Scalable High-Utility Pattern Mining from Data Streams

by

Jiaxing Mai

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

August 29, 2022

Thesis advisor
Dr. Carson K. Leung

Author
Jiaxing Mai

Scalable High-Utility Pattern Mining from Data Streams

Abstract

Traditional high-utility mining mainly focuses on improving the efficiency of discovering high utility patterns from static databases based on a simplified assumption that the unit utility for a given item is a constant. However, not much research effort has been put into mining dynamic profit from data stream yet. The emergence of big data has led to some performance challenges such that a proper big data management technique is required to discover useful knowledge from the dynamic data streams. Traditional static data mining algorithms cannot directly apply to dynamic data. Furthermore, as information in the data stream might not be uniformly distributed, it introduces extra challenges to process the data. To mine real-world data streams, it is logical to use big data stream processing frameworks. Leveraging these big data processing frameworks requires having scalable algorithms. Hence, for my MSc thesis, I design and develop a high utility data stream framework to speed up the execution time and be flexible to adapt to mining requirement after data are dynamically modified. Utilizing our proposed algorithm, the data stream mining performance is expected to be further enhanced against both synthetic and real-world datasets.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I sincerely give thanks to my research supervisor, Dr. Carson K. Leung, for his encouragement and academic support. I had taken Dr. Leung's database courses since my undergraduate studies. After I completed my undergraduate degree, I decided to continue to extend on knowledge into the area of data mining. The research has been a long journey and this M.Sc thesis would not have been achievable without the guidance and support from Dr. Leung.

I would also like to thank my MSc thesis examination committee members—Dr. Saman Muthukumarana (Statistics) and Dr. Shaowei Wang (Computer Science)—for reviewing my thesis and provide valuable feedback. I would also thank my advisory committee members—Dr. Shahin Kamali (who recently moved from Department of Computer Science at University of Manitoba to Department of Electrical Engineering and Computer Science at York University) and Dr. Xikui Wang (who moved from Department of Statistics in Faculty of Science to Warren Centre for Actuarial Studies and Research in I.H. Asper School of Business, both within University of Manitoba)—for reviewing the proposal of this thesis research.

Jiaxing Mai

B.Sc., The University of Manitoba, Canada, 2015

The University of Manitoba
August 29, 2022

Chapter 1

Introduction

In the current era of big data, high volume and different veracity of data can be generated at high speed from heterogeneous data sources. Advancements in networking, parallel computing, and mobile computing generate a large amount of data, and mining dynamic data stream brings up several fundamental challenges such as high volume, velocity and variety. The amount and complexity of data exceeded human beings analytic capability but valuable information could be embedded waiting to be discovered. According to the report from IBM Cloud [Mar10] in 2016, 90% of the digital data was created in the past two years. Hence, big data management techniques and big data science solutions are required to handle the exponential growth of data. Intending to extract knowledge from the subset of the data stream, there are different variations of processing models such as the sliding window model, landmark model, and time-fading model.

A high utility dynamic stream mining algorithm can be implemented via Apriori-based, Tree-based or List-based. SPMF [FGG+14], an open-source data mining library, published the experiment result of finding the most efficient high utility pattern mining algorithm.

Among EFIM [ZFL+17], FHM [FWZT14], HUI-Miner [WFGT15], HUP-Miner [Kri15], D2HUP [LWF16] and UP-Growth [TWSY10], the most efficient algorithm is EFIM for mining static databases. Although the data structure used by these algorithms and the performance result cannot be directly correlated to stream mining, it indicates the advantage of the list-based mining algorithm in the HUI domain.

Mining high utility data stream exposes new challenges and opportunities. Traditional high utility mining research is devoted to simplified scenarios assuming:

1. data can fit into the main memory, and
2. the external unit profit is a constant.

However, the real world scenario could be more complicated because:

1. it is the common requirement to get real-time feedback by processing data streams, and
2. the utility profit could fluctuate based on different factors such as the timestamp.

If the utility information is changed at a certain timestamp, the mining result should be synchronized with the updated information rather than re-scan the data stream. There are two main data processing models to process continuous data which are stream processing and batch processing. For stream processing, data is immediately ingested when it arrives. It is used for processing continuous arriving data; for batch processing, data is temporarily stored in a storage system for a specific amount of time, then apply traditional data mining algorithms to each batch. The downside of batch processing is that there is no real-time interactive feedback. In my thesis, I would focus on stream processing.

Hence, for my MSc thesis, I propose to design and develop an efficient high utility data stream algorithm for processing data stream called Enhanced-Stream-HUI-Miner.

1.1 Thesis Statement

More specifically, in this MSc thesis, *I design and develop a data structure called Enhanced-Utility-List (EUL).* By leveraging on the data structure, I design and develop a scalable stream mining algorithm for discovering high utility patterns from dynamic streams, such algorithm can handle mining process when the price is changed without re-scanning databases. Having such an algorithm is crucial for enhancing the feature and improving the performance of the mining process in various real-life data science applications.

In terms of **motivation**, traditional high utility mining assumes the external unit profit of the item in a transaction database is fixed which is not a practical assumption. The simplified assumption reduces the complexity of the HUI-Mining problem since the mining result would not need to be revisited and modified. Recent research starts to explore the dynamic unit profit scenarios but those algorithms are only applicable for mining static databases. To the best of our knowledge, no research work has been done for stream mining on dynamic profit data but it is necessary to study this scenario to fill the gap of analysing the growing volume of data streams.

More formally, my **research problem** can be defined as follows. Given a stream of transaction T with utility values and timestamps t and the unit profit of item i having a dynamic value correlated to t, our proposed algorithm aims to find the high utility patterns through the sliding window. Sample dynamic unit profit data stream can be referred to Table 1.1.

Example 1.1.1. In Table 1.1, there are two windows, namely, W_1 and W_2. In the sample, the window size is 2 so each window contains 2 transactions. In the Transaction column, T_1 contains $A(1)$,$B(2)$,$D(5)$. The number inside the bracket represents the number of occur-

Table 1.1: Sample dynamic unit profit data stream

Window ID	TID	Transaction	Unit Profit	Timestamp
W_1	T_1	$A(1)$,$B(2)$,$D(5)$	$\{3,4,5\}$	t_1
	T_2	$A(1)$,$C(1)$,$D(5)$	$\{1,2,5\}$	t_2
W_2	T_3	$A(2)$,$C(1)$,$D(4)$	$\{4,2,5\}$	t_3
	T_4	$A(1)$,$C(1)$,$D(5)$	$\{6,2,5\}$	t_4

rences of a particular item. As a result, there is 1 item A in transaction T_1, 2 items B in transaction T_1, and the same rule can apply to the rest of the transactions. The Unit Profit column represents the unit profit associated with the item in the Transaction column. For example, $\{3,4,5\}$ in transaction T_1 means the unit profit of item A is 3, B is 4 and C is 5. The unit profit is associated with an individual transaction rather than a constant in an external table so it could change during the stream of transactions.

Consequently, in this MSc thesis, I would explore the following two **key research questions**:

Q1. Can we design a high utility stream mining algorithm that is more efficient than existing algorithms?

Q2. Can we design a stream mining algorithm to perform high utility mining efficiently after the external profit data changes?

In terms of **key research contributions**, my key contributions of this work include the following:

1. We propose to enhance the feature of mining high utility patterns over data streams. We design a highly efficient utility list data structure that can mine high utility data in one phase. Our algorithm has the advantage of both high processing performance and

flexibility when dealing with data changes such as price changes / add transactions / delete transactions without the need of re-scanning the data source.

2. Compared to the existing high utility stream mining algorithms, our algorithm is more efficient due to the novel RUtilMap for each utility list to reduce the search space during the mining process. The number of RUtilsMap is the same as the number of distinct items so the memory consumption is not high.

1.2 Thesis Organization

This thesis is organized as follows. The next chapter gives background information and related work. We first describe the preliminary definition of high utility data mining. We then introduce some existing high utility algorithms for mining static databases and compare their differences. We then introduce some existing algorithms for mining data streams. Afterwards, we start to introduce our proposed algorithm called Enhanced-HUI-streaming for mining high utility data stream in Chapter 3.

In Chapter 4, we provide both analytic and experimental evaluations of our proposed algorithm. Finally, we present the conclusion and future work of our thesis in Chapter 5.

Chapter 2

Background and Related Work

In this chapter, we provide background information and related work for this thesis. First, we present the definition of Frequent Itemset Mining (FIM) and High Utility Itemset Mining (HUIM). We explain the differences between FIM and HUIM. Then, we introduce the definitions and lemmas that are related to the thesis. Moreover, we present the Two-Phase [LLC05], UP-growth [TWSY10], HUI [WFGT15], EFIM algorithm [ZFL$^+$17] as the related work since those algorithms represent different strategies of HUIM. We also present FP-Stream [GHRL03], CAN-Tree [LKLH07] for mining static data streams and present THUI-Mine [CTL08], MHUI-BIT, MHUI-TID [LHL11a], HUPMS [ATJC12], SO-HUPDS [JH20] for mining HUI from data streams.

2.1 Frequent Itemset Mining (FIM)

The problem of frequent pattern mining consists of extracting frequent patterns from transactional databases. In a transactional database, each record is called a transaction which holds a list of items. A transactional database denoted as D is defined as follows.

Let there be a set called I that consists of all items in D. If there are m unique items in D, $I = \{i_1, i_2, \ldots, i_m\}$. A transaction database D is a set of transactions, denoted as $D = \{T_1, T_2, \ldots, T_n\}$, where n is the number of transactions. Each transaction T_i is a subset of I with unique identifier i, also known as the transaction ID. The goal of frequent pattern mining is to discover itemsets having the support greater than the user-defined minimum threshold. The support measurement is defined in Definition 1. We provide a sample transactional database in Example 2.1.1.

Example 2.1.1. (Sample Transactional Database) A sample transaction database is specified in Table 2.1. In this sample, there are 5 transactions and each transaction has a different transaction ID (TID) with a list of items. For example, TID 1 contains an itemset with 6 items: $\{c, e, a, b, d, f\}$, TID 2 contains an itemset with 4 items: $\{c, e, b, d\}$. A similar representation can apply to TIDs 3, 4 and 5.

Table 2.1: A sample transaction database

TID	Items
1	c,e,a,b,d,f
2	c,e,b,d
3	c,a,d
4	c,e,a,g
5	c,e,b,g

Definition 1. (Support Measure) The support of an itemset X in a transaction database D is denoted as $sup(X)$, where $sup(X) = |\{T | X \subseteq T \wedge T \in D\}|$, which is the number of transactions containing X in the database D. In Example 2.1.2, we demonstrate the example of the support measure.

Example 2.1.2. (Support Measure Example) Let the user-defined minimum support

threshold be 5. In Table 2.1, the support of the itemset $\{c\}$ is 5 as it appears in TIDs 1, 2, 3, 4, and 5 respectively. Since the support of the itemset $\{c\}$ is no less than the minimum support threshold, the itemset $\{c\}$ is frequent.

2.2 High Utility Itemset Mining (HUIM)

The goal of high utility itemset mining is to discover itemsets that yield high profit in transactional databases. A database consists of transactions and each item in a transaction is associated with a unit utility value (internal utility) and the quantity (external utility). In contrast, the traditional frequent pattern mining does not capture both the utility value and the quantity. The goal of high utility mining is to discover itemsets to have the utility greater than the user-defined minimum utility threshold. In Table 2.2, we demonstrate a sample transactional database with utility values. The related concepts of utility are provided in Definitions 2, 3, 4 and, 5.

Table 2.2: A sample transaction database with utility value

TID	Items	Utilities	Total Utilities
1	c,e,a,b,d,f	1,3,5,10,6,5	30
2	c,e,b,d	3,3,8,6	20
3	c,a,d	1,5,2	8
4	c,e,a,g	6,6,10,5	27
5	c,e,b,g	2,3,4,2	11

Definition 2. (Utility Measure) The utility measure is to evaluate the utility of an item i appearing in a transaction T. The transaction quantity of item i in a transaction T is denoted as $qu(i, T)$. The item profit is denoted as $pu(i)$. The total utility generated by an item i in a transaction T is defined as $u(i, T) = qu(i, T) \times pu(i)$. Furthermore, the utility of

itemset X in a transaction T is defined as $u(X,T)$, where $u(X,T) = \sum_{i \in X} u(i,T)$, $X \subseteq T$.

Definition 3. (Itemset Utility) The itemset utility of a given itemset X is denoted as $U(X,T)$, where $U(X,T) = \sum_{X \subseteq T_q \in D} u(X,T_q)$. The suffix q is an identifier of T.

Definition 4. (High Utility Itemset) An itemset X is called a high utility itemset if its utility is no less than the user-defined minimum threshold ($minUtil$), which is formally defined as $U(X,T) = \sum_{X \subseteq T_q \in D} u(X,T_q) \geq minUtil$.

Definition 5. (High Utility Mining Problem) The goal of high utility mining is to discover all itemset X having utility no less than the given user-defined threshold in a transaction database / data stream D.

Example 2.2.1. (High Utility Itemset Example) Let the $minUtil$ be less than 30. In Table 2.2, the itemset $\{c, e, a, b, d, f\}$ is a high utility itemset as its utility is $(c:1) + (e:3) + (a:5) + (b:10) + (d:6) + (f:5) = 30$.

2.2.1 Key Differences between FIM and HUIM

Traditional association rule mining algorithms (ARM) ignores the weight of an item and the quantity. To address the limitation, weighted association rule mining (WARM) [WYY00, TMF03] was introduced which drives the mining process to focus on high weight support itemsets. WARM is a harder problem than ARM since one of the challenges of high utility mining (HUIM) is the lack of the downward closure property (Refer to Definition 6) proposed by Agrawal and Srikant [AS94]. Downward closure property is heavily used by the FIM algorithms for reducing the search space. However, such property does not hold when dealing with HUIM (Refer to Lemma 1). In the domain of HUIM, the downward closure property only holds when both the unit profit and the quantity are binary values (0/1). Under this

special scenario, HUIM is the same as FIM. In another word, HUIM is a more generic and more difficult problem than FIM.

Definition 6. (Downward Closure Property of FIM) Every subset of a frequent itemset is also frequent.

Example 2.2.2. (Downward Closure Property of FIM Example) Let the user-defined minimum support threshold be 4. In Table 2.1, the itemset $\{c, e\}$ has the support of 4 since it appears in 4 different transactions (Refer to TID 1, 2, 4 and 5). As a result, the itemset $\{c, e\}$ and its subsets, $\{c\}$ and $\{e\}$, are all frequent as their support would be no less than 4. More precisely, $\{c\}$ has the support of 5 and $\{e\}$ has the support of 4. Both the itemset $\{c\}$ and the itemset $\{e\}$ are frequent as long as the itemset $\{c, e\}$ is frequent.

Lemma 1. (No Downward Closure Property of HUIM) Every subset of a high utility itemset is not guaranteed to be high utility itemset.

Example 2.2.3. (No Downward Closure Property of HUIM Sample) Let the user-defined utility threshold be 30. In Table 2.2, the itemset $\{c, e, a, b, d, f\}$ has a total utility of 30 which is a high utility itemset. One of its subsets $\{f\}$ has the utility value of 5 which is not a high utility itemset. Another one of its subsets $\{c, e, a\}$ has the utility value of 31 which is a high utility itemset. Despite the itemset $\{c, e, a, b, d, f\}$ being a high utility itemset, its subsets may not be high utility itemset due to lack of the downward closure property. In other word, knowing an itemset is a high utility itemset does not indicate whether its subsets are high utility itemsets.

2.2.2 Transaction-Weighted Utilization (TWU)

Without the downward closure property, HUIM would be a real difficult problem since the search space can be as large as the number of all possible enumeration of all the subsets of itemsets. Since a length-k itemset has $2^k - 1$ subsets, any HUI algorithm would require an exponential growth of running time concerning the length of the itemset if there is no efficient way to prune itemsets. To improve the mining efficiency, Definition 10 is discovered by Two-Phase [LLC05] which can greatly reduce the search space and shed some light on HUIM related research. We demonstrate the definition of Transaction Utility in Definition 7, which is served as the building block of Transaction-weighted utilization (TWU) in Definition 8. Furthermore, we demonstrate the definition of High TWU itemset in Definition 9 and its downward closure property in Definition 10.

Definition 7. (Transaction Utility) The transaction utility of a given transaction T_q, denoted as $tu(T_q)$, is the sum of utility of all items in a transaction. $tu(T_q) = \sum_{i_p \in T_q} u(i_p, T_q)$.

Example 2.2.4. (Transaction Utility Example) In Table 2.2, $tu(T_1) = \sum_{i_p \in T_1} u(i_p, T_1) = 1 + 3 + 5 + 10 + 6 + 5 = 30$.

Definition 8. (Transaction-weighted Utilization (TWU)) The transaction-weighted utilization of an itemset X is the total transaction utility that contains X in a database D, denoted as $twu(X) = \sum_{X \subseteq T_q \in D} tu(T_q)$.

Example 2.2.5. (Transaction-weighted Utilization (TWU) Example) In Table 2.2, the transaction-weighted utilization of itemset $\{c, e\}$ is $twu(\{c, e\}) = tu(T_1) + tu(T_2) + tu(T_4) + tu(T_5) = 30 + 20 + 27 + 11 = 88$. $tu(T_3)$ does not contribute to the $twu(\{c, e\})$ since $\{c, e\}$ is not a subset of T_3.

Definition 9. (High Transaction-weighted Utilization Itemset) An itemset X is called high transaction-weighted utilization itemset when its transaction utility is no less than the user-defined threshold ϵ, denoted as $twu(X) \geq \epsilon$.

Definition 10. (Downward Closure Property of TWU) Every subset of a high transaction-weighted utilization itemset is also a high transaction-weighted utilization itemset.

