fact that as the threshold increases, the pruning strategies of both algorithms effectively remove itemsets with no potential from the candidates, thereby reducing the search space. On both dense and sparse datasets, HAEIM_Stream exhibits significantly better runtime performance than the two-phase algorithm HAEIMiner-DS. In the dense Chess dataset, HAEIM_Stream achieves a runtime improvement of over three orders of magnitude compared to HAEIMiner-DS, and on the two sparse datasets it also reduces runtime by more than 90% on average. This is because HAEIM_Stream utilizes TWAE-List to directly calculate the actual average-efficiency of the itemset, avoiding multiple database scans. Moreover, the MAEUB and EAEUB pruning strategies further reduce the overhead of list merging, while the ETP strategy promptly removes expired information during window sliding. These combined optimizations make HAEIM_Stream the fastest variant among all considered algorithms.

When comparing the runtime of algorithms employing different pruning strategies, HAEIM_Stream-3 consistently outperforms HAEIM_Stream-1 and HAEIM_Stream-2. This is mainly because HAEIM_Stream-3 adopts more compact pruning strategies that significantly reduce the number of

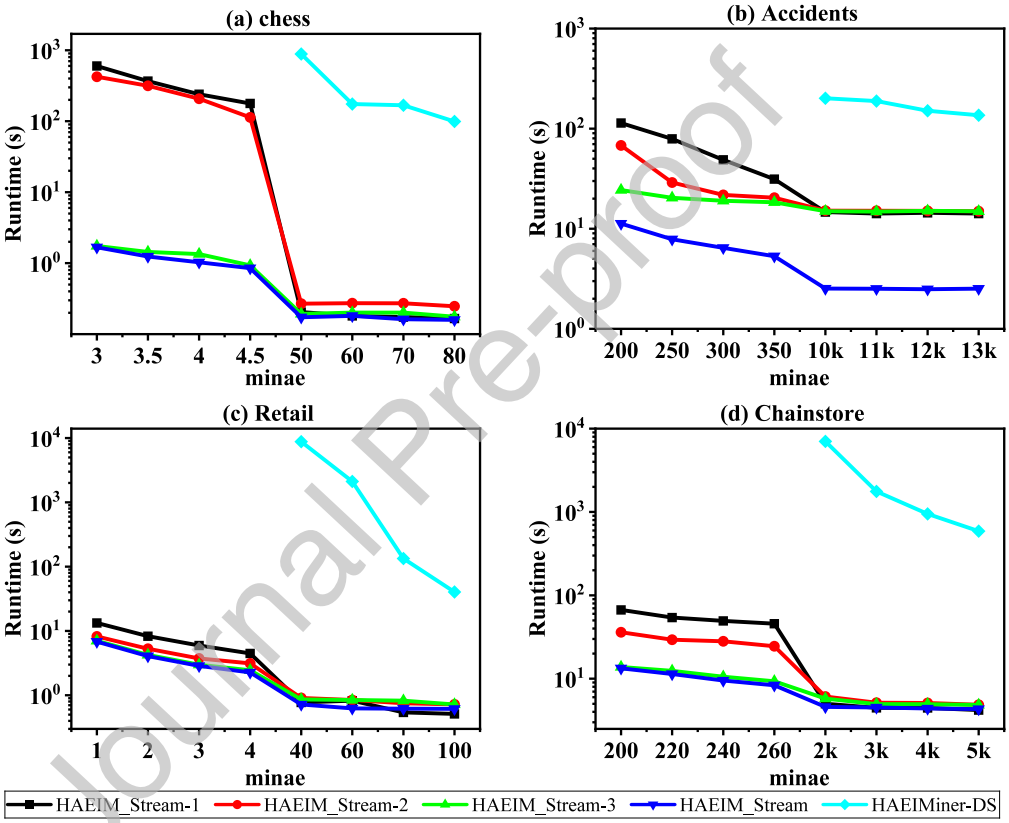


Figure 8: Runtime when varying *minae*

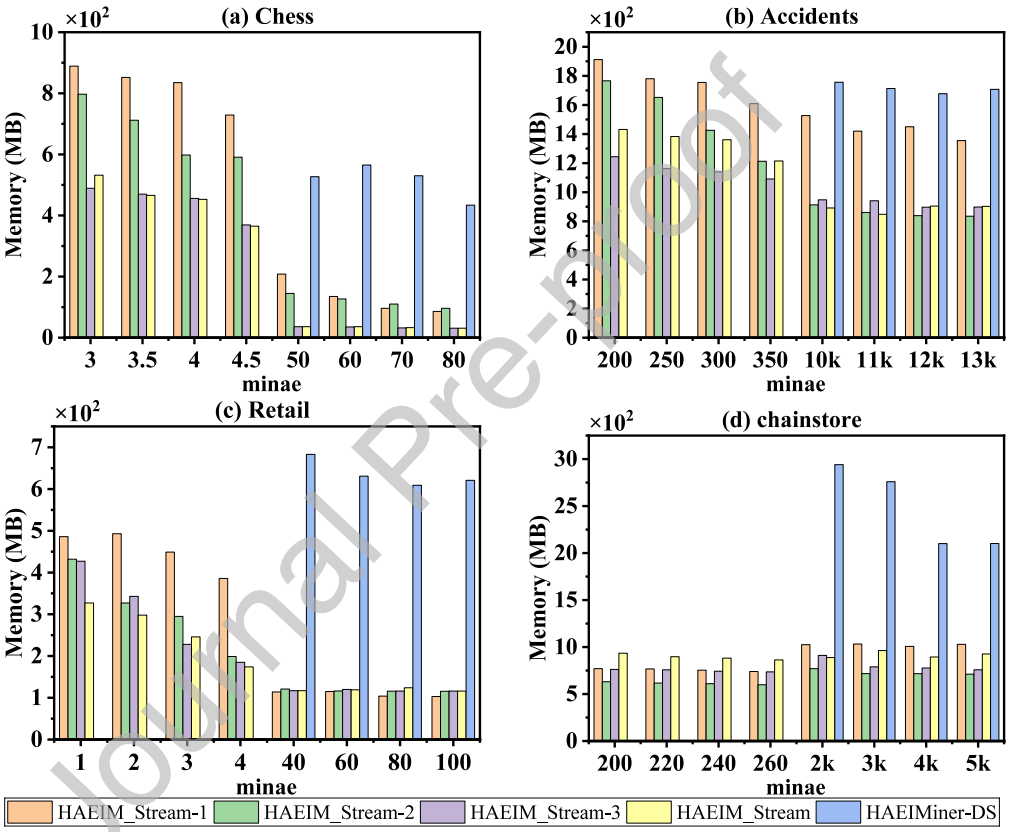


Figure 9: Memory when varying *minae*

candidate itemsets and the frequency of list merging operations. Compared with HAEIM_Stream-1, the HAEIM_Stream-2 algorithm benefits from the use of a more compact MAEUB pruning strategy instead of AEUB. As a result, HAEIM_Stream-2 produces fewer candidate itemsets and requires fewer merges, which further improves its runtime efficiency over HAEIM_Stream-1. During the mining process of high average-efficiency itemsets, the algorithm must compute the average efficiency for each candidate itemset and compare it with the minimum threshold. For promising itemsets, the algorithm further performs merging operations to construct extended itemsets. These merging operations require traversing the *tidlist* in the average-efficiency list to combine transactions with identical tids, which is computationally expensive. HAEIM_Stream-3 adopts the EAEUB pruning strategy, which effectively reduces the number of merge operations compared to HAEIM_Stream-2, and thus exhibits better runtime performance. We also observe that although the performance advantage of HAEIM_Stream over HAEIM_Stream-3 is not prominent on other datasets, mainly because the Chess dataset contains fewer transactions per batch and the other two datasets are inherently sparse. However, HAEIM_Stream consistently achieves better runtime performance on the dense dataset Accidents due to the introduction of the ETP strategy.