Example 2.2.6. (Downward Closure Property of TWU Example) Let the user-defined minimum utility threshold be 80. In Table 2.2, the transaction-weighted utilization of itemset $\{c, e\}$ is 88. Its subsets, $\{c\}$ and $\{e\}$, have transaction-weighted utilization $twu(\{c\}) = tu(T_1) + tu(T_2) + tu(T_3) + tu(T_4) + tu(T_5) = 30 + 20 + 8 + 27 + 11 = 96$ and $twu(\{e\}) = tu(T_1) + tu(T_2) + tu(T_4) + tu(T_5) = 30 + 20 + 27 + 11 = 88$ respectively. Because $twu(\{c, e\}) = 88$ which is greater than the user-defined minimum utility threshold, both $twu(\{c\})$ and $twu(\{e\})$ are no less than 88 and they must be high transaction-weighted utilization itemsets.

With the downward closure property of TWU in Definition 10, the utility monotonicity property can be yielded in Lemma 2.

Lemma 2. (Utility Monotonicity Property) The twu of every subset J of I is at least equal to the twu of itemset I.

$$\forall J \subseteq I,\ twu(J) \geq twu(I) \tag{2.1}$$

Having a monolithic property for HUIM can help with pruning the search space as the transaction-weighted utilization is an upper bound of the actual utility combined with Lemma 3.

Lemma 3. (Transaction-weighted Utilization is the upper bound of itemset utility) An itemset X has transaction-weighted utilization of $twu(X)$ (Refer to Definition 8), and the itemset utility is $U(X,T)$ (Refer to Definition 3), $twu(X) \geq U(X,T)$. If the twu of an itemset X is less than the user-defined minimum utility threshold, the itemset utility of X would be less than the user-defined minimum utility threshold.

Proof of Lemma 3:

$$\begin{aligned} twu(X) &= \sum_{X \subseteq T_q \in D} tu(T_q) = \sum_{X \subseteq T_q \in D} \sum_{i_p \in T_q} u(i_p, T_q) && (2.2) \\ U(X,T) &= \sum_{X \subseteq T_q \in D} u(X,T) = \sum_{X \subseteq T_q \in D} \sum_{i \in X} u(i,T) && (2.3) \\ X &\subseteq T_q && (2.4) \end{aligned}$$

Combining Eqs. (2.2), (2.3) and (2.4), we get

$$\begin{aligned} \sum_{X \subseteq T_q \in D} \sum_{i_p \in T_q} u(i_p, T_q) &\geq \sum_{X \subseteq T_q \in D} \sum_{i \in X} u(i,T) && (2.5) \\ twu(X) &\geq U(X,T) && (2.6) \end{aligned}$$

Combining Lemmas 2 and 3, we derive Lemma 4.

Lemma 4. (Transaction-weighted Utilization Pruning Strategy) When the transaction-weighted utilization of an itemset X is less than the user-defined minimum utility threshold, all its supersets Y would not be high utility itemsets. Formally, $\forall Y \supseteq X$, when $twu(X) < \epsilon$, $U(Y,T) < \epsilon$.

Proof of Lemma 4:

$$twu(X) \geq twu(Y), \forall X \subseteq Y \tag{2.7}$$

$$twu(X) \geq U(X,T) \tag{2.8}$$

$$twu(Y) \geq U(Y,T) \tag{2.9}$$

Combining Eqs. (2.7) and (2.9), we get

$$twu(X) \geq twu(Y) \geq U(Y,T) \tag{2.10}$$

The transaction-weighted utilization is equal to the total itemset utility only when the itemset is the equivalent set of every transaction occurrence as demonstrated in Example 2.2.7. The transaction-weighted utilization is an over-estimation to satisfy the downward closure property in the HUIM domain.

Example 2.2.7. (Transaction-weighted Utilization Special Case Example) In Table 2.3, the total utility of itemset $\{c, a, d\}$ is $U(\{c, a, d\}, T) = 1+5+2+1+5+2+1+5+2 = 24$. The transaction-weighted utilization of itemset $\{c, a, d\}$ is $twu(\{c, a, d\}) = tu(T_1) + tu(T_2) + tu(T_3) = 8 + 8 + 8 = 24$. In this special scenario, the transaction weighted utilization is the same as the itemset utility.

Table 2.3: A sample transaction database with utility values (special case)

TID	Items	Utilities	Total Utilities
1	c,a,d	1,5,2	8
2	c,a,d	1,5,2	8
3	c,a,d	1,5,2	8

TWU is useful in pruning the search space which is a great step forward toward solving the HUIM problem, but it is a loose upper bound and it could not effectively reduce the search space in certain scenarios. In Example 2.2.8, we demonstrate the limitation of TWU.

Example 2.2.8. (The Limitation of TWU Example) Let the user-defined minimum utility threshold be 30. In Table 2.4, $twu(\{c\}) = tu(T_1) + tu(T_2) + tu(T_3) = 19 + 23 + 21 = 63$ which is much more than the minimum utility threshold. However, none of the high utility itemsets includes the itemset $\{c\}$. The itemset utility of $\{c\}$ is $TU(\{c\}) = 3$. The reason $twu(c)$ has a high utility value is that the item c appears in every transaction. The transaction-weighted utilization cannot prune it early on as the upper bound is too loose.

Table 2.4: A sample transaction database with utility value to show the limitation of TWU

TID	Items	Utilities	Total Utilities
1	**c**,a,d	**1**,8,10	19
2	**c**,b,k	**1**,12,10	23
3	**c**,e,g	**1**,8,12	21

2.2.3 Remaining Utility

Quite a few algorithms are leveraging on TWU to reduce the search space. However, to enhance the HUIM algorithm even further and to enhance the runtime performance, it is necessary to find a tighter upper bound other than TWU. Therefore, we demonstrate the definition of remaining utility in Definition 11 as the background information.

Definition 11. (Remaining Utility in a Transaction) Let X be an itemset and be a subset of the transaction T_j. The remaining utility of X in a transaction T_j denoted as $ru(X, T_j)$, is equal to the total utility of all the remaining items appearing after X in transaction T_j. Formally, $ru(X, T_j) = \sum_{x_i \in (T_j/X)} u(x_i, T_j)$, $X \subseteq T_j$.

Example 2.2.9. (Remaining Utility in a Transaction Example) In Table 2.2, the remaining utility of the itemset $\{c, e\}$ in the transaction T_1 is equal to $ru(\{c, e\}, T_1) = u(a, T_1) + u(b, T_1) + u(d, T_1) + u(f, T_1) = 5 + 10 + 6 + 5 = 26$.

Definition 12. (Total Remaining Utility) Let X be an itemset. The total remaining utility, denoted as $RU(X)$, is equal to the total utility of all the remaining items appearing after X in the transactions where X is the subset. $RU(X) = \sum_{X \subset T_j \in D} ru(X, T_j)$, and j is the transaction id.

Example 2.2.10. (Total Remaining Utility Example) In Table 2.2, the total remaining utility of the itemset $\{c, e\}$ in the sample transaction database is $RU(\{c, e\}) = ru(\{c, e\}, T_1) + ru(\{c, e\}, T_2) + ru(\{c, e\}, T_3) + ru(\{c, e\}, T_4) + ru(\{c, e\}, T_5) = 26 + 14 + 0 + 15 + 6 = 61$.

2.2.4 Tighter Overestimated Utility Upper Bound (TOU)

A tighter than TWU upper bound can be derived by summing up the remaining utility and the utility, which is defined in Lemma 5.

Lemma 5. (Tighter overestimated Utility Upper Bound (TOU)) A Tighter overestimated upper bound can be derived by the summation of utility and the remaining utility, denoted as $TOU(X) = U(X, T) + RU(X)$, and $TOU(X) = U(X, T) + RU(X) \geq U(X, T)$

Proof of Lemma 5

$$U(X, T) = \sum_{X \subseteq T_q \in D} u(X, T) \tag{2.11}$$

$$RU(X) = \sum_{X \subset T_j \in D} ru(X, T_j) \tag{2.12}$$

$$TOU(X) = U(X, T) + RU(X) \geq U(X, T) \tag{2.13}$$

Example 2.2.11. (TOU Example) In Table 2.2, the TOU of the itemset $\{c, e\}$ is $TOU(\{c, e\}) = U(\{c, e\}) + RU(\{c, e\}) = (1 + 3) + (5 + 10 + 6 + 5) + (3 + 3) + (8 + 6) + (6 + 6) + (10 + 5) + (2 + 3) + (4 + 2) = 88$. In this case, $TOU(\{c, e\}) = twu(\{c, e\})$ because item c and e are next to each other. If we choose another itemset such as c, b, $TOU(\{c, b\}) = U(\{c, b\}) + RU(\{c, b\}) = (1 + 10) + (6 + 5) + (3 + 8) + 6 + (2 + 4) + 2 = 47$ which is smaller than $twu(\{c, b\}) = 30 + 20 + 11 = 61$.

The item in each transaction must be sorted by a consistent order as the prerequisite condition before TOU can be applied in pruning the search space. We can see TOU does not hold when the condition is not met in Example 2.2.12.

Example 2.2.12. (TOU Counter Example) In Table 2.5, the remaining utility of the itemset $\{c, e\}$ in the transaction T_1 includes the itemset $\{b, d\}$, but the remaining utility of the itemset $\{c, e\}$ does not include the itemset $\{b, d\}$. This violates Definition 12 because the remaining itemsets after X are not consistent across all transactions.

Table 2.5: A sample transaction database with utility value where TOU cannot be applied directly

TID	Items	Utilities	Total Utilities
1	**c,e**,a,b,d,f	**1,3**,5,10,6,5	30
2	b,d,**c,e**	8,6,**3,3**	20

Different ordering could affect the effectiveness of pruning itemsets. Ahmed *et al.*[ATJL09a] indicates that TWU in descending order would facilitate the mining performance. The remaining utility is the indicator to tell whether an itemset is worth extending. If TOU is less than the user-defined minimum utility threshold, the itemset including all its extensions would not be high utility itemsets. We can visualize Table 2.1 in a tree view by Figure 2.1. The remaining utility is the total utility of all the children for a specific node.

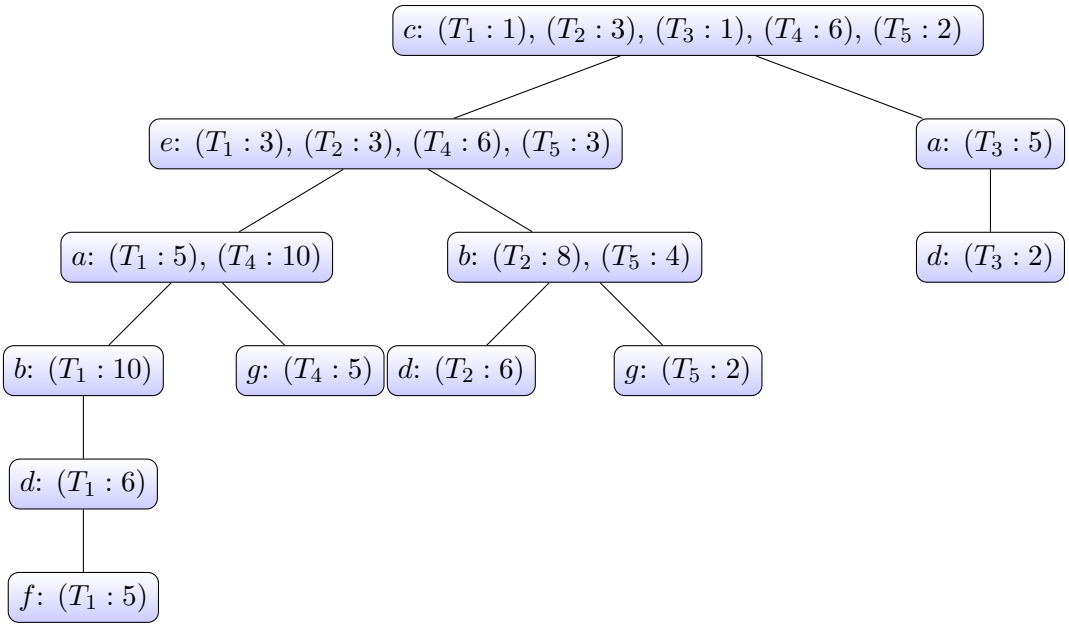

Figure 2.1: Tree representation of Table 2.2

Example 2.2.13. (The Impact of Ordering to TOU Example) In Table 2.2, the total remaining utility of the itemset $\{c, e\}$ is $RU(\{c, e\}) = a[(T_1 : 5), (T_4 : 10)] + b[(T_2 : 8), (T_5 : 4)] + b[(T_1 : 10)] + d[(T_1 : 6)] + f[(T_1 : 5)] + g[(T_4 : 5)] + d[(T_2 : 6)] + g[(T_5 : 2)] = 5 + 10 + 8 + 4 + 10 + 6 + 5 + 5 + 6 + 2 = 61$. The result is identical to Example 2.2.10. The remaining utility is the same as the subtree utility, and by altering the ordering of the itemset from Table2.2 to Table 2.6, the tree representation would be changed from Figure 2.1 to Figure 2.2. The number of levels to be explored would change based on the ordering.

There is a limitation on the sorting by TWU approach when dealing with data streams since the TWU changes dynamically. It is more practical to sort itemsets by the canonical order when dealing with data streams as the order would not change regardless of the size of the data.

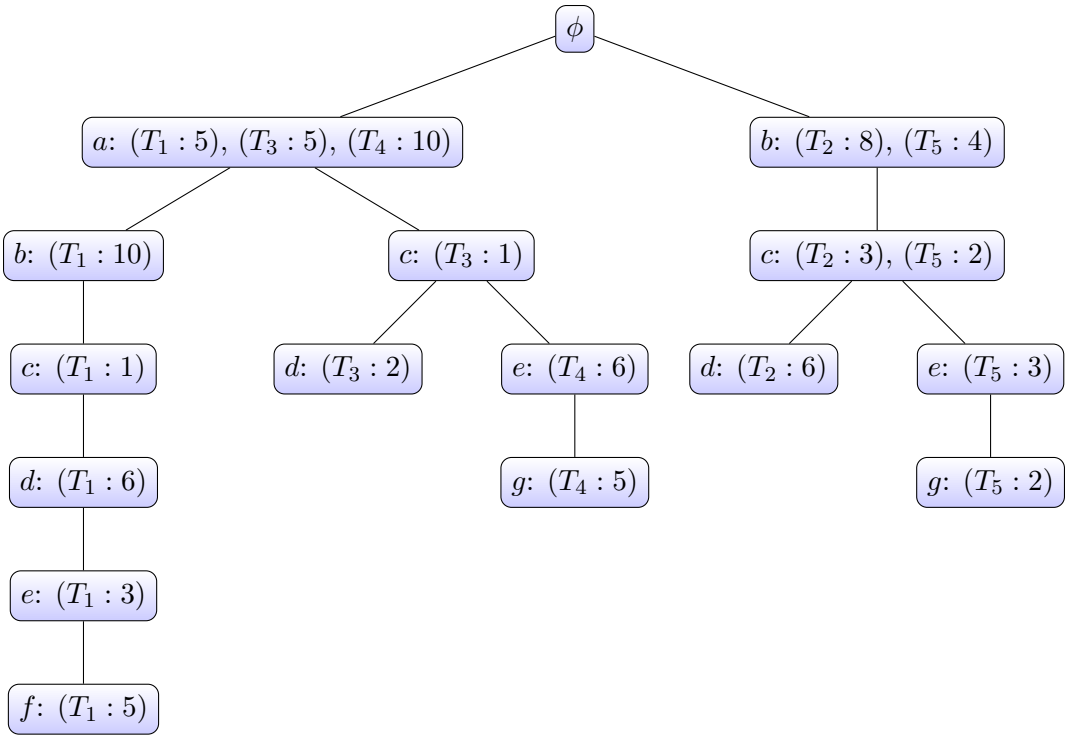

Figure 2.2: Tree representation of Table 2.6

Table 2.6: A sample transaction database with utility value sorted by the canonical order

TID	Items	Utilities	Total Utilities
1	a,b,c,d,e,f	5,10,1,6,3,5	30
2	b,c,d,e	8,3,6,3	20
3	a,c,d	5,1,2	8
4	a,c,e,g	10,6,6,5	27
5	b,c,e,g	4,2,3,2	11

2.3 High Utility Mining for Static Database

2.3.1 Two-Phase Algorithm

Two-Phase [LLC05] introduced an upper bound called Weighted Transaction Utility (TWU) which reduces the search space. In Definition 8, any item with TWU that is lower than minUtils can be pruned away. The algorithm separates the mining process into two phases: 1) generate candidates during the first database scan 2) verify the candidates during subsequent database scans. The two phases cycle repeats to generate longer patterns. Two-Phase is similar to Apriori in the HUIM domain. Comparing Two-Phase and Apriori, Two-Phase prunes itemsets based on Definition 10, and Apriori prunes itemsets based on Definition 6. Because of Lemma 4, Two-Phase suffers from a similar but even worse issue than Apriori because TWU is a loose upper bound so a large number of candidates cannot be determined to be not qualified and pruned away early by TWU. Furthermore, it suffers from having a high I/O cost caused by multiple database scans to verify whether the candidates are high utility itemsets. Please refer to Example 2.3.1.