Figure 9 illustrates the memory usage of the algorithms. Across both dense and sparse datasets, memory usage of the algorithms generally decreases as the *minae* value rises, although some threshold settings may cause temporary increases in memory consumption. In terms of memory efficiency, HAEIM_Stream consistently consumes less memory compared to HAEIMiner-DS. This is because HAEIMiner-DS must generate and store a large number of candidate lists prior to evaluation, causing a surge in memory usage. In contrast, HAEIM_Stream builds itemset lists within the TWAE-List and employs MAEUB and EAEUB pruning strategies to quickly eliminate large numbers of unpromising candidates, thereby significantly reducing the number of TWAE-Lists and effectively lowering overall memory consumption.

A further comparison shows that on the dense Chess and Accidents datasets, HAEIM_Stream and HAEIM_Stream-3 consume significantly less memory than HAEIM_Stream-1 and HAEIM_Stream-2. This advantage stems primarily from the pruning strategies used in the HAEIM_Stream, which minimize the number of candidate itemsets and thereby reduce the memory needed to store the corresponding average-efficiency lists. However, at

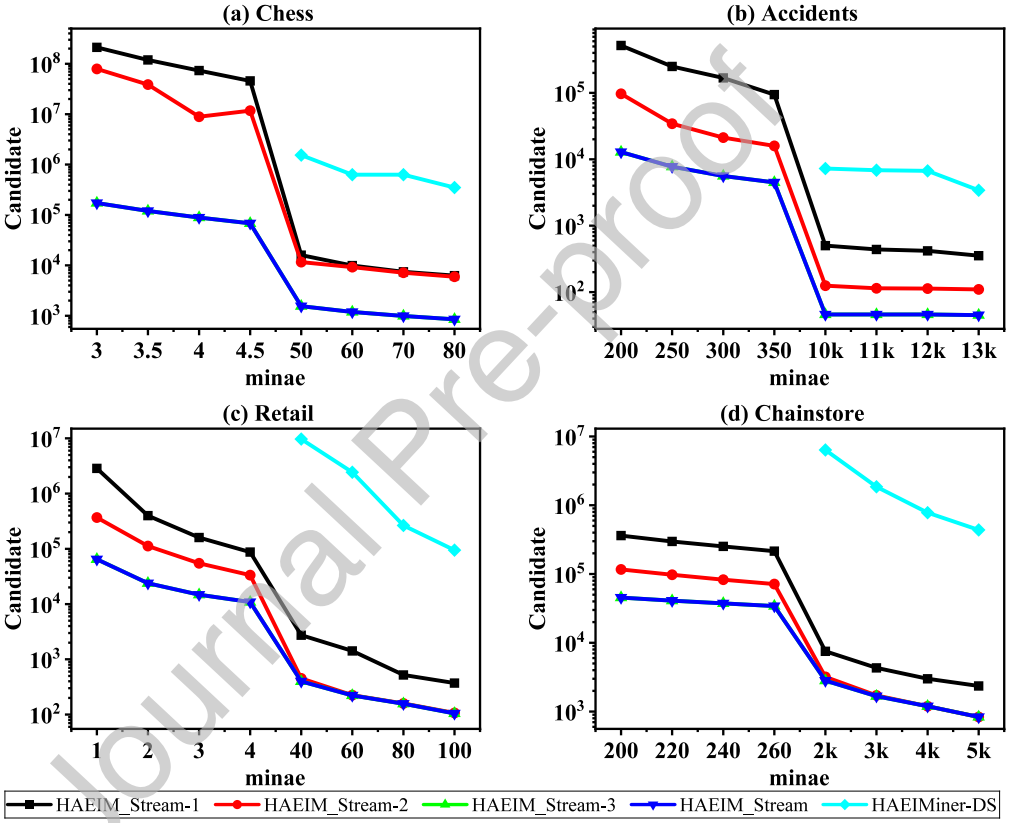


Figure 10: Candidates when varying *minae*

lower minimum threshold values, HAEIM_Stream uses slightly more memory than HAEIM_Stream-3, suggesting that the ETP strategy can lead to increased memory consumption in some cases. On the sparse Retail dataset, HAEIM_Stream also maintains a memory consumption advantage or parity compared to the other variants. In contrast, on the sparse Chainstore dataset, HAEIM_Stream requires more memory than the other three algorithms when the threshold is low. This may be due to the TWAE-List structure used by HAEIM_Stream, which stores additional transaction-level information. At lower thresholds, the algorithm maintains more candidate itemsets, increasing the volume of transactional data that must be maintained. Meanwhile, the ETP strategy may also contribute to a slight increase in overall memory consumption.

As shown in Figure 10, across all datasets, the number of generated candidates decreases monotonically with increasing *minae*, consistent with the runtime and memory trends. In our ablation scheme, HAEIM_Stream-2 consistently generates fewer candidates than HAEIM_Stream-1 because MAEUB has tighter bounds than AEUB. HAEIM_Stream-3 further reduces the candidates using the EAEUB, thereby eliminating the expensive list join operation. The candidates curve for the full HAEIM_Stream completely overlaps with that for HAEIM_Stream-3 (ETP does not change the candidates), but achieves lower runtime by avoiding processing stale tuples during window updates. Compared to the streaming baseline HAEIMiner-DS, all HAEIM_Stream variants generate significantly fewer candidates, especially on dense datasets with low minae. This explains the previously observed multi-order runtime gap and lower memory usage. On sparse datasets, the gap in candidates number remains significant, but is lessened due to their inherent sparsity. At very low thresholds, explosive growth in the candidates for HAEIMiner-DS also explains runs that exceeded 12 hours.

6.2. Performance evaluations with various window sizes

To investigate the impact of *winSize* on algorithm performance, experiments were conducted under different *winSize* settings while keeping *minae* and *batchSize* fixed. Before the formal experiments, we conducted parameter adjustment experiments based on the parameter selection principles described in the section 6.1. Table 8 presents the *minae* and *batchSize* settings for each dataset. Figure 11, Figure 12, and Figure 13 show the effects of *winSize* on runtime, memory consumption, and the number of candidates, respectively.

Table 8: *minae* and *batchSize* settings when varying *winSize*

Dataset	*minae*	*batchSize*
Chess	5	500
Accidents	400	20,000
Retail	2	10,000
Chainstore	200	20,000

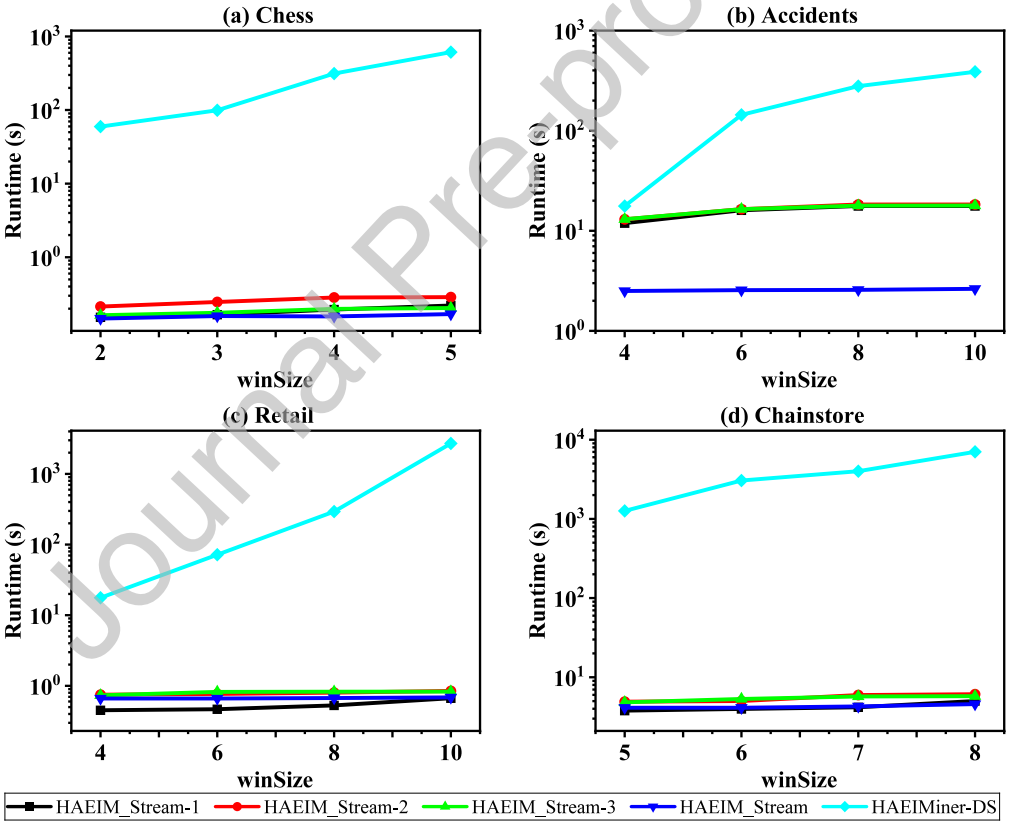


Figure 11: Runtime when varying *winSize*

As illustrated in Figure 11, the runtime of all four algorithms rises with the increase of the sliding window size, although the extent of the increase varies considerably. For the Chess dataset, when winSize increases from 2 to 5, the runtime of the two-phase algorithm HAEIMiner-DS increases by hundreds of seconds, while the runtime of HAEIM_Stream and its variants fluctuates around one second. On the Accidents dataset, the runtime of HAEIMiner-DS increases from tens of seconds to hundreds of seconds, while HAEIM_Stream remains essentially constant at around two seconds. The Retail and Chainstore datasets exhibit the same pattern. As the window size grows, the runtime of HAEIMiner-DS increases dramatically due to repeated database scans and bulk candidate generation. HAEIM_Stream and its variants leverage the TWAE-List structure, MAEUB and EAEUB pruning, and the ETP strategy to minimize list-merge and scan overhead, making the runtime essentially unaffected by the window size.

For the various HAEIM_Stream pruning strategies, the runtime of various HAEIM_Stream pruning strategies shows no notable difference on the sparse Retail and Chainstore datasets. However, on the dense Chess and Accidents datasets, HAEIM_Stream consistently outperforms the other variants. This advantage is most pronounced on the Accidents dataset, indicating that the ETP strategy plays a key role in reducing runtime when processing larger, denser transactions.

As shown in Figure 12, we compare the memory consumption of the HAEIM_Stream series with that of the two-phase algorithm HAEIMiner-DS under different sliding-window sizes. Overall, HAEIM_Stream and its variants require substantially less memory than HAEIMiner-DS at every window size, and this advantage is most pronounced on the dense datasets Chess and Accidents. For Chess, the peak memory of HAEIMiner-DS reaches several hundred megabytes, whereas HAEIM_Stream is kept within a few dozen megabytes, saving more than 90% of the memory cost. On Accidents, the gap widens further to several gigabytes. On Accidents, the gap widens to several GB. The fundamental reason is that HAEIMiner-DS needs to pre-generate and cache a large number of candidate lists, while HAEIM_Stream can significantly reduce the number of candidates using pruning strategies. On sparse datasets Retail and Chainstore, HAEIM_Stream still achieves a memory reduction of roughly $60\%-90\%$. We note that, because ETP temporarily keeps deletion marks, the memory usage of HAEIM_Stream is occasionally slightly higher than that of the variant without ETP (HAEIM_Stream-3), yet it remains far below that of HAEIMiner-DS. A further comparison of pruning

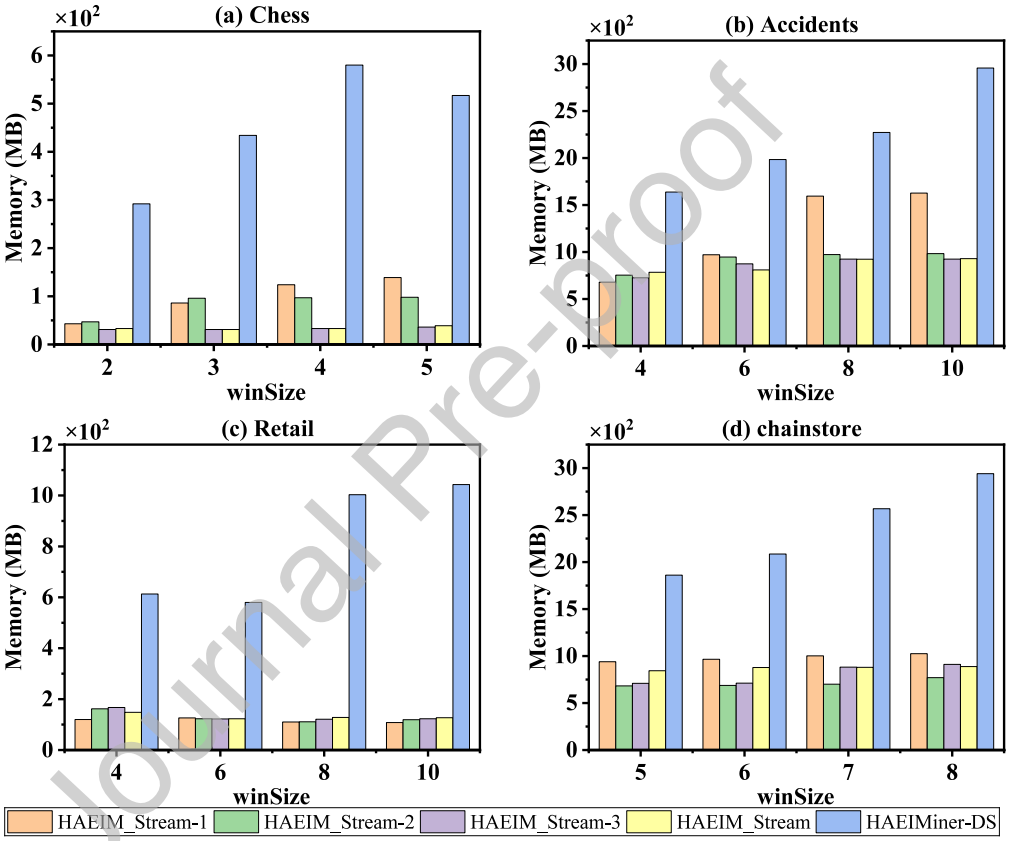


Figure 12: Memory when varying *winSize*

strategies shows that HAEIM_Stream-3 saves an additional $10\% - 20\%$ of memory relative to HAEIM_Stream-1 and HAEIM_Stream-2 for most window sizes. This suggests that tighter upper bounds and more aggressive pruning can effectively compress the number of TWAE-Lists.