Example 2.3.1. (Two-Phase Algorithm Example) Let us consider the transaction database shown in Table 2.8. Based on the utility measurement in Definition 2, the transaction database is simplified as shown in Table 2.9. Let the user-defined minimum utility threshold (minUtils) be 10. The Two-Phase algorithm first generates C_1, then calculates the TWU value of each item in C_1 and eliminate items with TWU value less than minUtils. The TWU value of the itemset $\{e\}$ is 8 which is not a high utility itemset. As a result, all its extension super-sets would not be high utility itemsets and they can be pruned away. Hence there are 4 patterns left in the L_1 table. Despite both the itemset $\{a\}$ and the itemset $\{d\}$ in the L_1 table being less than minUtils but their TWU are no less than minUtils, they

Table 2.7: A sample run of the Two-Phase algorithm (Example 2.3.1)

(a) C_1 Table

Pattern	TWU Value	Utility Value
$\{a\}$	29	4
$\{b\}$	30	10
$\{c\}$	30	12
$\{d\}$	30	9
$\{e\}$	8	3

(b) L_1 Table

Pattern	TWU Value	Utility Value
$\{a\}$	26	4
$\{b\}$	30	10
$\{c\}$	27	12
$\{d\}$	30	9

(c) C_2 Table

Pattern	TWU Value	Utility Value
$\{a, b\}$	21	11
$\{a, c\}$	18	11
$\{a, d\}$	21	9
$\{b, c\}$	22	14
$\{b, d\}$	30	19
$\{c, d\}$	22	14

(d) L_2 Table

Pattern	TWU Value	Utility Value
$\{a, b\}$	21	11
$\{a, c\}$	18	11
$\{a, d\}$	21	9
$\{b, c\}$	22	14
$\{b, d\}$	30	19
$\{c, d\}$	22	14

(e) C_3 Table

Pattern	TWU Value	Utility Value
$\{a, b, c\}$	13	10
$\{a, b, d\}$	21	17
$\{a, c, d\}$	13	9
$\{b, c, d\}$	22	20

(f) L_3 Table

Pattern	TWU Value	Utility Value
$\{a, b, c\}$	13	10
$\{a, b, d\}$	21	17
$\{a, c, d\}$	13	9
$\{b, c, d\}$	22	20

(g) C_4 Table

Pattern	TWU Value	Utility Value
$\{a, b, c, d\}$	13	13

(h) L_4 Table

Pattern	TWU Value	Utility Value
$\{a, b, c, d\}$	13	13

are considered as candidates to generate the C_2 table. Observing from the following tables, there is no pattern less than minUtils in C_2, C_3 and C_4 tables. This process repeats until there are no more potential high utility candidates.

Note that Two-Phase algorithm uses a level-wise candidate generation process which is much similar to the Apriori algorithm in the frequent pattern mining domain. The TWU value is a loose upper so it is usually much larger than the actual utility value. As a result, Two-Phase algorithm has limited ability in pruning itemsets.

Table 2.8: Transaction database example

TID	Items	Quantity	Unit Price	Total Utilities
T_1	a,b,d	1,2,1	a:1, b:2, d:3	8
T_2	b,c,d	1,1,1	b:2,c:4,d:3	9
T_3	a,b,c,d	2,2,1,1	a:1,b:2,c:4,d:3	13
T_4	a,c,e	1,1,1	a:1,c:4,e:3	8

Table 2.9: Transaction database example (simplified)

TID	Items	Utilities	Total Utilities
T_1	a,b,d	1,4,3	8
T_2	b,c,d	2,4,3	9
T_3	a,b,c,d	2,4,4,3	13
T_4	a,c,e	1,4,3	8

2.3.2 UP-Growth Algorithm

Inspired by FP-growth [HPYM04] and to reduce the I/O cost caused by the candidate generation process, UP-growth [TWSY10] algorithm is proposed. It is more efficient than Two-Phase because 1) there is no candidate generation and 2) the mining process relies on mining the tree structure in memory rather than re-scanning the database. The pruning strategy of FP-growth is based on Definition 10 which is the same strategy as Two-Phase, but a definition called RTU (reorganized transaction utility) is introduced. RTU is the transaction utility (Recall from Definition 7) after unpromising items are pruned by Definition 10. Because of the loose upper bound nature of TWU, FP-growth still suffers from having a large number of candidates to be processed in memory.FP-growth has a better pruning strategy than TWU, but it is still suffered from the overhead of candidate generation and additional database scans.

Example 2.3.2. (UP-growth Algorithm Example) The UP-growth algorithm consists of two scans of a database. For the first scan, the transaction utility of each transaction and the TWU of each item is computed. As shown in Table 2.12, if the minimum utility threshold is 40, the itemset $\{f\}$ and the itemset $\{g\}$ are not frequent thus would be removed from the transactions. Table 2.13 is the re-organized database after the first database scan. The items in each transaction are sorted by the TWU value in ascending order and the total utilities are re-computed after the non-frequent itemsets are pruned. UP-Tree shown in Figure 2.3 is created by inserting all re-organized transactions in Table 2.13. Each node in the UP-tree contains the pattern name, the support count, and the associated TWU value.

FP-growth algorithm uses the FP-tree for mining purposes. The algorithm starts from the last item in the header table which is $\{d\}$. There are three paths:

- $(\{d\} \rightarrow \{b\} \rightarrow \{a\} \rightarrow \{e\} \rightarrow \{c\}, 1, 25)$
- $(\{d\} \rightarrow \{b\} \rightarrow \{e\} \rightarrow \{c\}, 1, 20)$
- $(\{d\} \rightarrow \{a\} \rightarrow \{c\}, 1, 8)$

Under $\{d\}$'s conditional pattern, the path utility of the itemset $\{a\}$ is 33, the path utility of the itemset $\{b\}$ is 45, the path utility of the itemset $\{c\}$ is 53, and the path utility of the itemset $\{e\}$ is 45. Pattern $\{a\}$ is a local unpromising item since its utility is less than the user-defined minimum utility threshold. The remaining items are re-arranged based on their local utility in ascending order and in canonical order if the local utility value is identical. Those three paths are revised as below:

- $(\{d\} \rightarrow \{c\} \rightarrow \{b\} \rightarrow \{e\}, 1, 25)$
- $(\{d\} \rightarrow \{c\} \rightarrow \{b\} \rightarrow \{e\}, 1, 20)$

- $(\{d\} \rightarrow \{c\}, 1, 8)$

The re-organized paths are for creating the $\{d\}$'s conditional tree as shown in Figure 2.4. By applying UP-growth, a list of potential high utility itemsets are produced: $(\{d\} : 53)$, $(\{d, e\} : 45)$, $(\{d, e, b\} : 45)$, $(\{d, e, c\} : 45)$, $(\{d, e, b, c\} : 45)$, $(\{d, b\} : 45)$, $(\{d, b, c\} : 45)$, $(\{d, c\} : 53)$. By tracing the header table, all the potential high utility itemsets are found, and another database scan is required to calculate the specific utility.

Table 2.10: Local utility pattern $\{d\}$'s condition

Item	a	b	c	e
Local Utility	33	45	53	45

Table 2.11: Transaction database example

TID	Items	Quantity	Unit Price	Total Utilities
T_1	a,c,d	1,1,1	a:5, c:1, d:2	8
T_2	a,c,e,g	2,6,2,5	a:5, c:1, e:3, g:1	27
T_3	a,b,c,d,e,f	1,2,1,6,1,5	a:5, b:2, c:1, d:2, e:3, f:1	30
T_4	b,c,d,e	4,3,3,1	b:2, c:1, d:2, e:3	20
T_5	b,c,e,g	2,2,1,2	b:2, c:1, e:3, g:1	11

Table 2.12: Transaction TWU example

Item	a	b	c	d	e	f	g
TWU	65	61	96	58	88	30	38

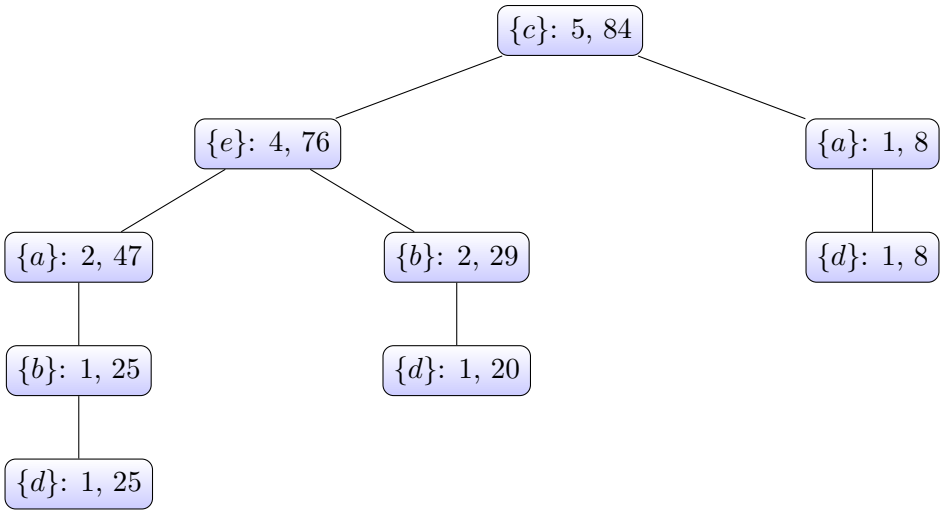

Figure 2.3: UP-Tree

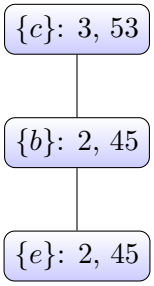

Figure 2.4: $\{d\}$'s conditional UP-Tree

Table 2.13: Reorganized transaction database example

TID	Items	Quantity	Unit Price	Total Utilities
T_1	c,a,d	1,1,1	c:1, a:5, d:2	8
T_2	c,e,a	6,2,2	c:1,e:3,a:5	22
T_3	c,e,a,b,d	1,1,1,2,6	c:1,e:3,a:5, b:2, d:2,	25
T_4	c,e,b,d	3,1,4,3	c:1, e:3, b:2, d:2,	20
T_5	c,e,b	2,1,2	c:1, e:3, b:2	9

Besides UP-growth, several other HUIM algorithms based on tree-structures have been proposed such as CUP-Tree [EGA07] and HUP-Tree [LHL11b]. However, one of the limitations of these tree algorithms is the generation of a high number of conditional trees for

pattern growth since the process has a high space cost.

Other than Apriori-based algorithms and tree-based algorithms, there are list-based HUIM algorithms. HUI-Miner [WFGT15] proposed a high utility mining approach without the candidate generation using the utility list structure.

2.3.3 Utility-List Algorithm: HUI-Miner

Both Two-Phase and FP-growth and their variant algorithms are all required candidate generation and additional database scans. The next enhancement of HUIM related research is whether it is possible to have HUIM algorithms without the candidate generation process. A data structure called utility-list is proposed and some new pruning strategy based on the heuristic information. We describe the Utility-List based algorithm in Example 2.3.3.

Example 2.3.3. (Utility-List Algorithm Example) Let the user-defined minimum utility threshold be 30. In Table 2.8, the Utility-List algorithm scans the transaction database for the first time to calculate the TWU of each value as shown in Table 2.12. From the TWU table, it creates a list of utility lists of items by the following condition $twu(item) \geq minUtility$. Since the user-defined minimum utility threshold is 30, no items are pruned by the TWU. The transactions as shown in Table 2.20 are revised sorted by TWU in descending order.

The utility list is consist of Tid, Iutils and Rutils. With the list of utility lists, the mining process can be performed recursively. For each utility list as a prefix, it explores the possible extensions by concatenating the utility list in the same level. A utility list that is worth exploring only when the sum of utility and the remaining utility is no less than the user-defined minimum utility threshold.

Table 2.14: List of Utility Lists in Level 1

Item: f		
Tid	Iutil	Rutil
1	5	25

Item: g		
Tid	Iutil	Rutil
4	5	22
5	2	9

Item: d		
Tid	Iutil	Rutil
1	6	19
2	6	14
3	2	6

Item: b		
Tid	Iutil	Rutil
1	10	9
2	8	6
5	4	5

Item: a		
Tid	Iutil	Rutil
1	5	4
3	5	1
4	10	12

Item: e		
Tid	Iutil	Rutil
1	3	1
2	3	3
4	6	6
5	3	2

Item: c		
Tid	Iutil	Rutil
1	1	0
2	3	0
3	1	0
4	6	0
5	2	0

Table 2.15: List of Utility Lists in Level 2 with prefix f

Item: f,d		
Tid	Iutil	Rutil
1	11	19

Item: f,b		
Tid	Iutil	Rutil
1	15	9

Item: f,a		
Tid	Iutil	Rutil
1	10	4

Item: f,e		
Tid	Iutil	Rutil
1	8	1

Item: f,c		
Tid	Iutil	Rutil
1	6	0

Table 2.16: List of Utility Lists in Level 3 with prefix f,d

Item: f,d,b		
Tid	Iutil	Rutil
1	21	9

Item: f,d,a		
Tid	Iutil	Rutil
1	16	4

Item: f,d,e		
Tid	Iutil	Rutil
1	14	1

Item: f,d,c		
Tid	Iutil	Rutil
1	12	0

Table 2.17: List of Utility Lists in Level 4 with prefix f,d,b

Item: f,d,b,a		
Tid	Iutil	Rutil
1	26	4

Item: f,d,b,e		
Tid	Iutil	Rutil
1	24	1

Item: f,d,b,c		
Tid	Iutil	Rutil
1	22	0

Table 2.18: List of Utility Lists in Level 5 with prefix f,d,b,a

Item: f,d,b,a,e		
Tid	Iutil	Rutil
1	29	1

Item: f,d,b,a,c		
Tid	Iutil	Rutil
1	27	0

Table 2.19: List of Utility Lists in Level 6 with prefix f,d,b,a,e

Item: f,d,b,a,c,e		
Tid	Iutil	Rutil
1	30	0

Table 2.20: A sample revised transaction database with utility value

TID	Items	Utilities	Total Utilities
1	f,d,b,a,e,c	5,6,10,5,3,1	30
2	d,b,e,c	6,8,3,3	20
3	d,a,c	2,5,1	8
4	g,a,e,c	5,10,6,6	27
5	g,b,e,c	2,4,3,2	11

Table 2.21: TWU

Item	f	g	d	b	a	e	c
TWU	30	38	58	61	65	88	96

EFIM [ZFL+17] which is known as the most efficient high utility mining algorithm integrates both TWU and the subtree utility as the upper bound, and it uses database projection and transaction merging to further reduce the search space. Most recently, Nguyen *et al.* [NNV+21] proposed an algorithm called iEFIM-Closed which is based on EFIM and to mine closed itemsets only to reduce the candidates, and it separates unit profit from quantity to handle dynamic profit databases. However, both EFIM and iEFIM-Closed is a

static algorithm which cannot re-mine the databases efficiently when the profit is changed. During the mining, they would aggregate the information which cannot assist the continuous mining task when that happens. Refer to Section A in the Appendix, we can see all the array manipulation tricks used by EFIM to speed up the runtime, but it would lose the ability to continuous update the information after the process. We can see in the following related work that all incremental mining and stream mining algorithms are either list-based or tree-based, but no EFIM related algorithm.

All algorithms above are feature equivalent which means they attempt to solve the same problem in different ways. They made a simplified assumption that the item external unit utility is a constant.

More recent HUIM research shifts the focus from purely increasing efficiency to finding real-world problems to solve. Taking the timestamp into consideration, Fournier-Viger *et al.* [FZL+19] proposed local high utility itemsets (LHUI) and peak high utility itemsets (PHUI) where the former captures the high utility itemset within a user-specific time interval and the latter captures the time when the itemset utility becomes unusually high.

Nguyen *et al.* [NNN+19] proposed an algorithm called MEFIM which is a modified version of EFIM which considers the unit profit is dynamic. However, this algorithm is only suitable for mining static databases and it is the state of the art so far. All algorithms described in this section mine static databases. To the best of our knowledge, there is no algorithm for mining dynamic data streams under the scenario of the unit profit being dynamic.

2.4 Incremental High Utility Mining: EIHI

Traditional HUIM algorithms have drawbacks such as having to rescan datasets when a transaction is added, deleted or changed in a database. The lack of the ability to mine datasets incrementally is highly inefficient to handle dynamic datasets. To mitigate the limitation, the concept of incremental mining techniques are introduced to mine high utility itemsets in a dynamic environment where a database grows frequently. IHUP [ATJL09b] is the first algorithm to handle the incremental high utility mining problem with the end goal of achieving the build once and mine many property. IHUP algorithm is based on the combination of Two-Phase and FP-growth. It maintains the transaction in a tree data structure called IHUP_{twu}-Tree. Since it uses TWU as the pruning technique, its tree node contains TWU and the transaction frequency (TF). The tree node is ordered by the lexicographic order so the tree structure is consistent regardless when a new transaction is added, modified or deleted. As it is the first high utility incremental mining algorithm. IHUP chooses existing algorithm such as Two-Phase as the baseline and shows better performance. FUP-HUI-INS [HLW09] introduced the average utility upper bound to overestimate the actual utility to reduce the execution time, but its core functionality is similar to the Two-Phase algorithm and it generates a large amount of candidates. To overcome the limitation, HUI-LIST-INS [LGHP14] inherent the HUI algorithm and build the utility list structures for mining HUI incrementally. It proposed the merge list algorithm which is to check the whether newly added transactions are part of the original database and trigger the incremental mining process when the condition is met. Inspired by HUI-List-INS algorithm, the EIHI algorithm introduced HUI-trie data structure to increase the searching efficiency, and it also introduced a property to determine whether an itemset is a high utility itemset. If the newly added itemset does not appear

in the original database, the original high utility itemsets would still be high utility itemsets, and the original low utility itemsets would still be low utility itemsets. The experimental shows that EIHI is considerably faster than HUI-List-INS algorithm. The previous approach requires at least two scans for processing dynamic database due to the reason of constructing TWU for pruning purpose. Ryang and Yun proposed a list-based data structure that can build the global data structure during the initial scan and incrementally update the data structure without further scanning the dataset. However, its baselines are IHUP algorithms and its variant. To the best of our knowledge, EIHI is the fastest incremental high utility mining algorithm up to date. We show the example of EIHI algorithm in Example 2.4.1

Example 2.4.1. (EIHI Algorithm Example) Let us consider the dataset in Table 2.20 and let us assume the minimum utility be 30. The main procedure is to calculate the TWU of each items and prune away items having TWU less than minimum utility. To explore the extensions of a given itemset, it uses the sum of utility and the remaining utility. It is the same algorithm as HUI-miner up to this stage as Example 2.3.3.