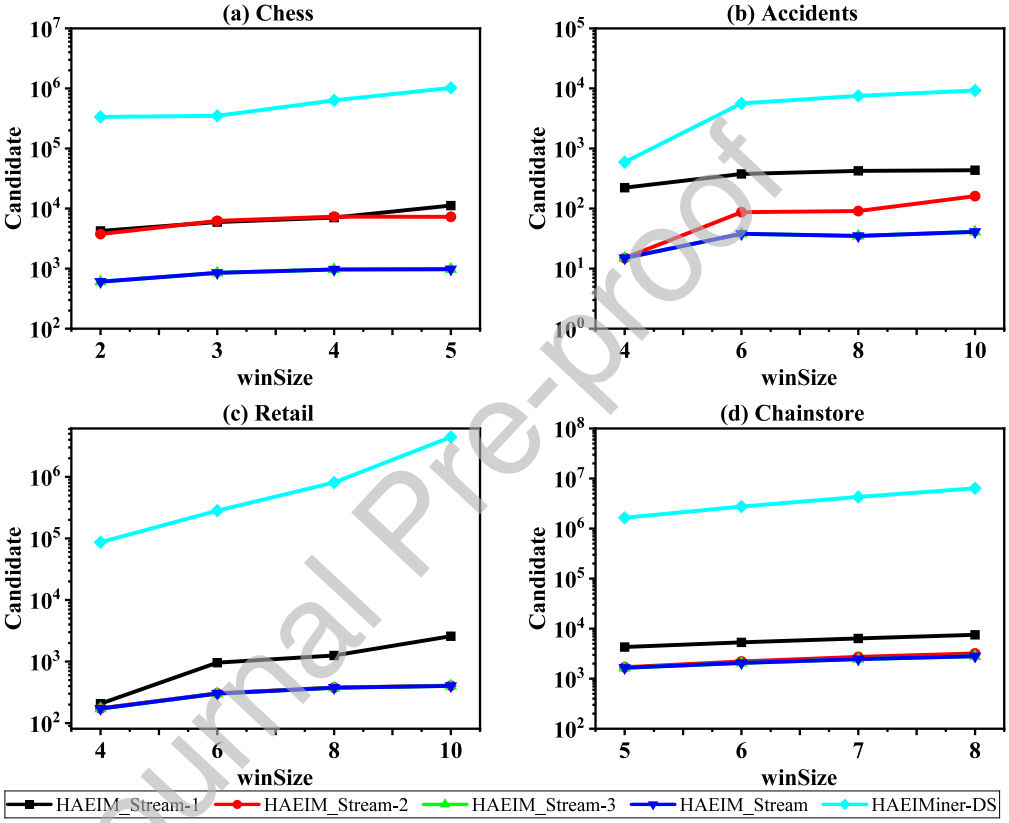


Figure 13: Candidates when varying *winSize*

Figure 13 further explains the above trends by showing the number of candidates. As *winSize* grows, candidate counts rise for all methods because more transactions per window increase co-occurrence opportunities and potential joins. The growth is especially steep for HAEIMiner-DS, which rebuilds candidates from scratch at every window and relies only on AEUB, leading to a much larger search space. In contrast, the HAEIM_Stream variants generate far fewer candidates. HAEIM_Stream-2 has already significantly lowered the curve. HAEIM_Stream-3 continues this trend by adding

EAEUB, suppressing many early connections and thus producing the fewest candidates in the ablations. The full HAEIM_Stream keeps candidate counts comparable to HAEIM_Stream-3, as ETP mainly accelerates window maintenance rather than changing candidate volume. Overall, the reduction in candidates correlates with the runtime and memory improvements observed in Figure 11 and Figure 12.

6.3. Performance evaluations with various batch sizes

To investigate the impact of *batchSize* on algorithm performance, experiments were conducted under different *batchSize* settings while keeping *minae* and *winSize* fixed. Before the formal experiments, we conducted parameter adjustment experiments based on the parameter selection principles described in the section 6.1. Table 9 presents the *minae* and *batchSize* settings for each dataset. Figure 14, Figure 15, and Figure 16 show the effects of *batchSize* on runtime, memory consumption, and the number of candidates, respectively.

Table 9: *minae* and *winSize* settings when varying *batchSize*

Dataset	*minae*	*winSize*
Chess	80	3
Accidents	15000	6
Retail	60	5
Chainstore	2000	4

As shown in Figure 14, as the batch size increases, the runtime of all algorithms rises. However, the growth rate for the HAEIM_Stream series is markedly smaller than that of the two-phase HAEIMiner-DS. Concretely, HAEIM_Stream is 1,462×, 61×, 2,181×, and 617× faster than HAEIMiner-DS on the Chess, Accidents, Retail, and Chainstore datasets, respectively. This phenomenon arises as larger batch sizes introduce more transactions per window, consequently increasing the overall workload, while HAEIM_Stream effectively reduces runtime by utilizing the TWAE-List structure.

A closer inspection of pruning strategies reveals that, under a large minimum average-efficiency threshold , HAEIM_Stream-2 (which replaces AEUB with the tighter MAEUB) can require more time than HAEIM_Stream-1. In this setting, the high *minae* already provides a stringent filter, so the

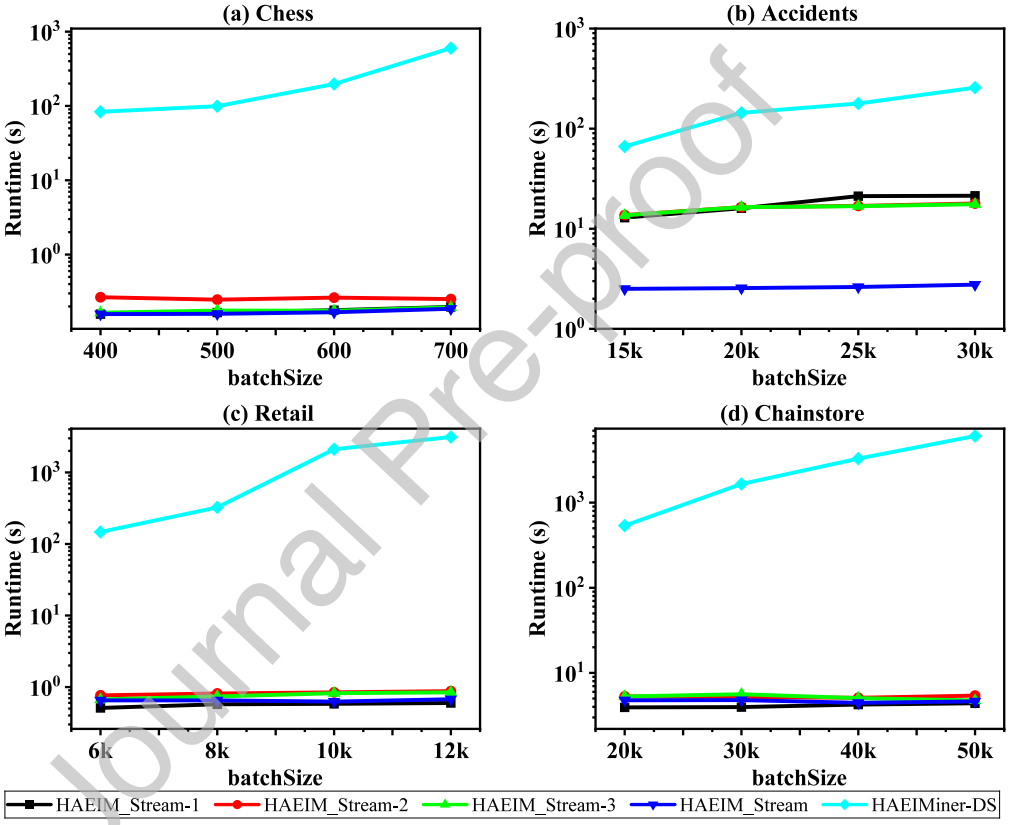


Figure 14: Runtime when varying *batchSize*

additional cost of computing MAEUB outweighs its pruning benefit. We also observe that HAEIM_Stream-3, which incorporates the EAEUB pruning strategy, runs faster than HAEIM_Stream-2, confirming the effectiveness of EAEUB in reducing runtime. After further adding the ETP strategy, the full HAEIM_Stream surpasses HAEIM_Stream-3, a superiority that is especially pronounced on the Accidents dataset.

As shown in Figure 15, memory usage grows on all four datasets as the batch size increases. However, the growth rate differs markedly among the tested methods. Due to the TWAE-List structure and the combination of MAEUB, EAEUB, and ETP strategies, the HAEIM_Stream series store candidate lists and transaction-level information more compactly. Therefore, its memory usage grows much slower compared to HAEIMiner-DS. On the dense Chess and Accidents datasets, HAEIM_Stream-1 reduces memory consumption by 93% and 64% respectively compared to HAEIMiner-DS, while the additional MAEUB and EAEUB pruning strategies reduce memory consumption by another $10\% - 20\%$. On the sparse Retail dataset, the memory footprints of all HAEIM_Stream series are nearly identical. For the Chainstore dataset, HAEIM_Stream-1 always consumes more memory than HAEIM_Stream-2, indicating that the MAEUB pruning strategy effectively cuts the number of candidates. Meanwhile, HAEIM_Stream consistently uses more memory than HAEIM_Stream-3, suggesting that the ETP strategy may introduce additional memory overhead.

Figure 16 shows the number of generated candidates. As *batchSize* increases, there are more transactions per window, increasing the likelihood of potential connections. Therefore, the number of candidates increases for all methods. The surge is most significant for HAEIMiner-DS, which rebuilds the candidate at each window and relies solely on AEUB. In contrast, HAEIM_Stream variants maintain a compact search space. HAEIM_Stream-2 significantly reduces the number of candidates by replacing AEUB with the more compact MAEUB. HAEIM_Stream-3 continues this by adding EAEUB, avoiding many early connections and further reducing the number of candidates. The full HAEIM_Stream shows the same number of candidates as HAEIM_Stream-3, as ETP primarily accelerates window maintenance rather than changing the number of candidates. These trends are consistent with the runtime and memory results in Figure 14 and Figure 15.

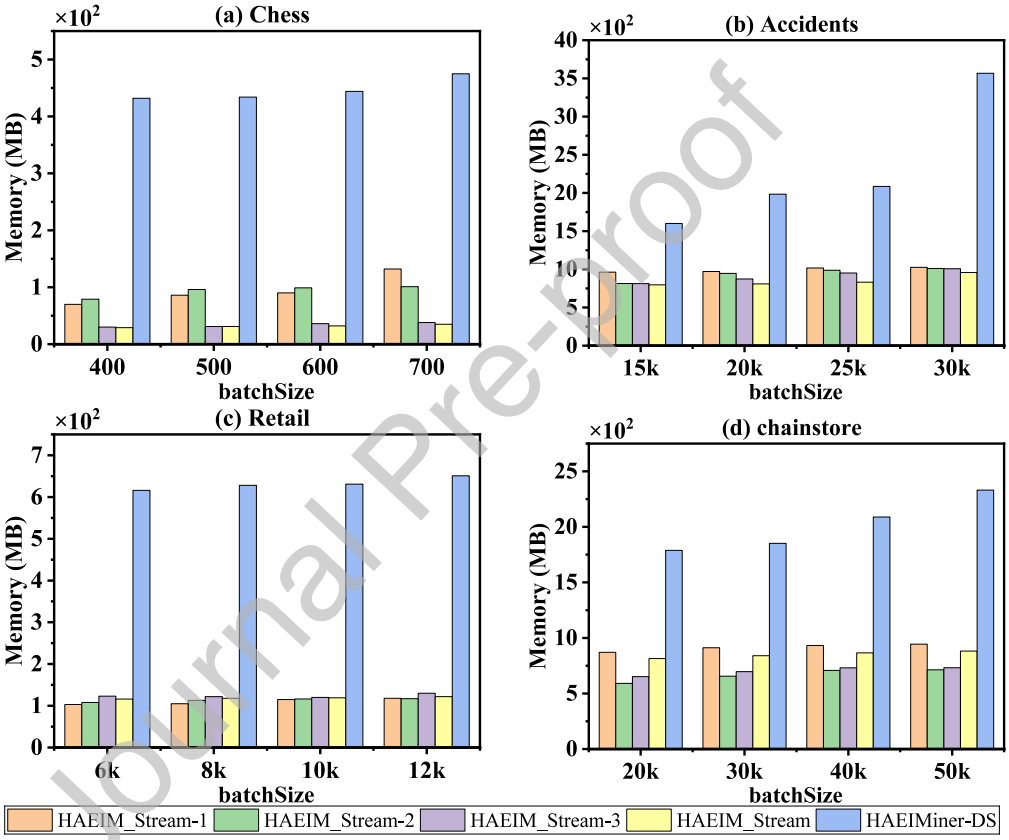


Figure 15: Memory when varying *batchSize*

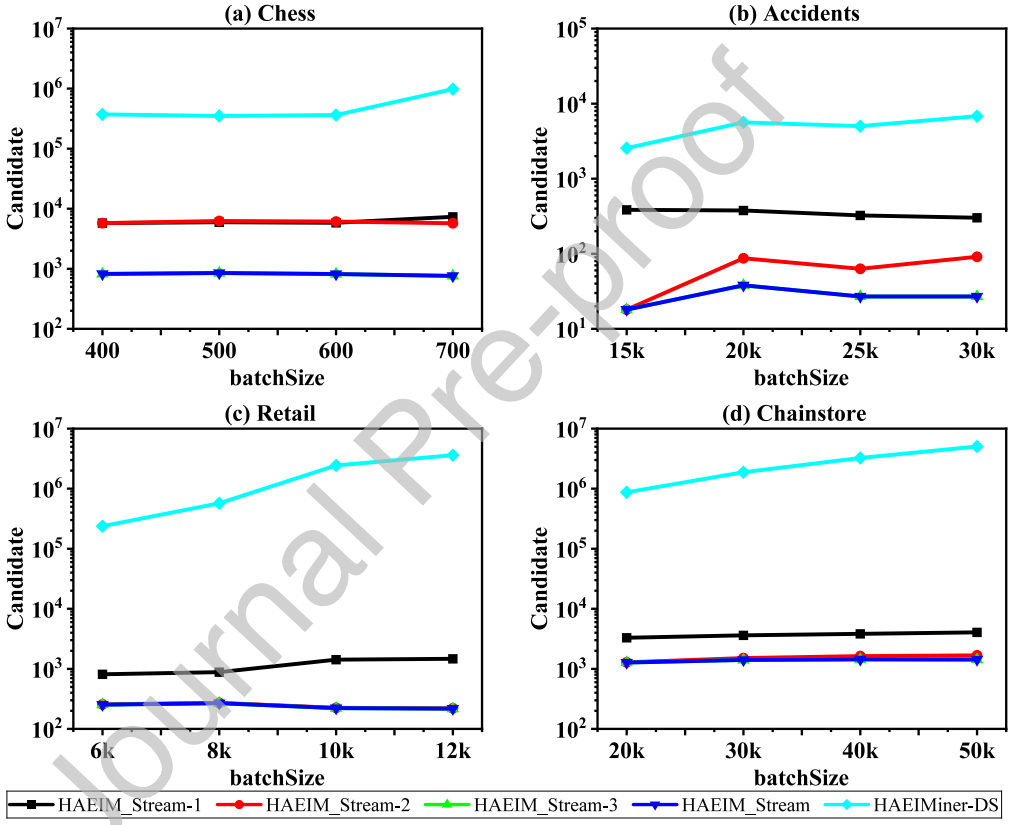


Figure 16: Candidates when varying *batchSize*

6.4. Performance evaluations of scalability

To evaluate the scalability of HAEIM_Stream, we performed three experiments. We first simulated stream scale by varying the number of transactions in the Chainstore dataset while fixing $minae = 2000$, $winSize = 8$ and $batchSize = 20k$. Specifically, we formed five scales using the first 20%, 40%, 60%, 80%, and 100% of transactions in their original order. Figure 17 shows the runtime and memory usage of HAEIM_Stream and HAEIMiner-DS under different database scales.