When the database is updated (e.g., as shown in Table 2.22, where TID 6 is being added), EIHI possesses the following property: if the newly added item is not a part of the original high utility itemsets, there is no need to re-mine the whole dataset. For instance, if h is a new item that is not a part of the original high utility itemsets, then all original high utility itemsets remain as high utility and all original low utility itemsets remain as low utility. The algorithm only needs to check if the newly added item can generate any high utility itemsets. In the newly added TID 6, if h is not part of the existing high utility itemsets but f is a part of the existing high utility itemsets. The utility of $\{f, d, b, a, e, c\}$ would be updated from 30 to 31 while h does not affect the original high utility itemsets.

Table 2.22: A sample revised transaction database with utility value to demonstrate incremental mining (Example 2.4.1)

TID	Items	Utilities	Total Utilities
1	f,d,b,a,e,c	5,6,10,5,3,1	30
2	d,b,e,c	6,8,3,3	20
3	d,a,c	2,5,1	8
4	g,a,e,c	5,10,6,6	27
5	g,b,e,c	2,4,3,2	11
6	f,h	1,2	3

2.5 High Utility Mining over Data Streams

Incremental High Utility Mining minimizes the number of database scan compared to traditional high utility mining when the transaction is added, deleted or changed. The related research setup the foundation of handling dynamic databases. Similar to dynamic databases, data streams have information continuous being added, deleted and changed, but it is more time sensitive than dynamic databases as data streams are continuous and unbounded so old information may not be interesting in the current time window. If storing all the information from data streams, it would require endless storage which is not feasible and the stale data with less meaning interferes with the recent and current data to corrupt the report. In other words, mining from data streams is a generic case of mining from dynamic databases when the old information would be discarded consistently.

2.5.1 Frequent Itemset Data Stream Mining Algorithms

Previous work studied the problem of maintaining all frequent itemsets over the history. Giannella *et al.* [GHRL03] introduced a data structure called FP-stream implemented using

FP-tree to maintain the frequency histories. It updates the data structure on each arrival of the data stream batch. It also uses exponential time granularity along with an ageing function to reduce the weight of older records. This is more practical compared to the previous study which maintains the frequent patterns over the entire data stream.

Leung *et al.* [LKLH07] proposed a tree data structure called CanTree to capture the itemsets according to the canonical order which is pre-defined by the user before the mining process. There is no additional database scan required when using CanTree and this data structure is suitable for the constraint mining and interactive mining. CanTree requires upward projection only which greatly reduces the computation cost. Based on the canonical order, the order of items is independent of the itemset frequency which removes the cost of the tree node swapping and merging. However, CanTree is designed for mining static data but not for the dynamic stream. Therefore, some extensions are required to adopt CanTree into the domain of mining data streams.

Inspired by CanTree, Leung and Khan [LK06] proposed a new data structure called Data Stream Tree (DSTree) which maintains and mines frequent patterns effectively. DSTree is used for exact data stream mining with the fixed size sliding window model. DSTree is similar to CanTree with the following difference: CanTree is designed for incremental mining while the DSTree is designed for stream mining. Instead of having only one frequency count per tree node, the DSTree has a list of frequency count to capture the content of several batches rather than only the current batch. As a result, DSTree is suitable to deal with dynamic data. Like CanTree, DSTree also arranges transactions in canonical order to avoid frequency related swapping, splitting, and merging. Using DSTree, the mining process can be delayed till needed to reduce the computation overhead. However, DSTree assumes the main memory is not the main concern and made an assumption that the DSTree can fit

into the main memory. In summary, DSTree has lower maintenance cost when compared to CanTree since decreasing frequency count does not require expensive tree node deletion but only to shift the dependency list. The techniques in this section are association rule mining without utility value in a dynamic stream. In contrast, our proposal examines high utility mining with dynamic unit profit in a data stream, which is more complex than the typical association rule mining.

2.5.2 High Utility Data Stream Mining Algorithm: THUI-Mine

High Utility pattern mining over data stream has become a more challenging research problem than mining over the static database. The first high utility data stream algorithm is THUI-Mine [CTL08]. THUI-Mine algorithm proposed a sliding window-based approach to mine temporal high utility patterns. THUI-Mine is based on Two-Phase algorithm with the extension of sliding window filtering. It separates a database into partitions. The previous partition carries the candidate information to the next partition. The key idea of THUI-Mine algorithm is to maintain a TWU-table during the mining process. We show the THUI-Mine algorithm in Example 2.5.1 for clarity purposes. Compared to Two-Phase algorithm, THUI mine algorithm discard old record and be able to mine recent information from data streams in a resource-limited environment.

Example 2.5.1. (THUI-Mine Algorithm Example)

Let the minimum utility threshold be 120, THUI-Mine algorithm breaks transaction into partitions. In this example, the size of the partition is 3 transactions. THUI-Mine uses the sliding window approach. Let the size of the window be 3 partitions. The filtering threshold of each partition is $120/3 = 40$. Each partition would be calculated separately and only TWU2I would be kept. When the mining process is requested, only 1 more database scan

Table 2.23: A sample transaction database with utility value to demonstrate THUI-Mine algorithm

Partition	TID	Items	Utilities	Total Utilities
1	1	c,e	26,5	31
	2	b,d,e	60,6,5	71
	3	a,d	36,6	42
2	4	b,d	10,42	52
	5	c,e	12,10	22
	6	a,b,e	3,40,5	48
3	7	b,e	100,5	105
	8	a,c,d,e	3,1,18,5	27
	9	a,b,c	3,10,27	40
4	10	b,c	60,2	62
	11	b,d	30,12	42
	12	b,c	20,1	21

is required to check whether the candidate itemsets are high utility itemsets. In Table 2.24 for example, the potential candidates are $\{b, c\}, \{b, d\}, \{b, e\}, \{b, c, d\}, \{b, c, e\}, \{b, d, e\}, \{b, c, d, e\}$. A second database scan is required to check whether these 7 candidates are high utility itemsets. This stream mining algorithm has step size which is equal to the size of the partition and the window size which is equal to a given number of partitions. It proposes one partition at a time and not qualified patterns (when the twu is less than the minimum utility / the number of partitions in a window) would be dropped before processing the next partition. The mining process would be similar to Two-Phase algorithm as demonstrated in Example 2.3.1.

Table 2.24: Temporal high 2-items utility itemsets generated by THUI-mine

P1

C2	Start	TWU
a,d	1	42
b,d	1	71
b,e	1	71
d,e	1	71

P2

C2	Start	TWU
♦ b,d	1	$71 + 52 = 123$
a,b	2	48
a,e	2	48
♦ b,e	1	$71 + 48 = 119$
♦ d,e	1	71
♦ a,d	1	42

P3

C2	Start	TWU
♦ b,e	1	$119 + 105 = 224$
a,c	3	67
♦ a,b	2	$48 + 40 = 88$
b,c	3	40
♦ a,e	2	$48 + 27 = 75$
♦ b,d	1	$71 + 52 = 123$

P3 − P1

C2	Start	TWU
b,e	2	$224 - 71 = 153$
a,c	3	67
a,b	2	88
b,c	3	40
a,e	2	75
b,d	2	$123 - 71 = 52$ (pruned)

P4

C2	Start	TWU
♦ b,c	3	$40 + 83 = 123$
b,d	4	42
♦ b,e	2	153

2.5.3 High Utility Data Stream Mining Algorithms: MHUI-BIT and MHUI-TID

There are some limitation of THUI-Mine algorithm as it generates a large amount of false candidates and high memory requirement. Li *et al.* [LHL11a] proposed MHUI-BIT and MHUI-TID algorithms for mining high utility itemsets over data streams and both algorithms outperform THUI-Mine. Both algorithms use a level-wise approach to first generate a set of candidates itemsets and store the information in a tree, another database scan is required to determine high utility patterns among the candidate itemsets. In Example 2.5.2, we demonstrate the MHUI-BIT and MHUI-TID algorithms (they are essentially the same algorithm with different data structure interpretation) so that we can see clearly its similarities and differences compared to THUI-Mine.

Example 2.5.2. (MHUI-BIT and MHUI-TID Algorithm Example)

Let MHUI-BIT/TID algorithm perform on the database in Table 2.25. Let the minimum utility threshold be 120 and the windows size is 9. MHUI-BIT/TID algorithm constructs

Table 2.25: A sample transaction database with utility value to demonstrate MHUI-BIT and MHUI-TID algorithms

TID	Items	Utilities	Total Utilities
1	c,e	26,5	31
2	b,d	60,6	66
3	a,d	36,6	42
4	b,d	10,42	52
5	c,e	12,10	22
6	a,b,e	3,40,5	48
7	b,e	100,5	105
8	a,b,c,d,e	3,10,1,18,5	37
9	a,b,c,e	6,10,27,10	53
10	b,c	60,2	62
11	b,d	30,12	42
12	b,c	20,1	21

Table 2.26: BitVectors and TIDLists generated by MHUI-BIT and MHUI-TID

BitVectors

Item	TransW1	TransW2	Type
a	$\langle 001001011 \rangle$	$\langle 010010110 \rangle$	No change
b	$\langle 010101111 \rangle$	$\langle 101011111 \rangle$	InsertItem
c	$\langle 100010011 \rangle$	$\langle 000100111 \rangle$	IntersecItem
d	$\langle 011100010 \rangle$	$\langle 111000100 \rangle$	No change
e	$\langle 100011111 \rangle$	$\langle 000111110 \rangle$	DeleteItem

TIDLists

Item	TransW1	TransW2	Type
a	$\langle 3, 6, 8, 9 \rangle$	$\langle 2, 5, 7, 8 \rangle$	No change
b	$\langle 2, 4, 6, 7, 8, 9 \rangle$	$\langle 1, 3, 5, 6, 7, 8, 9 \rangle$	InsertItem
c	$\langle 1, 5, 8, 9 \rangle$	$\langle 4, 7, 8, 9 \rangle$	IntersecItem
d	$\langle 2, 3, 4, 8 \rangle$	$\langle 1, 2, 3, 7 \rangle$	No change
e	$\langle 1, 5, 6, 7, 8, 9 \rangle$	$\langle 4, 5, 6, 7, 8 \rangle$	DeleteItem

either the bit vectors table or the tid lists in Table 2.26. For example, 5 items a, b, c, d, e, are high TWU itemsets. The candidates 2-itemsets are generated by bitwise AND operation. MHUI-BIT/TID algorithm keeps the result in a tree data structure called LexTree-2HTU. Updating the LexTree-2HTU is the key component of MHUI-BIT/TID algorithm. When the item is an InsertItem, which means the itemset may not be 2HTU in the previous window but could be 2HTU in the current window. Comparing TransW1 to TransW2, only item b is InsertItem and its potential 2HTU with prefix b is b,c. The LexTree-2HTU would include b,c after verifying its transaction weight utilization is greater than the minimum

utility threshold. When the item is a DeleteItem, which means the itemset may be 2HTU in the previous window but may be not 2HTU in the current windows. Comparing TransW1 to TransW2, item e is belong to the DeleteItem type. The algorithm checks the child node of item e and update the LexTree-2HTU accordingly. Finally, when the item is an IntersectItem, the original 2HTU may be not high utility and vice versa. Comparing TransW1 to TransW2, item c is an IntersectItem and only an existing 2HTU, c,e, need to be checked and see if it is still a qualified 2HTU. The mining process is the same as Two-Phase algorithm based on the updated TWU table.

2.5.4 High Utility Data Stream Mining Algorithm: HUPMS

THUI-Mine, MHUI-BIT, MHUI-TID are all based on Two-Phase algorithms in essence so they all have the performance degradation issue due to the candidate generation process and multiple database scans issues. To address the limitation of the Apriori-based algorithm, Ahmed *et al.* [ATJC12] proposed an algorithm to avoid candidate generation for interactive stream mining. They proposed a tree structure called High Utility Stream Tree (HUS-Tree) and an algorithm called High Utility Pattern Mining over data stream (HUPMS) for incremental mining. HUS-Tree captures information batch by batch, and HUPMS removes the old batch when the window slides. The header table contains both the item id and the TWU. HUPMS requires additional window scans to determine the actual high HUPs and the HUS tree is to calculate the potential candidates. Unlike DSTree, the HUS-Tree is only used for capturing TWU information for generating candidates, and the mining process requires re-scanning the current batch window. The TWU model generates candidates and to determine the HUI required additional database scans. We show the HUPMS example in Example 2.5.3. Based on HUPMS, Ryang and Yun [RY16] proposed a more efficient tree-

based data structure called SHU-tree using the decreased overestimated utilities (RTWU) to maintain high utility patterns over the data stream. The definition of RTWU is defined in the Definition 5 in the original paper. The RTWU is the sum of the estimated item utility where only the item utility and its previous items utility in a transaction are considered. This reduces both the TWU in the header table as well as the node utilities. The downside of HUPMS and SHU-tree is that they both require additional database scan.

Example 2.5.3. (HUPMS Algorithm Example) Let us consider the database in Table 2.27. HUPMS algorithm divides database into batches. Let the batch size be 2 and the window size be 3 batches. HUPMS algorithm updates the HUS-tree on each batch as shown in Figure 2.5. When the window is full, The TWU value in each node would shift and those nodes which contain all 0 will be removed from HUS-tree.

Table 2.27: A sample transaction database with utility value to demonstrate HUPMS algorithm

TID	Items	Utilities	Total Utilities
1	d,e	24,40	64
2	a,c	4,24	28
3	b,c,d	12,24,16	52
4	a,b	8,48	56
5	b,e	18,20	38
6	a,b,d	12,30,32	74
7	a,b,c	4,12,21	37
8	a,b,d,e	8,24,40,30	102

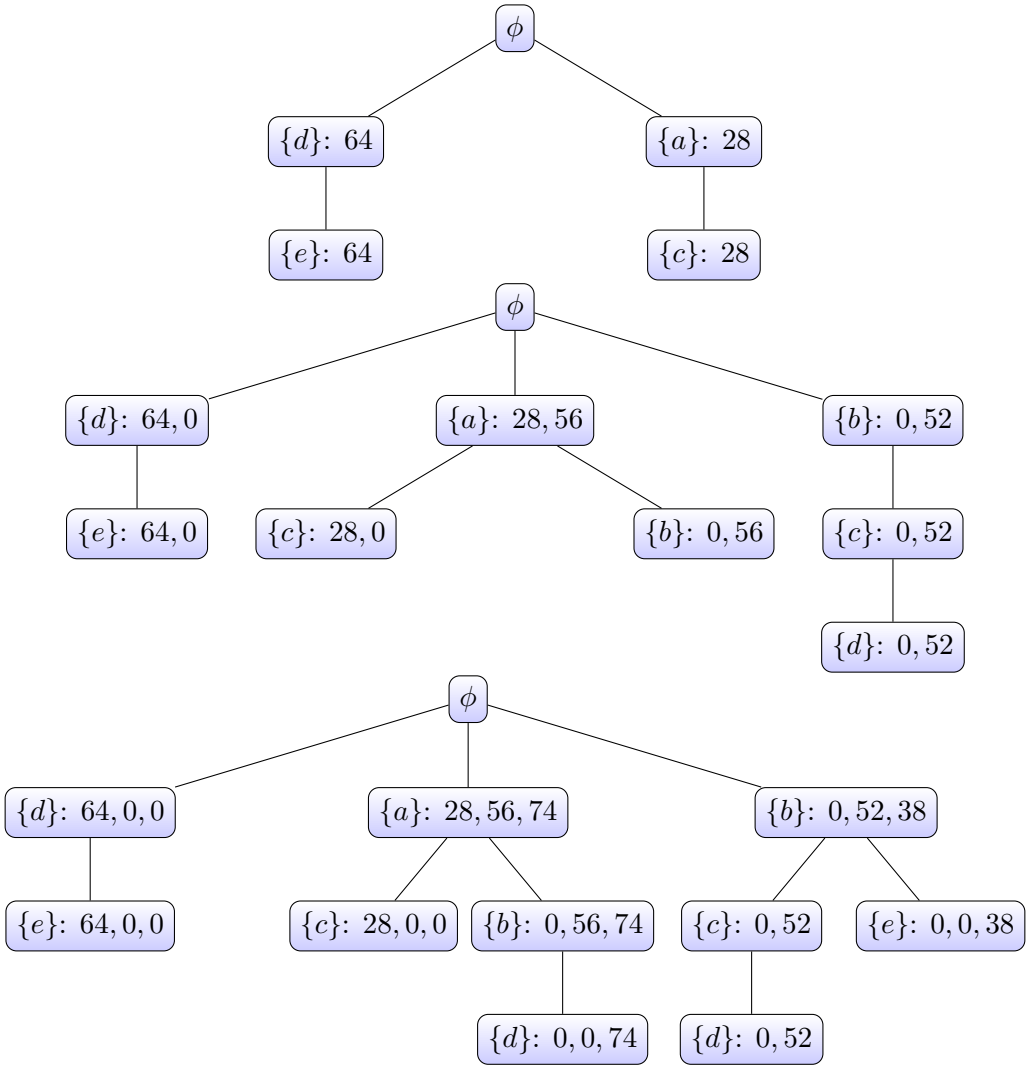

Figure 2.5: HUS-Tree construction sample

Example 2.5.4. (SHU-Tree Stream Miner Example) Let us consider the database showed in Table 2.27. Let the batch size be 2 transactions and the window size be 3 batches (6 transactions). In Figure 2.28, we can see the SHU-tree is similar to HUS-tree but it captures RTWU instead of TWU. We demonstrate the key concept and the full example is in the original paper. The SHU-tree having a tighter overestimated upper bound makes it more efficient than HUS-tree.