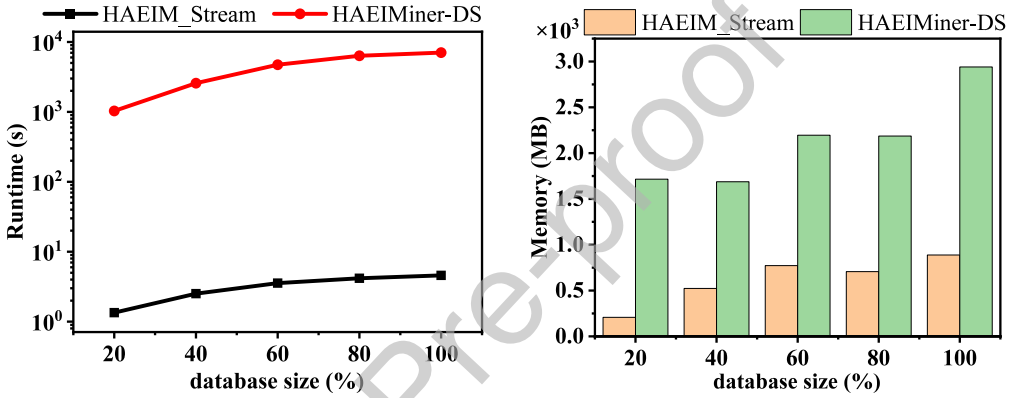


Figure 17: Runtime and memory scalability test

As the database size increases from 20% to 100%, both algorithms show an increase in runtime and memory consumption. However, HAEIM_Stream demonstrates better scalability overall. In terms of runtime, HAEIMiner-DS exhibits a significantly steeper growth compared to HAEIM_Stream, especially at larger data scales, where it shows a near-exponential increase. In contrast, HAEIM_Stream maintains a more gradual growth trend, which indicates that the algorithm can effectively reduce the number of candidate itemsets and control the complexity of the list merging operation. For memory usage, HAEIMiner-DS consistently consumes significantly more memory than HAEIM_Stream, with usage continuing to rise as the dataset grows. At the largest scale, HAEIMiner-DS consumes nearly 3 GB of memory, whereas HAEIM_Stream remains below 1 GB. This demonstrates that the TWAE-List structure adopted by HAEIM_Stream effectively reduces memory overhead. Experimental findings demonstrate that HAEIM_Stream scales well with large data streams and surpasses the baseline HAEIMiner-DS algorithm in terms of both runtime efficiency and memory usage.

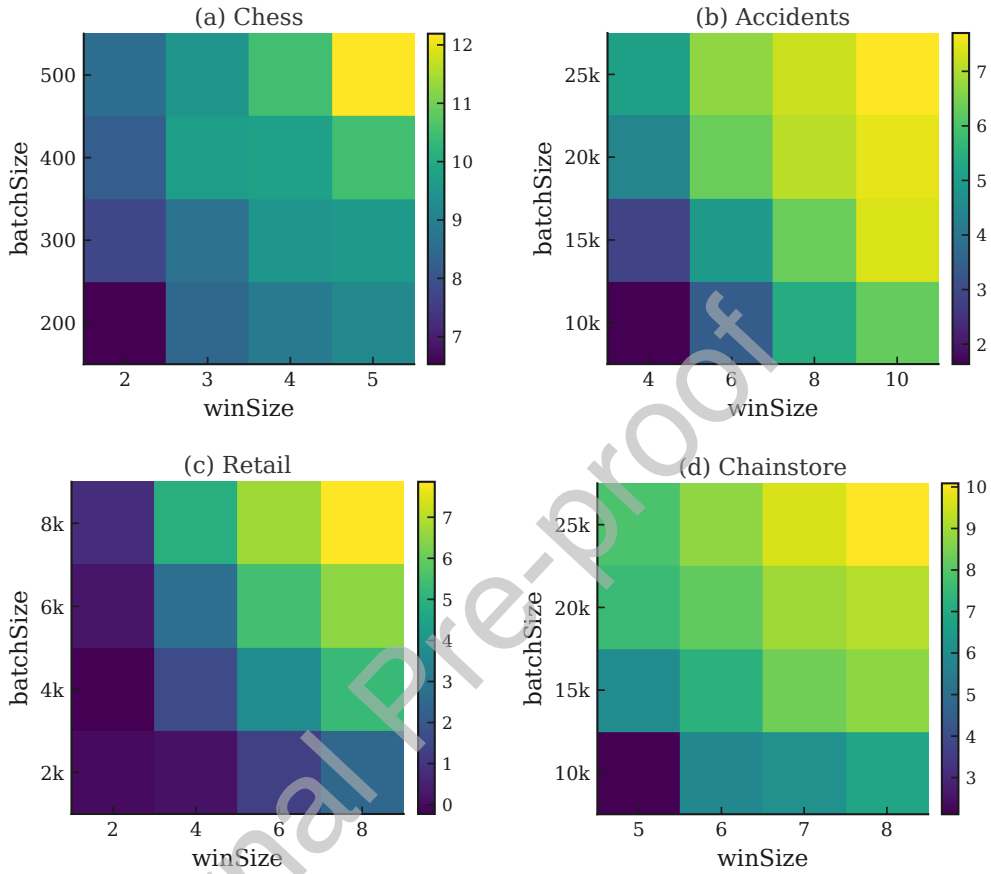


Figure 18: Runtime speedup when varying *winSize* and *batchSize*

To simulate variations in data stream intensity, we then simultaneously adjust the window size and batch size while maintaining the minimum average efficiency threshold for each dataset. Based on our setup, *minae* is fixed to 80 on Chess, 13k on Accidents, 100 on Retail, and 5k on Chainstore. We chose runtime as the evaluation metric. Since the runtimes of the two algorithms differ significantly, we plotted the base-2 logarithm of the speedup of the two algorithms. Figure 18 summarizes the results as a heat-map, with *winSize* on the horizontal axis and *batchSize* on the vertical axis.

Across all four datasets, the heat-maps exhibit a consistent pattern. As either *winSize* or *batchSize* increases, the HAEIM_Stream algorithm increasingly outperforms HAEIMiner-DS, with the gap widening further when

both parameters grow simultaneously. This behavior indicates superior scalability of HAEIM_Stream for larger windows and batches. The underlying reason is that larger windows contain more transactions, which amplifies candidate generation and repeated rescans in HAEIMiner-DS. In contrast, HAEIM_Stream utilizes multiple effective pruning strategies to suppress candidate growth, thus achieving exponential speedup in runtime.