Table 2.28: A sample transaction database with utility value to demonstrate SHU-stream miner algorithm

TID	Items	Utilities	Total Utilities
1	a,b,d	3,6,14	23
2	a,b,c,e	6,2,16,15	39
3	c,e	4,20	24
4	a,b,d	9,4,21	34

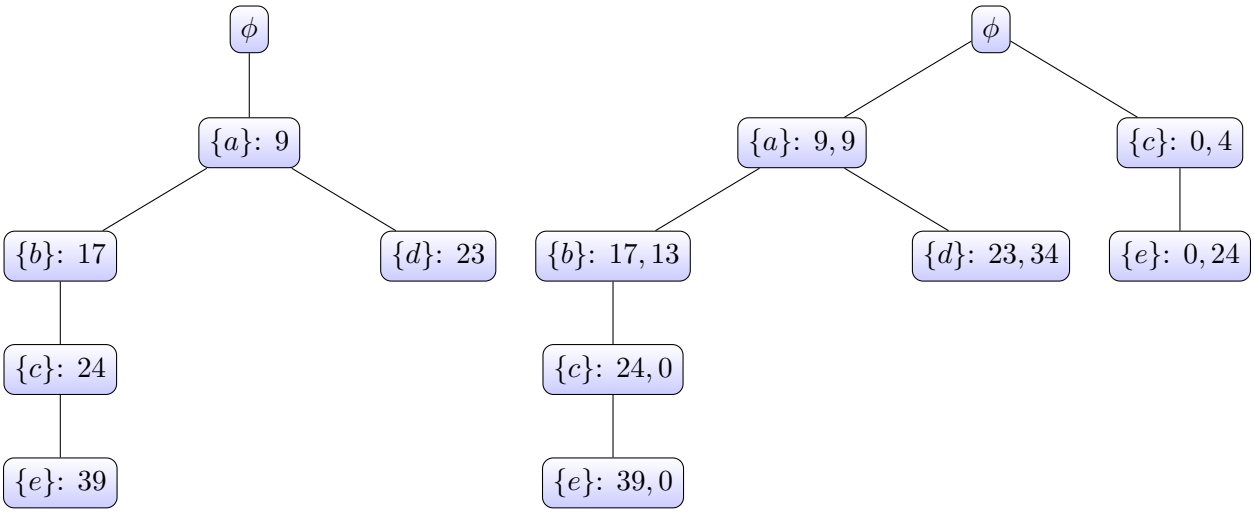

Figure 2.6: SHU-Tree construction sample

2.5.5 High Utility Data Stream Mining Algorithm: SOHUPDS

As mentioned above, the existing algorithms for mining high utility patterns over a database are all two-phase algorithms in essence. Jaysawal and Huang [JH20] proposed the first one-phase algorithm called High Utility Patterns over a Data Stream (SOHUPDS) for mining high utility patterns. It uses a projected database strategy and utilizes the previous sliding window for updating the current sliding window. We show the SOHUPDS algorithm in Example 2.5.5. The algorithm in this section are designed for mining high utility itemsets in a dynamic stream when the external unit profit is constant. In contrast, our proposal explores when the external unit profit is dynamic.

Example 2.5.5. (SOHUPDS Algorithm Example) Let us consider the database in Table 2.29 to demonstrate the SOHUPDS algorithm. The data structure constructs a node for every item. For example, after the first transaction is inserted, 3 IUDataListSW nodes are created for item a, b and d. After the first batch is inserted. 6 nodes are created for item a,b,c,d,e,f. We show one of the node in Figure 2.7. It captures a lot of information of the item such which transaction it came from (tIndex), the position of the item within a transaction (position). Those information is equivalent to reconstruct the database for mining purposes.

Table 2.29: A sample transaction database with utility value to demonstrate SOHUPDS algorithm

TID	Items	Utilities	Total Utilities
1	a,b,d	4,6,2	12
2	a,c,e,f	8,4,6,5	23
3	a,b,c,d,e,h	4,6,2,10,3,2	27
4	b,c,d,e	12,3,6,3	24
5	b,c,e,g	6,1,3,6	16
6	c,d,e	1,6,3	10
7	a,b,f	8,12,1	21
8	c,e	2,3	5

Most recently, Nam *et al.* [NYYL20] proposed recent high utility mining with indexed list structure. It is based on HUI and with window decay factor. Baek *et al.* [BYK+21] introduced RHUPS algorithm which claimed to be the first work to find time-sensitive stream database. It uses the list data structure and the sliding window model. Under our evaluation, its main difference compared to SHU-stream miner algorithm is its window function which manage a certain amount of recent data to achieve better performance. The result and the number of candidates are different compared to the previous work so it is more like a feature extension of high utility data stream mining.

Kim *et al.* [KYB+21] proposed an algorithm for analyzing damped stream data called DSHUP which is based on SHU-growth [RY16], and it applies a window decay factor on the old windows to generate fewer candidates. When the window decay factor is 1, DSHUP is essentially the same as SHU-growth. Kim *et al.* [KYB+21] proposed the damped mining for average utility pattern called DMAUP to extract the average utility overtime which is a different direction of research of processing time sensitive information from data streams. Cheng *et al.* [CHZ+21] proposed an efficient algorithm called ETKDS for mining Top-K high

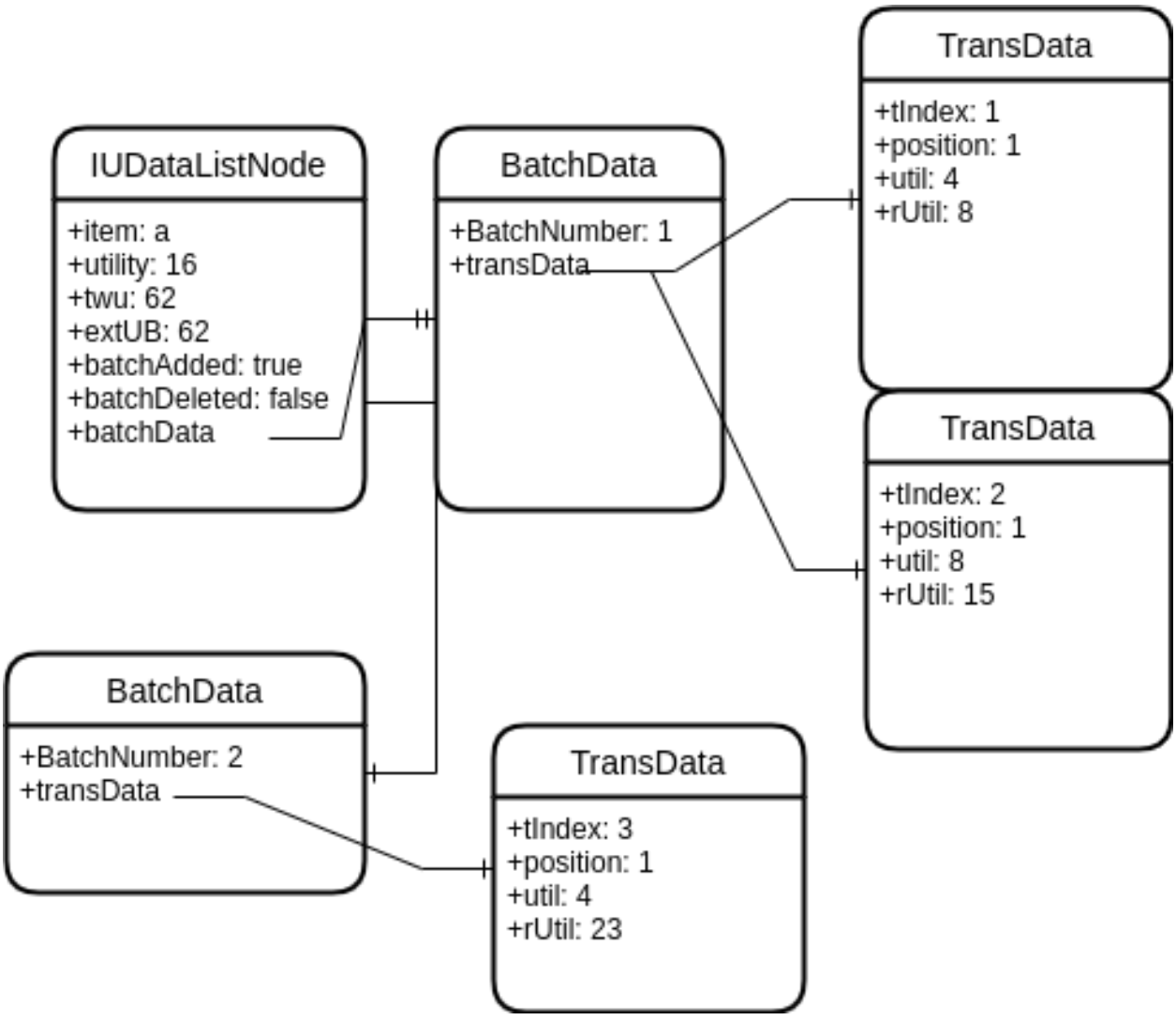

Figure 2.7: The IUDataList node of SOHUPDS algorithm

utility itemsets over data streams using sliding window model, it is similar to SOHUPDS which processes the data in batch model and uses TWU as the pruning strategy. The latest related research is proposed by Mondal *et al.* [MMCR22], they proposed the notion of diversification to reduce the number of itemset generation. It redefines the notion of high utility which is different from the goal of our paper.

Chapter 3

Mining High Utility Patterns from Data Streams

In this chapter, we describe our research works and contributions. Recall from the background chapter, mining from a static database is a special case of mining from data streams when the dataset size is finite. Because of the similarity, any algorithm that can mine HUI from data streams can mine HUI from static databases. However, algorithms designed for mining static databases are not ideal for mining data streams. As mentioned in the background section, algorithms based on two-phase are not practical to use in the real world scenario for mining data streams as they require re-scanning transaction records. On the other hand, algorithms based on UP-tree cannot capture all the necessary information for mining in the tree structure, so the tree structure can only serve as a helper to assist pruning and additional database scans are still required to determine the final candidates.

Inspired by utility list algorithms which have the advantage of preserving transaction information for mining high utility itemsets without re-scanning databases, we would like to

explore the potential of this type of algorithm for mining HUI data streams. We propose a new data structure called Enhanced-Utility-List (EUL) which is the data model for our proposed algorithm called EUL-Stream-Miner for mining high utility patterns from data streams. Our proposed algorithm has the advantage of mining more efficiently compared to the baselines and it has the advantage of continuous mining when the price changes without the need of re-scanning the data streams.

3.1 Enhanced Utility List

We propose Enhanced Utility List (EUL) as the data structure to capture incremental / streaming transactions in a compressed format designed for high utility itemset mining. We also propose two new definitions called Item Remaining Utility Map (refer to Definition 13) and RutilityMap (Refer to Definition 14) as a part of the EUL data structure. By using EUL, there is no need to re-scan data streams and the mining process is highly efficient. From a high-level point of view, each unique item in a transaction will generate an associated EUL, so the total number of EUL is equal to the total number of unique items. The mining process is performed on a list of EULs. In Table 3.2, we demonstrate the EUL data structure. Each EUL contains the item name as the key ID for differentiating the EUL. When an item in a transaction matches the item name in the corresponding EUL, its TID, Utility (Refer to Definition 2), Remaining Utility (Refer to Definition 11) will be added to the corresponding EUL.

Definition 13. (Item Remaining Utility Map) Let X be an itemset in the transaction T, and x_i is an item in an itemset X, denoted as $x_i \in X$, $i \in \mathbb{Z}^+$. The item remaining utility of item x_i is equal to the summation of its item utility and its remaining utility, denoted as

$iru(x_i) = u(x_i) + ru(x_i, T)$. The Item Remaining Utility Map uses the item name as the key (K) and its item remaining utility as value (V), denoted as $\{x_i \rightarrow iru(x_i)\}$. For all the items in the itemset X, its Item Remaining Utility Map is defined as $iru\_map(X) = \forall x_i \in X, \{x_i \rightarrow iru(x_i)\}$.

Example 3.1.1. (Item Remaining Utility Map Example) Let $X = \{a, b, c\}$ with associated utility values $\{2, 4, 5\}$. $iru(a) = u(a) + ru(a, T) = 2 + 9 = 11$, $iru(b) = u(b) + ru(b, T) = 4 + 5 = 9$, $iru(c) = u(c) + ru(c, T) = 5 + 0 = 5$. As a result, $iru\_map(\{a, b, c\}) = \{a \rightarrow iru(a), b \rightarrow iru(b), c \rightarrow iru(c)\} = \{a \rightarrow 11, b \rightarrow 9, c \rightarrow 5\}$.

Definition 14. (RutilityMap) Let $T_1, T_2, \ldots T_n \subseteq D$, where T_i is the transaction and D is the database. Let X_i be an itemset in transaction T_i, and $x_j \in X_i$, $i, j \in \mathbb{Z}^+$. The RutilityMap is the aggregation of all item remaining utility maps which is formally defined as $RutilityMap = \sum iru\_map(X_i)$.

Example 3.1.2. (RutilityMap Example) Let $X_1 = \{a, b, c\}$ with associated utility values $\{2, 4, 5\}$ and $X_2 = \{b, c\}$ with associated utility values $\{1, 3\}$. $iru(a, T_1) = u(a) + ru(a, T_1) = 2 + 9 = 11$, $iru(b, T_1) = u(b) + ru(b, T_1) = 4 + 5 = 9$, $iru(c, T_1) = u(c) + ru(c, T_1) = 5 + 0 = 5$, $iru(b, T_2) = u(b) + ru(b, T_2) = 1 + 3 = 4$, $iru(c, T_2) = u(c) + ru(c, T_2) = 3 + 0 = 3$. As a result, $iru\_map(\{a, b, c\}) = \{a \rightarrow iru(a), b \rightarrow iru(b), c \rightarrow iru(c)\} = \{a \rightarrow 11, b \rightarrow 9, c \rightarrow 5\}$, $iru\_map(\{b, c\}) = \{b \rightarrow iru(b, T_2), c \rightarrow iru(c, T_2)\} = \{b \rightarrow 4, c \rightarrow 3\}$. Finally, $RutilityMap = iru\_map(\{a, b, c\}) + iru\_map(\{b, c\}) = \{a \rightarrow 11, b \rightarrow 13, c \rightarrow 8\}$.

Example 3.1.3. (The Advantage of RutilityMap of Pruning The Search Space) Let us consider the sample dataset in Table 3.1. If the minimum utility threshold be 9. No items are pruned by TWU at the first level since all the TWU of all items are greater than 8. $TWU(a) = 16$, $TWU(b) = 23$, $TWU(c) = 10$, $TWU(d) = 11$, and $TWU(e) = 10$.

As for the remaining utility, the remaining utility of a is 14, item a would try to join item c and item e itemset ac and itemset ae are not high utility itemsets.

By using RutilityMap, $iru\_map(\{a\}) = \{b \rightarrow 5, c \rightarrow 3, e \rightarrow 5\}$. The total utility of a is 3, by checking the RutilityMap, we observe that it is not worth joining item c and item e as the sum would be less than the minimum utility. As a result, it would save computational cost.

Table 3.1: A sample dataset to demonstrate the advantage of RutilityMap

TID	Items	Utilities	Total Utilities
1	a,b,c	1,2,3	6
2	a,b,e	2,3,5	10
3	c,d	2,2	4
4	b,d	2,5	7

Table 3.2: Enhanced Utility List data structure

Item Name			
TID	**Utility**	**Remaining Utilities**	**Item Remaining Utility Map**
RutilityMap			
Sum Utility	**Sum Remaining Utility**		

Example 3.1.4. (Add Item To EUL)) When the transaction in Table 3.3 is inserted to EUL, there will be three EULs created for items a, b and d. For simplicity, we will focus on the item a EUL. As demonstrated in Table 3.4, the associated TID of the item a is T_1. Its utility is 1 and its remaining utility is the total item utility after a which is 7. The Item Remaining Utility Map is the maximum utility for each remaining item if they are the prefix. In this case, the item b has the maximum utility of $\{b : 4\} + \{d : 3\} = 7$, and the item d has the maximum utility of itself which is 3 because it is the last item in the transaction T_1. The

Sum Utility is 1 by combining all the transactions in the EUL, and the Remaining Utilities is 7 by combining all the Remaining Utilities in the EUL. The RutilityMap is the aggregation of the Item Remaining Utility Map. In this case, it is the same as Item Remaining Utility Map since there is only 1 transaction.

Table 3.3: The sample transaction

TID	Items	Utilities	Total Utilities
T_1	a,b,d	1,4,3	8

Table 3.4: Enhanced Utility List data structure for a sample

Item Name: a			
TID	**Utility**	**Remaining Utilities**	**Item Remaining Utility Map**
T_1	1	7	$\{b:7, d:3\}$
RutilityMap : $\{b:7, d:3\}$			
Sum Utility: 1		**Sum Remaining Utility**: 7	

Example 3.1.5. (Merge EULs operation)) Each EUL represents an individual item in a transaction. The operation of merging EULs is required when the pattern grows. For example, joining EUL-a to EUL-b will form EUL-joint-ab. In Example 3.1.4, we compute EUL-b using the same approach in Example 3.1.4 and the result would be Table 3.5. During the joining process, for each intersection transaction between EUL-a and EUL-b, the utility values would be sum together yielding 5. The remaining utility is the same as the remaining utility in the EUL that joins with the prefix EUL. In this case, the remaining utility of EUL-joint-a-b is the same as the remaining utility in EUL-b which is 3, and the Item Remaining Utility Map is the same as the one in EUL-b which is $\{d:3\}$. As a result, the joint utility of EUL-a and EUL-b would be Table 3.6.