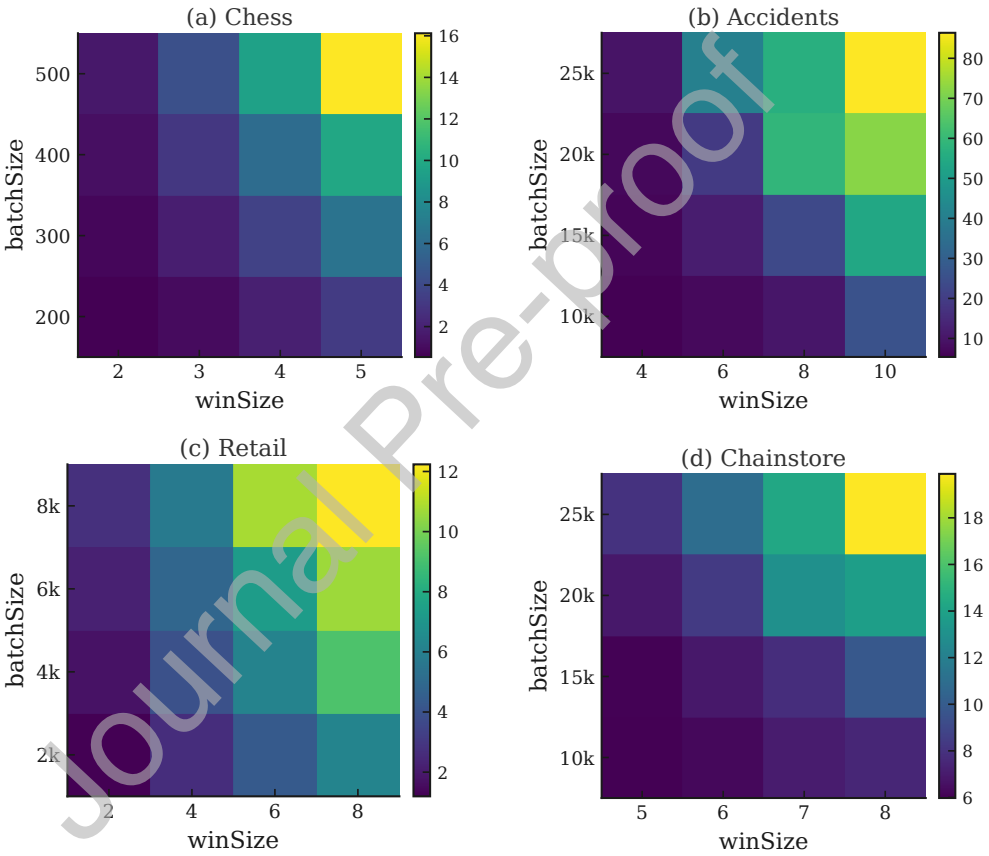


Figure 19: Runtime (s) when varying *winSize* and *batchSize*

We further decrease the minimum average-efficiency thresholds on all four datasets. Specifically, *minae* is set to 2 on Chess, 200 on Accidents, 1 on Retail, and 200 on Chainstore. Because the runtime of HAEIMiner-DS exceeds 12 hours under these settings, we report only the results for HAEIM_Stream. The outcomes are visualized as heat-maps in Figure 19, where color encodes

runtime in seconds.

Across all datasets, the heat-maps display a clear monotonic pattern. Runtime grows with either *winSize* or *batchSize*, and the increase is most pronounced when both rise simultaneously. This aligns with the fact that larger sliding windows contain more transactions, thereby enlarging the candidate space. Despite the lower *minae* implies more promising itemsets, HAEIM_Stream maintains reasonable responsiveness. The steepest gradient appears on the dense Accidents dataset, yet the scaling remains smooth across the grid. Chainstore exhibits a similar but milder trend. Retail and Chess show comparatively short runtimes with gentler growth. These results indicate that HAEIM_Stream effectively suppresses candidate growth, preventing the explosive runtime growth observed for HAEIMiner-DS, thereby achieving superior scalability.

7. Conclusions

This paper proposes a one-phase algorithm, HAEIM_Stream, based on the TWAE-List structure, to address the issues of utility evaluation in data stream high utility pattern mining that do not consider itemset length and itemset investment. Compared with the two-phase algorithm HAEIMiner-DS, the runtime and memory consumption of HAEIM_Stream are significantly reduced. In addition, the MAEUB and EAEUB pruning strategies are proposed in HAEIM_Stream to remove unpromising itemsets in advance and avoid unnecessary merging operations. The algorithm also employs the ETP strategy to delete outdated utility information of previous batches from the global average-efficiency list. Finally, a series of comparative experiments are conducted on various datasets, and the time and space performance of the algorithm is evaluated under different parameter settings. The results indicate that HAEIM_Stream achieves significantly better time and space efficiency than the HAEIMiner algorithm on both dense and sparse datasets.

However, several limitations were observed. Because TWAE-List stores per-transaction tuples, the ETP strategy may incur higher memory usage on large batches of sparse data due to the additional maintenance data. When *minae* is already high, the additional cost of computing MAEUB may outweigh the benefits of pruning. The cost model assumes global, positive, and stationary inputs, ignoring cost variations and negative cost cases. Future work will continue to explore more effective upper bounds and list structures.

Meanwhile, extending HAEIM_Stream to top-k or adapting it to structured streams such as sequences and graphs could broaden its applicability.

Gufeng Li

https://orcid.org/0000-0003-4877-5975

Works (7 of 7)

Efficient high utility itemsets mining over data streams with compact utility list structure
Applied Intelligence
2025-08 | journal-article
DOI: 10.1007/s10489-025-06729-2
Source: Crossref

HUPSP-LAL: Efficiently mining utility-driven sequential patterns in uncertain sequences
Expert Systems with Applications
2025-04 | journal-article
DOI: 10.1016/j.eswa.2025.126536
Source: Crossref

List-based mining top-k average-utility itemsets with effective pruning and threshold raising strategies
Applied Intelligence
2023-11 | journal-article
DOI: 10.1007/s10489-023-04864-2
Source: Crossref

Efficient mining high average-utility itemsets with effective pruning strategies and novel list structure
Applied Intelligence
2022-07-06 | journal-article
DOI: 10.1007/s10489-022-03722-x
Source: Crossref

Microwave photonics broadband unambiguous frequency measurement based on a Sagnac loop and a linear optical filter
Applied Optics
2022-06-10 | journal-article
DOI: 10.1364/AO.456821
Source: Crossref

Photonic Microwave Up-Conversion Link With Compensation of Chromatic Dispersion-Induced Power Fading

IEEE Photonics Journal

2019-08 | journal-article

DOI: 10.1109/JPHOT.2019.2928031

Source: Crossref

SFDR and gain enhancement in photonic downconversion link by linearization and full spectrum utilization

Applied Optics

2019-01-20 | journal-article

DOI: 10.1364/AO.58.000579

Source: Crossref

Peer review (1)

• review activity for **Knowledge-based systems. (2)**

Record last modified Jul 15, 2025, 2:03:38 PM

Credit Author Statement

Gufeng Li: Conceptualization, Writing - Original Draft, Methodology. Shuo Chen: Writing- Original Draft, Software, Data Curation. Xuanwei Zhang: Investigation, Formal analysis, Validation. Tao Shang: Writing - review & editing, Project administration, Supervision.

Declaration of interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is supported by Conceptual Verification Foundation of Xidian University Hangzhou Institute of Technology [grant numbers GNYZ2024GY004].

References

Agrawal, R. and Srikant, R. (1998). Fast algorithms for mining association rules.

Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., and Choi, H. (2012a). Interactive mining of high utility patterns over data streams. *Expert Syst. Appl.*, 39:11979–11991.

Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., and Choi, H. (2012b). Interactive mining of high utility patterns over data streams. *Expert Syst. Appl.*, 39:11979–11991.

Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., and Choi, H. (2012c). Interactive mining of high utility patterns over data streams. *Expert Syst. Appl.*, 39:11979–11991.

Chen, J., Liu, A., Zhang, H., Yang, S., Zheng, H., Zhou, N., and Li, P. (2024). Improved adaptive-phase fuzzy high utility pattern mining algorithm based on tree-list structure for intelligent decision systems. *Scientific Reports*, 14.

Cheng, Z., Fang, W.-H., Shen, W.-Y., Lin, J. C.-W., and Yuan, B. (2022). An efficient utility-list based high-utility itemset mining algorithm. *Applied Intelligence*, 53:6992–7006.

Chu, C.-J., Tseng, V. S., and Liang, T. (2008). An efficient algorithm for mining temporal high utility itemsets from data streams. *J. Syst. Softw.*, 81:1105–1117.

Dawar, S., Sharma, V., and Goyal, V. (2017). Mining top-k high-utility itemsets from a data stream under sliding window model. *Applied Intelligence*, 47:1240 – 1255.

Djenouri, Y., Belhadi, A., Fournier-Viger, P., and Lin, C.-W. (2018). Fast and effective cluster-based information retrieval using frequent closed itemsets. *Inf. Sci.*, 453:154–167.

Duong, Q.-H., Fournier-Viger, P., Ramampiaro, H., Nørvåg, K., and Dam, T.-L. (2018). Efficient high utility itemset mining using buffered utility-lists. *Applied Intelligence*, 48:1859–1877.

Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.-W., and Tseng, V. S. (2014a). Spmf: a java open-source pattern mining library.

Fournier-Viger, P., Wu, C.-W., Zida, S., and Tseng, V. S. (2014b). Fhm: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In *International Syposium on Methodologies for Intelligent Systems*.

Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. In *ACM SIGMOD Conference*.

Han, M., Li, M., Chen, Z., Wu, H., and Zhang, X. (2023). High utility pattern mining algorithm over data streams using ext-list. *Applied Intelligence*, 53:27072–27095.

Hong, T.-P., Lee, C.-H., and Wang, S.-L. (2011). Effective utility mining with the measure of average utility. *Expert Syst. Appl.*, 38:8259–8265.

Huang, H., Chen, S., and Chen, J. (2024). Iphm: Incremental periodic high-utility mining algorithm in dynamic and evolving data environments. *Heliyon*, 10.

huy Duong, Q., Fournier-Viger, P., Ramampiaro, H., Nørvåg, K., and Dam, T.-L. (2017). Efficient high utility itemset mining using buffered utility-lists. *Applied Intelligence*, 48:1859 – 1877.

Huynh, B., Tung, N. T., Nguyen, T. D. D., Bui, Q.-T., Nguyen, L. T. T., Yun, U., and Vo, B. (2024). An efficient strategy for mining high-efficiency itemsets in quantitative databases. *Knowl. Based Syst.*, 299:112035.

Jaysawal, B. P. and Huang, J.-W. (2020). Sohupds: a single-pass one-phase algorithm for mining high utility patterns over a data stream.

Jaysawal, B. P. and Huang, J.-W. (2024). Sohupds+: An efficient one-phase algorithm for mining high utility patterns over a data stream. *ACM Transactions on Knowledge Discovery from Data*, 19:1 – 32.

Kim, H., Kim, H., Cho, M., Vo, B., Lin, C.-W., Fujita, H., and Yun, U. (2023). Efficient approach of high average utility pattern mining with indexed list-based structure in dynamic environments. *Inf. Sci.*, 657:119924.

Kim, H., Yun, U., Baek, Y., Kim, J., Vo, B., Yoon, E., and Fujita, H. (2021). Efficient list based mining of high average utility patterns with maximum average pruning strategies. *Inf. Sci.*, 543:85–105.

Krishnamoorthy, S. (2015). Pruning strategies for mining high utility itemsets. *Expert Syst. Appl.*, 42:2371–2381.

Krishnamoorthy, S. (2017). Hminer: Efficiently mining high utility itemsets. *Expert Syst. Appl.*, 90:168–183.

Kumar, M. J. K. and Rana, D. P. (2021). High average-utility itemsets mining: a survey. *Applied Intelligence*, 52:3901 – 3938.

Kumar, R. and Singh, K. (2023). High utility itemsets mining from transactional databases: a survey. *Applied Intelligence*, 53:27655–27703.

Lan, G.-C., Hong, T.-P., and Tseng, V. S. (2012). A projection-based approach for discovering high average-utility itemsets. *J. Inf. Sci. Eng.*, 28:193–209.

Lee, C., Ryu, T., Kim, H., Kim, H., Vo, B., Lin, C.-W., and Yun, U. (2022). Efficient approach of sliding window-based high average-utility pattern mining with list structures. *Knowl. Based Syst.*, 256:109702.

Li, G., Shang, T., and Zhang, Y. (2022). Efficient mining high average-utility itemsets with effective pruning strategies and novel list structure. *Applied Intelligence*, 53:6099–6118.

Li, H.-F., Huang, H.-Y., Chen, Y.-C., Liu, Y.-J., and Lee, S.-Y. (2008). Fast and memory efficient mining of high utility itemsets in data streams. pages 881–886.

Lin, C.-W., Li, T., Fournier-Viger, P., Hong, T.-P., Zhan, J. Z., and Voznák, M. (2016). An efficient algorithm to mine high average-utility itemsets. *Adv. Eng. Informatics*, 30:233–243.

Lin, C.-W., Ren, S., Fournier-Viger, P., Hong, T.-P., Su, J.-H., and Vo, B. (2017a). A fast algorithm for mining high average-utility itemsets. *Applied Intelligence*, 47:331–346.

Lin, J. C.-W., Ren, S., Fournier-Viger, P., and Hong, T.-P. (2017b). Mining of high average-utility itemsets with a tighter upper-bound model. In *International Conference on Multidisciplinary Social Networks Research*.

Liu, M. and Qu, J.-F. (2012). Mining high utility itemsets without candidate generation.

Liu, Y., keng Liao, W., and Choudhary, A. N. (2005). A two-phase algorithm for fast discovery of high utility itemsets. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*.

Ryang, H. and Yun, U. (2016). High utility pattern mining over data streams with sliding window technique. *Expert Syst. Appl.*, 57:214–231.

Tung, N. T., Nguyen, L. T. T., Nguyen, T. D. D., and Huynh, B. (2025). Efficient mining top-k high utility itemsets in incremental databases based on threshold raising strategies and pre-large concept. *Knowl. Based Syst.*, 315:113273.

Uno, T., Kiyomi, M., and Arimura, H. (2004). Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Workshop on Frequent Itemset Mining Implementations*.

Yan, Y., Niu, X., Zhang, Z., Fournier-Viger, P., Ye, L., and Min, F. (2024). Efficient high utility itemset mining without the join operation. *Inf. Sci.*, 681:121218.

Yildirim, I. (2024). Mining high average-efficiency itemsets. pages 1–6.

Yildirim, I. (2025a). An efficient algorithm for fast discovery of high-efficiency patterns. *Knowl. Based Syst.*, 313:113157.

Yildirim, I. (2025b). Mining high-efficiency itemsets with negative utilities. *Mathematics*.

Yun, U. and Kim, D. (2017). Mining of high average-utility itemsets using novel list structure and pruning strategy. *Future Gener. Comput. Syst.*, 68:346–360.

Yun, U., Kim, D., Ryang, H., Lee, G., and Lee, K.-M. (2016). Mining recent high average utility patterns based on sliding window from stream data. *J. Intell. Fuzzy Syst.*, 30:3605–3617.

Yun, U., Lee, G., and Yoon, E. (2017). Efficient high utility pattern mining for establishing manufacturing plans with sliding window control. *IEEE Transactions on Industrial Electronics*, 64:7239–7249.

Zhang, X., Chen, G., Song, L., Gan, W., and Song, Y. (2023). Hepm: High-efficiency pattern mining. *Knowl. Based Syst.*, 281:111068.

Zida, S., Fournier-Viger, P., Lin, C.-W., Wu, C.-W., and Tseng, V. S. (2015). Efim: A highly efficient algorithm for high-utility itemset mining. In *Mexican International Conference on Artificial Intelligence*.