Table 3.5: Enhanced Utility List data structure for item b (EUL-b)

Item Name: b			
TID	**Utility**	**Remaining Utilities**	**Item Remaining Utility Map**
T_1	4	3	$\{d:3\}$
RutilityMap : $\{d:3\}$			
Sum Utility: 4		**Sum Remaining Utility**: 3	

Table 3.6: Enhanced Utility List data structure for item b (EUL-joint-ab)

Item Name: ab			
TID	**Utility**	**Remaining Utilities**	**Item Remaining Utility Map**
T_1	5	3	$\{d:3\}$
RutilityMap : $\{d:3\}$			
Sum Utility: 5		**Sum Remaining Utility**: 3	

3.1.1 Design Analysis

As mentioned in the background section, the key to enhancing the high utility mining algorithm performance is to reduce the search space. In the traditional frequent pattern mining, the downward closure property is powerful to prune the search space. When an itemset X is infrequent, its extensions X-*extensions* must be infrequent and can be pruned away; when an itemset X is frequent, all the subsets of the itemset X must be frequent. For HUIM, when an itemset X is not a high utility pattern, its extensions X-*extensions* could be high utility patterns. The key property of HUIM is that rare items could contribute more value regardless of their frequency that would break the downward closure property. When an itemset X is not a high utility pattern, the number of occurrences of extensions X-*extensions* must be less than or equal to the number of occurrences of the itemset X. With the property of unit pricing can be dynamic, the downward closure property would be

less relevant for HUIM as the utility of a particular item could fluctuate. Without getting into the domain of price analysis, how can we still prune the search space effectively?

One of the breakthroughs of HUIM is the discovery of the downward closure property of TWU in Definition 10. When the TWU of an itemset X is less than the user-defined minimum utility threshold, all its extensions, *X-extensions* must not be high utility patterns and can be pruned away. When the TWU of the itemset X is greater than the minimum utility threshold, all TWU of the subsets must be greater than the minimum utility. Having TWU is the workaround compared to the original frequent pattern mining. Since lower frequency items can contribute to more value than higher frequency items that break the downward closure property, TWU includes all the values in a particular transaction where an item appears by summing up the transaction utility, so its extensions,*X-extensions*, must appear in the subset of the transactions that the item X appears, so the downward property holds. The key condition of triggering the downward closure property of TWU for pruning the search space is when the TWU of an itemset is less than the minimum high utility value. We mentioned TWU is a loose upper bound in the background section since its value can be way larger than the individual item utility. We need to find a way to narrow the search space.

The other breakthrough of HUIM is the discovery of using the remaining utility in pruning the search space. For an itemset X, its extension utility would be the total item utility that appeared after X. This is a tighter upper bound compared to TWU. Compared to TWU, the remaining utility avoid double-counting the item utility. We demonstrate the advantage of Remaining Utility over TWU in Example 3.1.6. As we mentioned in the background section, the itemset order can be re-arranged as long as the order is consistent throughout the transaction stream. The order can affect the number of items to be explored as well

because the order of prefix to be explored is different when the transaction order is different. Remaining utility is an important indicator of whether an itemset is worth exploring or not. Therefore, it is one of the important components of Enhanced Utility Lists. The next question is can we tighten the upper bound further?

Example 3.1.6. (The Advantage of Remaining Utility Over TWU Example)) Let us consider an itemset $\{a, b, c, d, e\}$ with associated utility values $\{1, 1, 1, 1, 1\}$. The TWU of the item a, b, c, d, e including all their supersets are all 5. As the pattern growth progress started, TWU does not change progressively according to the mining progress. If the item c is the prefix of the potential candidates, the potential extensions do not need to include a and b anymore thus can be excluded. However, TWU includes all the transaction utility including X as the upper bound, but the remaining utility approach includes all the utility only if X is the potential prefix. As a result, Remaining Utility is a tighter upper bound than TWU. The consistency of the itemset order is to guarantee Remaining Utility is valid.

The number of itemsets that is qualified for a prefix can be further reduced. When we extend an itemset of the length k to the itemset of the length $k+1$, we need to differentiate whether the next item is worth exploring. For a utility list X and a utility list Y, will utility XY be still qualified as a prefix when we join two utility lists together? To solve this problem, we need more information than knowing the remaining utility as the remaining utility represents all item utility that is appeared after the prefix item. The remaining utility takes the individual transaction into account. We demonstrate the limitation of Remaining Utility in Example 3.1.7.

Example 3.1.7. (The Limitation of Remaining Utility Example) In Table 3.7, the remaining utility of the itemset $\{a\}$ is 4, the remaining utility of the itemset $\{b\}$ is 3, and the remaining utility of the itemset $\{a, b\}$ is 0. However, the traditional utility list would

consider the remaining utility of the itemset $\{a, b\}$ as 3 since it only captures the total utility and the total remaining utility, so it would not be able to efficiently compute the remaining utility for each itemset.

Table 3.7: The sample transaction to demo the limitation of Utility List

TID	Items	Utilities	Total Utilities
T_1	a,b	1,4	5
T_2	b,d	4,3	7

Our proposed solution is to create a map inside the original utility list to dynamically compute the remaining utility with respect to other utility lists. The remaining utility map would tighten upper bound further. In the background section, we mentioned a couple of algorithms for mining high utility itemsets over data streams. But in our opinion, those are not native data stream algorithms as they need to rescan the dataset to compute the final candidates. The key to solving this problem is to build a data structure that is ready for HUI mining proposes. Such data structure model would be dynamically updated when the information flows in and users can request the high utility itemsets in real-time. Having good performance is only solving part of the problem, but our data structure is also required to handle temporal/streaming data without any data rescanning. We will show the benefit of our enhanced utility list in Section 3.2 when we demonstrate our Enhanced HUI-Stream Mining algorithm.

3.2 Enhanced-HUI-Stream-Mining Algorithm

The building block of our enhanced-HUI-Stream-Mining algorithm is the enhanced utility list (EUL). In Figure 3.1, our high utility mining algorithm is responsible for consuming

data streams, constructing the enhanced utility list from data streams, and mining using the enhanced utility list.

Figure 3.1: Enhanced-HUI-Stream-Mining-Model

Given a user-specified minimum utility threshold, our goal is to mine all high utility itemsets from data streams. The enhanced-HUI-Stream-Mining algorithm transforms every transaction into the data structure with a list of utility lists.

Example 3.2.1. (EUL construction Example) Let the minimum utility threshold be 30. In Figure 3.2, each transaction in data streams must be transformed as soon as it arrives. In this case, each transaction is sorted by the canonical order (Definition 15) with the specification by the item name in the alphanumeric ascending order. The transaction T_1: $\{c:1, e:3, a:5, b:10, d:6, f:5\}$ transforms to $\{a:5, b:10, c:1, d:6, e:3, f:5\}$. The same sorting rule is applied to every other incoming transaction in the data stream. The benefit of sorting by canonical order is that no reconstruction of the EULs is needed despite the data being inserted, deleted, or modified.

Each transaction is then inserted into the list of EULs. Initially, the EULs are empty. When the first transaction T_1: $\{a:5, b:10, c:1, d:6, e:3, f:5\}$ arrives, the algorithm checks the existence of each item in the transaction and determines whether a corresponding EUL has been created. If there is no corresponding EUL, a new one will be created and inserted the item to the EUL. If there is an existing EUL, the item would be inserted into the existing EUL. Each entry in the EUL represents an item in the transaction. Its transaction ID, its utility, its remaining utility and the item remaining utility map would be recorded

by the way we mentioned in Example 3.1.4. Once the EUL is updated, its sumUtility, sumRemainingUtiltiy, and the RutilityMap would be updated in real-time.

Definition 15. (Canonical Order) Canonical order means the order is defined by a separate specification.

Example 3.2.2. (Canonical Order Example) The alphanumeric ascending/descending order is one of the canonical order specifications. Let T be a transaction with the itemset $\{c:1, b:3, a:5\}$. The transaction transforms to $\{c:1, b:3, a:5\}$ after being sorted by alphanumeric ascending order.

Unlike the related work on high utility data stream mining, there is no concept of the batch in our algorithm since the list of EUL would be updated in real-time as soon as the data arrives. Users can request mining high utility itemsets from our list of EULs without re-visiting the data source.

Example 3.2.3. (Enhanced-HUI-Streaming algorithm Example) Let the user-defined minimum utility threshold be 30 and the mining process is started after the transaction T_5 arrives. In Table 2.2, the mining process would be performed on data structure as demonstrated in Figure 3.2.

There are 7 EULs created after T_5 arrives. Each of the EULs is associated with a unique item in the data stream, namely, EUL-a, EUL-b, EUL-c, EUL-d, EUL-e, EUL-f and EUL-g. Each of the initial EULs is corresponding to a singleton itemset. If the SumUtility is greater than or equal to the user-defined minimum utility threshold, the associated item in EUL is a high utility itemset. In this case, the sumUtility of EUL-a is 20, EUL-b is 22, EUL-c is 13, EUL-d is 14, EUL-e is 15, EUL-f is 5 and EUL-g is 7. None of the EULs has SumUtility greater than or equal to the minimum utility threshold, so there is no Level 1 high utility

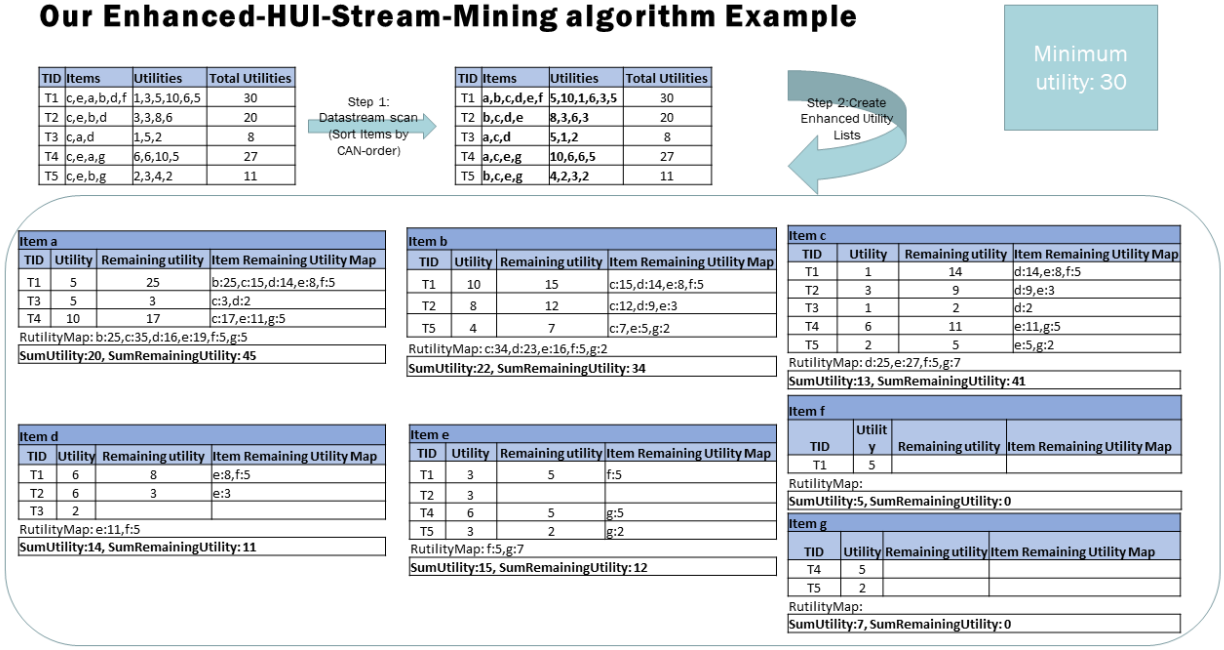

Figure 3.2: Enhanced-HUI-Stream-Mining-algorithm

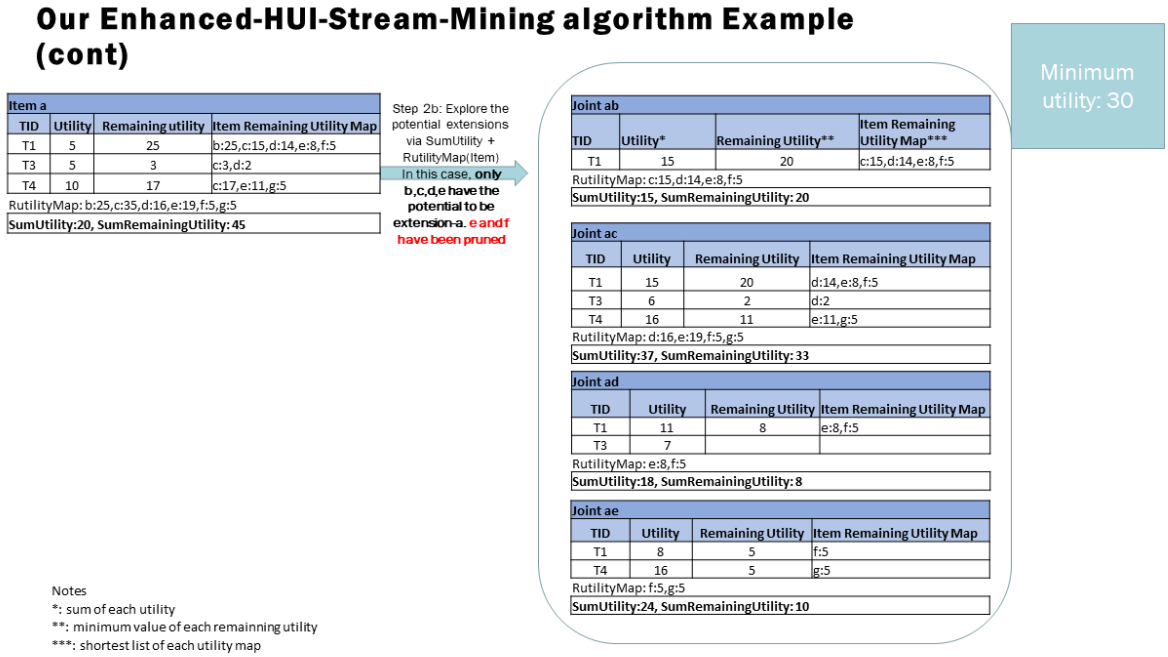

Figure 3.3: Enhanced-HUI-Stream-Mining-algorithm-Continue

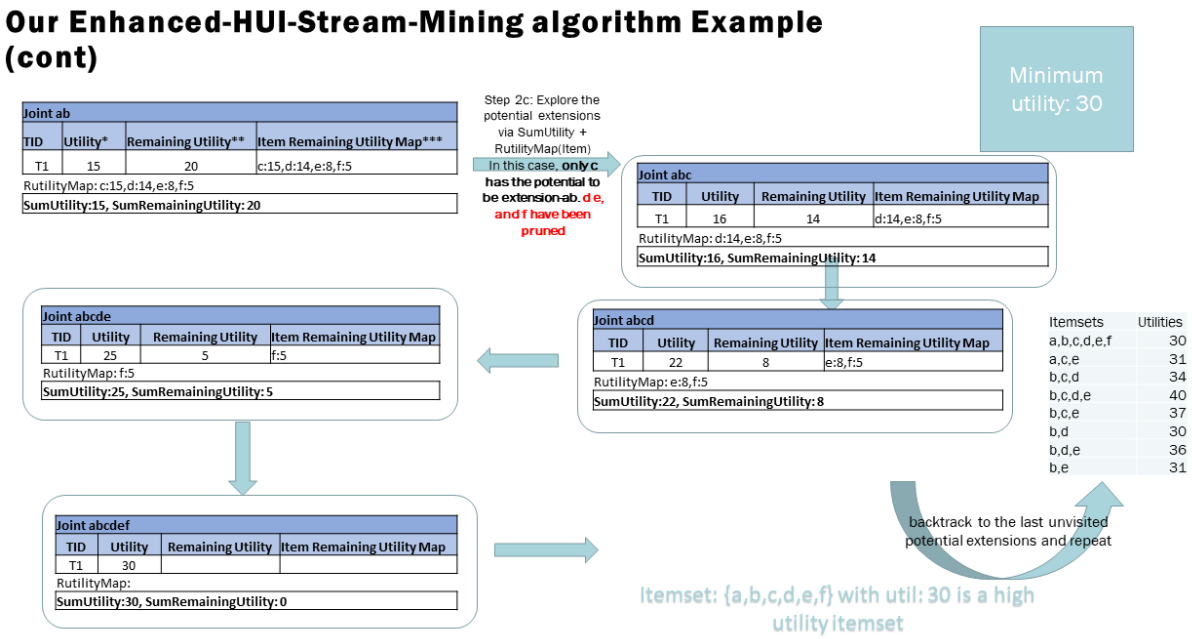

Figure 3.4: Enhanced-HUI-Stream-Mining-algorithm-Continue

itemsets.

Each EUL can join the EUL afterwards to create a new EUL. For example, EUL-a can join { EUL-b, EUL-c, EUL-d, EUL-e, EUL-f, EUL-g } and EUL-b can join { EUL-c, EUL-d, EUL-e, EUL-f, EUL-g}. The same rule is applied to the rest of the EULs. Before joining two EULs together, there are two conditions as filters to reduce the number of joining. The first condition is that the summation of SumUtility and the SumRemainingUtility must be greater than or equal to the user-defined minimum threshold. If the criteria are not met, the EUL is not qualified for being extended. Otherwise, it would step into the second condition which is the summation of SumUtility and the corresponding other value in the RUtilityMap must be greater than or equal to the user-defined minimum utility threshold to be qualified as potential prefix EUL. For example, the sum of SumUtility and the SumRemainingUtility in EUL-a is equal to $20 + 45 = 65$ which is greater than 30. For the rest of the EULs, the

result is as follows:

- EUL-b: The total of SumUtility and the SumRemaining Utility is $22 + 34 = 56$, which is greater than the minimum utility threshold.
- EUL-c: The total of SumUtility and the SumRemaining Utility is $13 + 41 = 54$, which is greater than the minimum utility threshold.
- EUL-d: The total of SumUtility and the SumRemaining Utility is $14 + 11 = 25$, which is less than the minimum utility threshold.
- EUL-e: The total of SumUtility and the SumRemaining Utility is $15 + 12 = 27$, which is less than the minimum utility threshold.
- EUL-f: The total of SumUtility and the SumRemaining Utility is $5 + 0 = 5$, which is less than the minimum utility threshold.
- EUL-g: The total of SumUtility and the SumRemaining Utility is $7 + 0 = 7$, which is less than the minimum utility threshold.

By going through the first filter, we know that only EUL-a, EUL-b, and EUL-c are qualified for being extended as the prefix-EUL, and there is no need to explore EUL-d, EUL-e, EUL-f and EUL-g for the reason they do not satisfy the minimum utility threshold.

Then, EUL-a, EUL-b and EUL-c go through the second filter. Let us use EUL-a as an example. Although EUL-a can potentially join all the remaining EULs, the joint result is not guaranteed to be qualified as the high utility itemsets. The summation of the sumUtility and the RutilityMap result is as follow:

- SumUtility of a + RutilityMap of b in EUL-a $= 20 + 25 = 45$ which is greater than the user-defined minimum utility threshold.

- SumUtility of a + RutilityMap of c in EUL-a = $20 + 35 = 55$ which is greater than the user-defined minimum utility threshold.
- SumUtility of a + RutilityMap of d in EUL-a = $20 + 16 = 36$ which is greater than the user-defined minimum utility threshold.
- SumUtility of a + RutilityMap of e in EUL-a = $20 + 19 = 39$ which is greater than the user-defined minimum utility threshold.
- SumUtility of a + RutilityMap of f in EUL-a = $20 + 5 = 25$ which is less than the user-defined minimum utility threshold.
- SumUtility of a + RutilityMap of g in EUL-a = $20 + 5 = 25$ which is less than the user-defined minimum utility threshold.

Filtering by the RutilityMap, EUL-a would not join EUL-f and EUL-g for not meeting the minimum utility. EUL-joint-af and EUL-joint-ag are not qualified for being the prefix EUL and all the extensions regarding EUL-joint-af and EUL-joint-ag would be pruned away so the search space would be greatly reduced. EUL-a would join EUL-b, EUL-c, EUL-d, and EUL-e as demonstrated in Figure 3.3. When joining two EULs together, rows with the same TID would be merged similarly as Example 3.6, and the merge result will be in Figure 3.3. Five EULs, EUL-joint-ab, EUL-joint-ac, EUL-joint-ad, and EUL-joint-ae, are created. None of them are high utility patterns as their sum utility are less than the user-defined minimum threshold.

The algorithm works recursively. To explore the potential extension of EUL-joint-ab in Table 3.8, we need to check the two filter conditions. In this case, for the first filter, the total of the SumUtility and the SumRemainingUtility is $15 + 20 = 35$ which is greater than

Algorithm 1 Enhanced HUI-Stream Miner

Require: PUL, ULs, potentialPULs, minUtil

1: **for** $potentialPUL \leftarrow potentialPULs$ **do**
2: **if** $potentialPUL.sumUtil \geq minUtil$ **then**
$OutputHighUtilityItemset()$
3: **end if**
4: **if** $potentialPUL.sumUtil + potentialPUL.sumRUtil \geq minUtil$ **then**
5: $NextLevelExtensionULs \leftarrow newList$
6: $NextLevelExtensionPULs \leftarrow newList$
7: **for** $UL \leftarrow ULs$ **do**
8: **if** $UL.item = PUL.item$ **then**
CONTINUE
9: **end if**
10: **if** $PUL.getDynamicSumRUtil(UL.item) \neq null$ **then**
$NextLevelExtensionULs.add(joint(PUL, UL))$
if $PUL.sumUtils+PUL.getDynamicSumRUtil(UL.item) \geq minUtil$ **then**
$NextLevelExtensionPULs.add(joint(PUL, UL))$
11: **end if**
12: **end if**
13: **end if**
14: $HUIStreamMiner(potentialPUL, NextLevelExtensionULs, NextLevelExtensionPULs)$
15: **end for**
16: **end if**
17: **end for**

the minimum threshold. For the second filter, only c has the potential to extend ab. The result is demonstrated in Figure 3.4 by mining recursively.

Table 3.8: Enhanced Utility List data structure for (EUL-joint-ab)

Item Name: ab			
Sum Utility: 15		**Sum Remaining Utility**: 20	
TID	**Utility**	**Remaining Utilities**	**Item Remaining Utility Map**
T_1	15	20	$\{c : 15, d : 14, e : 8, f : 5\}$
RutilityMap : $\{c : 15, d : 14, e : 8, f : 5\}$			

In Algorithm 1, we show the pseudocode of our enhanced HUI-Stream Miner. The notation definitions are listed as follows:

- **PUL**: The prefix enhanced utility list, which is the utility list to be extended.
- **ULs**: The potential enhanced utility lists to join PUL.
- **potentialPULs** The list of potential PULs after the joint process. They are created to serve as the next level of potential enhanced prefix utility list.
- **minUtil**: The user-defined minimum utility value which is used throughout the algorithm.

3.2.1 Mining under Data Addition Scenarios

When a new transaction record arrives, each transaction would update the data structure incrementally. In the EUL data structure, each EUL contains a list of records and each record is associated with a TID. Since every new record contains a new transaction id, all the existing records in EULs do not need to be changed but only a new extension is required so the computational cost is low regardless of the size of the data.

Example 3.2.4. (Add Record to EUL Example)) In Figure 3.5, there is a new transaction called T_6 with the transaction itemset $\{a : 3\}$ arriving. The number of EULs to be updated is equal to the size of the transaction itemset. Only EUL-a is required an update with a new record. Its sumUtility, sumRemainingUtility, and the RutilityMap are updated accordingly.

Algorithm 2 is the pseudocode for our enhanced-HUI-Stream mining to deal with new incoming records. The notations used by the pseudocode are listed as follow.

- **ULs**: The potential Enhanced Utility Lists
- **Transaction**: The transaction to be added

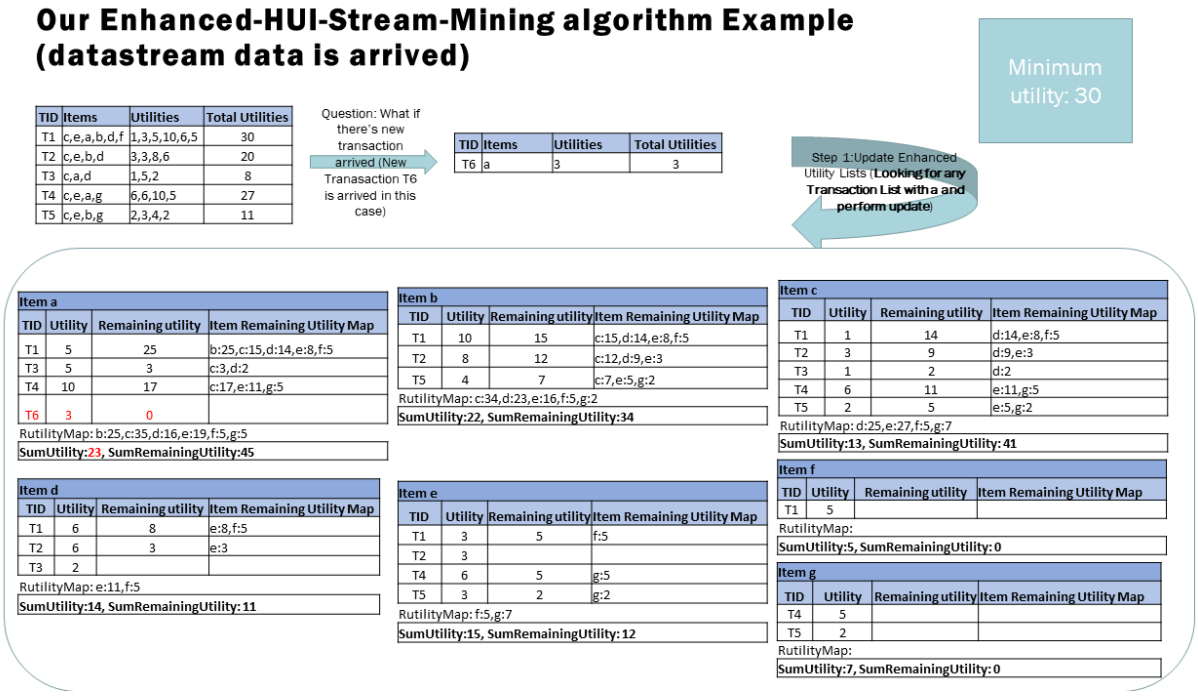

Figure 3.5: Enhanced-HUI-Stream-Mining-algorithm-new-record

- **UL-Record**: The entity that contains TID, Utility, Remaining Utility and Item Remaining Utility Map in the associated Enhanced Utility List

- **transformItemToEULRecord** The function that converts an item in a transaction to an entity in the Enhanced Utility List

Algorithm 2 Enhanced HUI-Stream Miner - Add record

Require: ULs, Transaction

1: **for** $Item \leftarrow Transaction$ **do**
2: $\quad UL - Record \leftarrow transformItemToEULRecord(Item, Transaction)$
3: $\quad createOrUpdate(UL - Record, ULs)$
4: **end for**

3.2.1.1 Optimizations for the next mining after data addition

Our EHUI-Stream algorithm would preserve the list of EULs to prepare for the next mining process after data addition. Inserting records to EUL is highly efficient and the EHUI mining process should be efficient based on the analysis, but there is an important question that remained. What is the advantage of this algorithm compared to some of the batch processing algorithms? For batch processing, the data can be processed offline in batches and have optimization on the dataset before processing which is the advantage over our stream mining scenario since we have to process the data as soon as it arrives. The key advantage of the stream mining algorithm is not about the raw performance over the best static mining algorithm, but the response time during the mining process. When an additional record arrives and the user request mining again, how much time does it take to output the result with the data differences? Our EHUI-Stream algorithm would be able to fill the gap of this issue.

Refer to Figure 3.5, when the new transaction,T_6 arrives, it has item a with the utility of 3. Assuming a user requested a mining process before T_6 arrives, and they request another round of mining process after T_6 arrives. The naive solution is to run the proposed algorithm we demonstrated in Algorithm 1 again. Despite the algorithm being efficient, it is not ideal for the following reasons.

- Every mining process would start from the beginning regardless of the scope of the data change. After mining a massive amount of data in the EUL-Stream data structure, if the user requests another mining process after a few more transactions are inserted, the mining time would increase compared to the previous one assuming that there is no outdated data is removed and the size of data is increased. This poses a limit on

the algorithm on how much data it can mine for a reasonable amount of response time.

- Data stream is continuous and unbounded, and the user would request mining frequently. If the mining takes too long the result would be already outdated due to continuous data flow.

Knowing the response time is crucial for our stream mining algorithm, but how can we optimize the stream mining process to reduce the response time and to avoid the proportional run-time increment due to the data volume increased overtime? The naive solution is to implement a sliding window over the data stream so that we always keep the size of data at the range that a system can handle. This workaround solves the run-time issue in some way by avoiding the accumulated records growing beyond the capability that a system can handle, but it does not address the issue that a larger window would have slower runtime because there are more records inside a given window.

The key to solving this issue is that we need a way to determine the scope of the mining after the transactions are inserted. In Figure 3.5, a new transaction T_6 arrives with the itemset $\{a : 3\}$. If the user requests the mining process after T_6 arrives with the same minimum utility threshold, we observe that the only itemset that could be affected is itemset $\{a\}$. Property 1 shows the explanation of the observation.

Property 1. (Potential EHUI changes after a transaction is added) Let X be an itemset in the newly added transaction T. Since X can only contribute to the utility value in the subset of X, only the subset of X could have utility changes.

The optimization relies on Property 1 and it provides the possibility that we incrementally mine the itemsets and only focus on the transaction differences between the mining process. Here is how the optimization works. When the user requests a mining process, a list of

high utility patterns will be output as the result. During the subsequent mining process, only modified EULs would require re-mining. The previous output can be re-used if the high utility pattern is not the subset of the modified EULs. This optimization reduces the computation overhead of data stream mining and reduce the response time. The mining performance is related to the data differences between the current mining and the previous mining. The more frequent the mining process happens, the faster the response time since most of the existing EULs have remained unchanged. On the contrary, the less frequent the mining process happens, the longer the response time.

3.2.2 Mining under Data Deletion Scenarios

Recall from the introduction section that the characteristics of data stream are continuous and unbound. The historical record would be less relevant as time passes. Our algorithm deprecates or removes the old records without disrupting the existing results.

Example 3.2.5. (EHUI Delete Record Example) In Figure 3.6, the transaction T_1 is marked as deprecated and to be removed. Our Enhanced-HUI-Mining algorithm is notified of the changes and remove all the T_1 in all the existing EULs. The SumUtility, SumRemainingUtility and RUtilityMap of each EUL are updated accordingly when the deletion process is completed. In the example, EUL-f is removed due to no record is left after the delete record process.

In Algorithm 3, we demonstrate the pseudocode of removing records in our Enhanced-HUI-Streaming algorithm. The notations used in the algorithm are listed below.

- **EULs**: All existing Enhanced Utility Lists
- **TransactionsToBeRemoved**: The list of transactions to be removed.

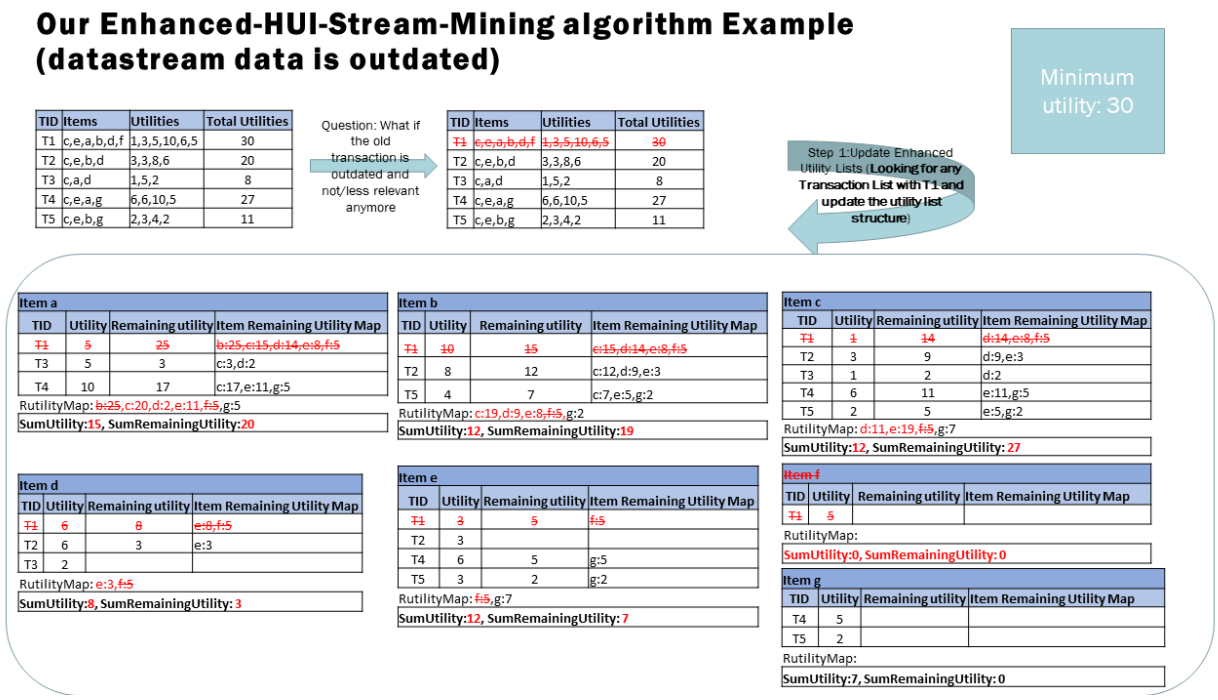

Figure 3.6: Enhanced-HUI-Stream-Mining-algorithm-delete-record

- **removeRecordIfExist()**: This is the function that takes in EUL and TransactionsToBeRemoved as the parameters. The goal of the function is to remove any record in the EUL that is satisfied the criteria.

- **destroy()**: This is the function that de-allocates the resource that was occupied by an EUL. We only invoke this function when the EUL is empty.

Algorithm 3 Enhanced HUI-Stream Miner - Delete record

Require: ULs, $TransactionsToBeRemoved$

1: **for** $UL \leftarrow ULs$ **do**
   $removeRecordIfExist(UL, TransactionsToBeRemoved)$ **if** $UL = \emptyset$ **then** $destroy(UL)$
3: **end if**
4: **end for**

3.2.2.1 Optimizations for the next mining after data deletion

Similar to the optimizations of the next mining after data addition, the optimization for the next mining after data deletion relies on the transaction differences between two consecutive mining processes by Property 3.6. However, there is a major difference between the changes of EHUI patterns after data addition versus data deletion. In the data addition scenario, the latest high utility patterns are the superset of the previous high utility patterns because all the previous high utility patterns remain and only new patterns could be added. Whereas in the data deletion scenario, the latest high utility patterns are the subset of the previous high utility patterns because no new patterns are added to the previous high utility patterns and a subset of the pattern could be removed due to transaction reduction.

Property 2. (Potential EHUI changes after a transaction is deleted) Let X be an itemset in the recent removed transaction T. Since the itemset X can only contribute to the utility value in the subset of the itemset X, only the subset of X could have utility changes.

3.2.3 Mining under Dynamic Pricing Scenarios

Commonly, the pricing for items could be fluctuating over time. The changes would have an impact on utility and eventually affect the result of the high utility itemsets. Our algorithm needs to be able to handle this scenario efficiently without rescanning the data streams. We demonstrate the way we deal with utility changes using our algorithm.

Example 3.2.6. (EHUI Price Change Example) In Figure 3.7, assuming the utility of item c in the transaction T_1 is changed from having the utility of 1 to the utility of 2. Our enhanced-HUI-Stream-Mining algorithm updates the internal model to reflect the latest update. The rule is to look for any T_1 in EULs, and update any T_1 that contains the item c.

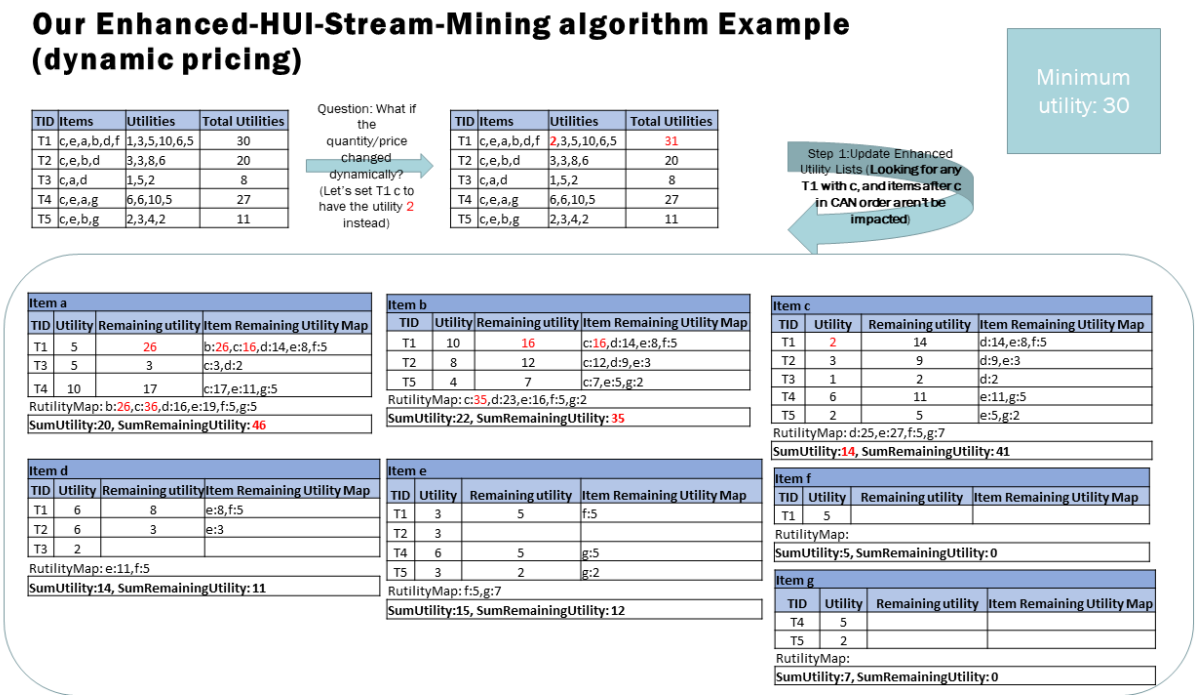

Figure 3.7: Enhanced-HUI-Stream-Mining-algorithm-Dynamic-Pricing

In the item remaining utility map, all the items before the item c will be updated based on the canonical order. In this case, the item a and the item b are required to be updated in T_1 if they present. For example, the item c exists in EUL-a's T_1. The original Item Remaining Utility Map for T_1 in EUL-a is $\{b : 25, c : 15, d : 14, c : 8, f : 5\}$, so the Item Remaining Utility Map will be updated to $\{b : 26, c : 16, d : 14, c : 8, f : 5\}$.

Our EUL data structure preserves all the transaction id information. When the price needs to be updated, there is no need to re-scan the database / data stream. The associated changes would be reflected in the data structure precisely and efficiently.

Algorithm 4 lists the pseudo-code required to update the EULs when the utility information changes. The notations used by the pseudocode are listed as follow.

- **EULs**: The existing potential Enhanced Utility Lists

- **UtilityUpdateMeta**: The input information to differentiate the update scope. It could be ranging from a single item utility changes or by a rule to select a list of items
- **locateTIDToUpdate**: The function that takes UtilityUpdateMeta and EUL as input parameters and update UL based on the UtilityUpdateMeta. UtilityUpdateMeta provides the information about the TID to be updated and what the new value is, and the function will update the information in the EUL accordingly
- **TIDs** The transaction IDs that are updated based on the UtilityUpdateMeta

Algorithm 4 Enhanced HUI-Stream Miner - Utility Changes

Require: $ULs, UtilityUpdateMeta$
1: **for** $UL \leftarrow ULs$ **do**
2: $TIDs \leftarrow locateTIDToUpdate(UL, UtilityUpdateMeta)$
3: **end for**

3.2.3.1 Optimizations for the next mining under dynamic pricing scenarios

When the price changes, multiple transactions could be under the impact. The data structure of EULs minimizes the impact of price changes. If there are no changes to the existing utility but the changes only apply to newly added records, the optimization would be the same as the optimization when handling data addition. If the previous price needs to be updated, the item whose utility is changed and the item whose remaining utility is changed are modified and require re-mining.

Property 3. (Potential EHUI changes after item unit utility is changed) Let x be an item in a transaction database D and the associated unit utility is denoted as x_u. If x_u is changed and the previous record does not require an update, Property 1 is applied. If

x_u is changed and the previous record required update, any transaction contains x required update and the remaining utility for items before x are required update.

Chapter 4

Evaluation

In this chapter, we evaluate our proposed algorithm by several experiments. In Table 4.1, we listed out the baselines used by the experiments. Since there is no strictly feature equivalent HUI stream mining algorithm exists to be the baseline, we will use different algorithms to test its batch mining performance, incremental mining performance and stream performance. We will also include an experiment to test the dynamic pricing performance of our EHUI-stream algorithm. For batch processing performance, we compare against EFIM algorithm. For the incremental mining performance, we compare against EIHI algorithm, and for the stream mining performance, we compare against HUPMS. Finally, we would test the response time and the runtime performance as it is the unique feature of our purposed algorithm when dealing with price changes. Additionally, we measure the scalability of our purposed algorithm in terms of the size of the data.

Table 4.1: Expected baseline feature comparison

Feature	EFIM	EIHI	HUPMS	Our algorithm
Support Batch Mining	✓			✓
Support Incremental Mining		✓		✓
Support Stream Mining			✓	✓
Mine Efficiently after data is changed				✓

4.1 Experimental Setup

- Real-world datasets from the UCI machine learning repository [DG17] and SPMF open-source library [FGG+14]
- Desktop Computer using Ubuntu 20.04 with Core i7 6700, 32Gb of RAM for designing and developing the algorithm
- All the implementations were designed to be runnable using SPMF java library

4.2 Experimental Evaluation

In this section, we discuss the experimental evaluation of our proposed algorithm. We use Foodmart, Retail, Mushroom and BMS datasets for evaluation. We evaluated the dataset characteristics and place a summary of that information in Table 4.2. Noted that Foodmart and Retail, BMS are sparse datasets, and mushroom dataset is a dense dataset.

4.2.1 Batch Processing Performance Evaluation

Despite our EHUI-stream algorithm is a stream oriented algorithm, the purpose of batch processing performance evaluation is that we would like to evaluate how efficient our EHUI-stream algorithm compared to the best HUI static algorithm. However, HUI static algorithm

Table 4.2: Dataset characteristic

Dataset	**Foodmart**	**Retail**	**Mushroom**	**BMS**
Density	sparse	sparse	dense	sparse
Total Number of Transactions	4,141	88,162	8,124	59,602
Longest Length of Transaction	14	76	23	267
Shortest Length of Transaction	1	1	23	1
Average Length of Transaction	4	10	23	2
Largest Single Item Utility	2,166	140	100	9,000
Smallest Single Item Utility	50	1	1	0
Average Single Item Utility	655	16	18	724
Largest Transaction Utility	9,180	1,491	757	180,357
Smallest Transaction Utility	106	1	212	0
Average Transaction Utility	2,900	169	420	1,819
Total Transaction Utility	12,011,023	14,910,915	3,413,720	108,457,438

scan through the dataset and perform some optimization beforehand such as re-ordering itemsets by TWU ascending order or merging duplicate records for dense dataset. Since stream algorithm does not have the advantage of pre-processing the entire datasets, we would disable the pre-processing advantage of the batch processing algorithm to perform the batch experiments.

Experiment 4.2.1. In this experiment, we used Foodmart dataset for evaluation. For this dataset, we varied the minimum utility threshold to be in the range of 1000 to 5000 with the step size of 1000 to test the runtime performance of our proposed algorithm in comparison with EFIM. The minimum utility value depends on the characteristic of the dataset. A good minimum utility threshold should not produce an overwhelming amount of patterns. If the minimum utility is too low, all the possible enumeration would be output as the result. If the minimum utility is too high, there will be no high utility itemsets. Since Enhanced-HUI-Stream is a data-streaming algorithm, we adjusted the windows size to be the same as the size of the dataset to perform batch processing operation. In Figures 4.1 and 4.2, we

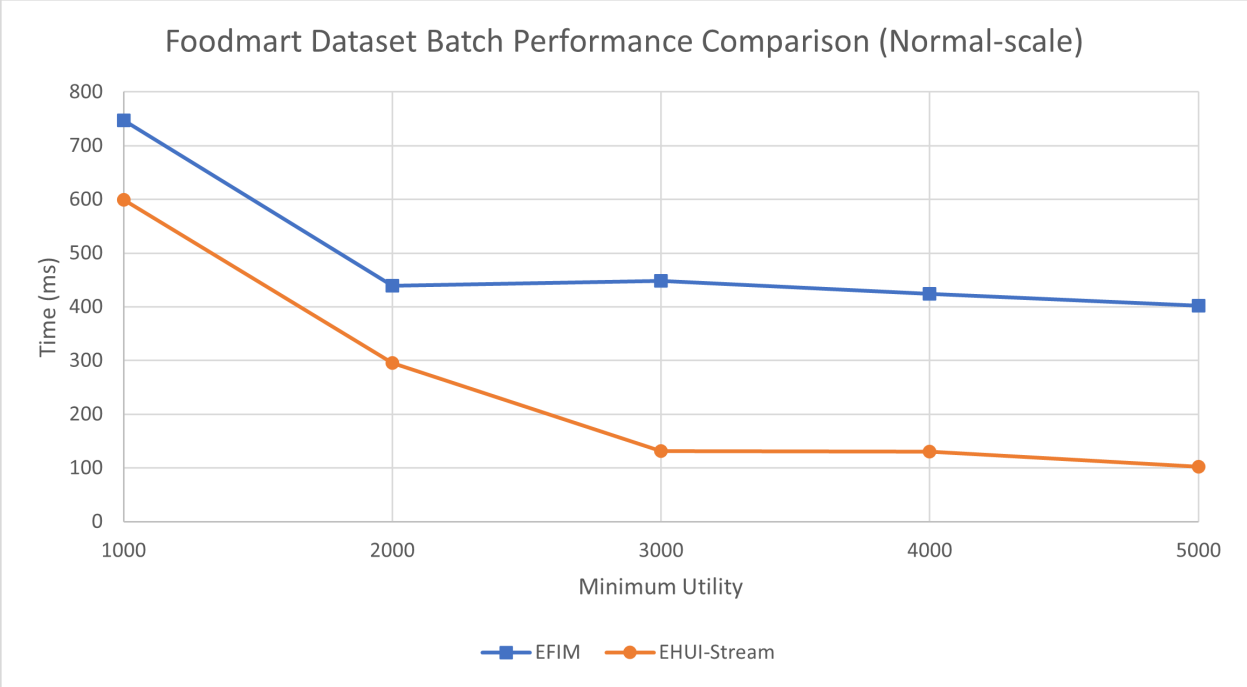

Figure 4.1: Batch runtime performance comparison using the foodmart dataset

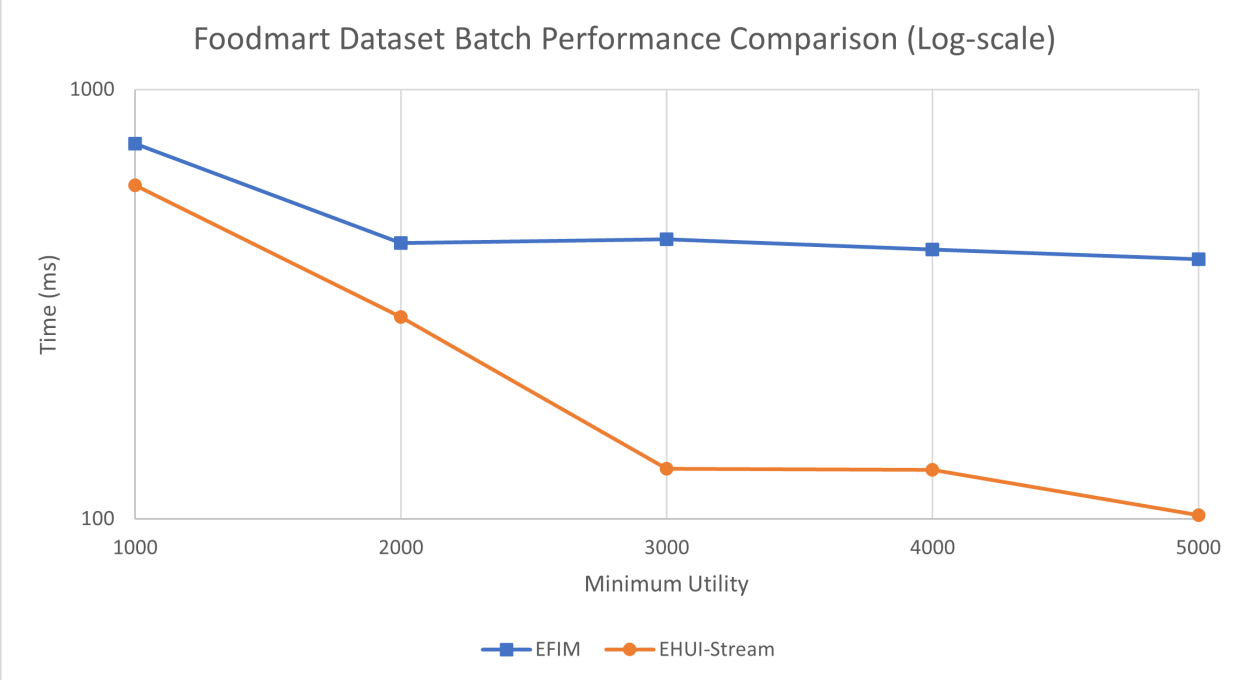

Figure 4.2: Batch runtime performance comparison using the foodmart dataset (log-scale)

show the runtime performance comparison. In Figure 4.3, we show the memory consumption graph. We observe that the runtime performance of EHUI-stream is faster than EFIM and with lower memory consumption. For our EHUI-stream algorithm, the execution time is within the range of 100 to 600ms, and the execution time of EFIM is within the range of 400 to 747ms. Our EHUI-stream memory consumption is also lower than EFIM. During the experiment, our EHUI-stream has memory consumption around 60 MB for various minimum utility, and EFIM has memory consumption from 60 to 100 depending on the minimum utility.

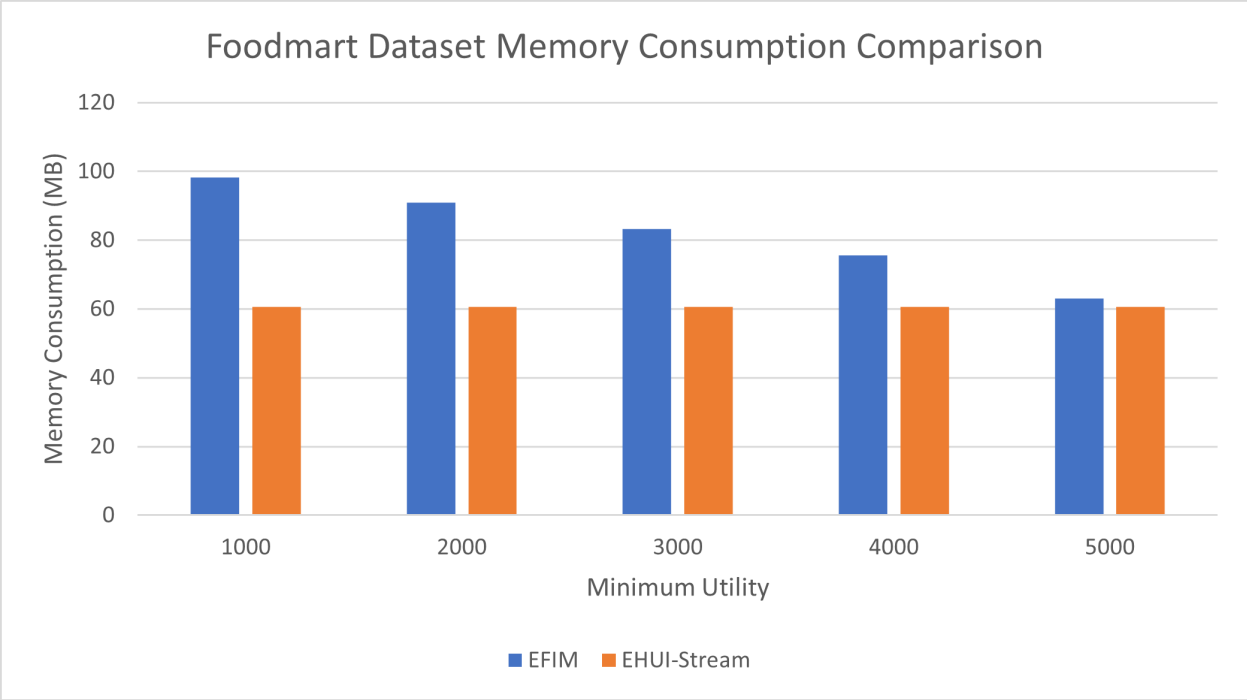

Figure 4.3: Batch memory consumption comparison using the foodmart dataset

Experiment 4.2.2. In this experiment, we use Retail dataset for evaluation to compare our algorithm against EFIM. We set the minimum utility threshold within the range of 30000 to 80000 with the step size of 10000. In Figures 4.4 and 4.5, we show the runtime performance comparison. Retail dataset is a more complex dataset than Foodmart not only because

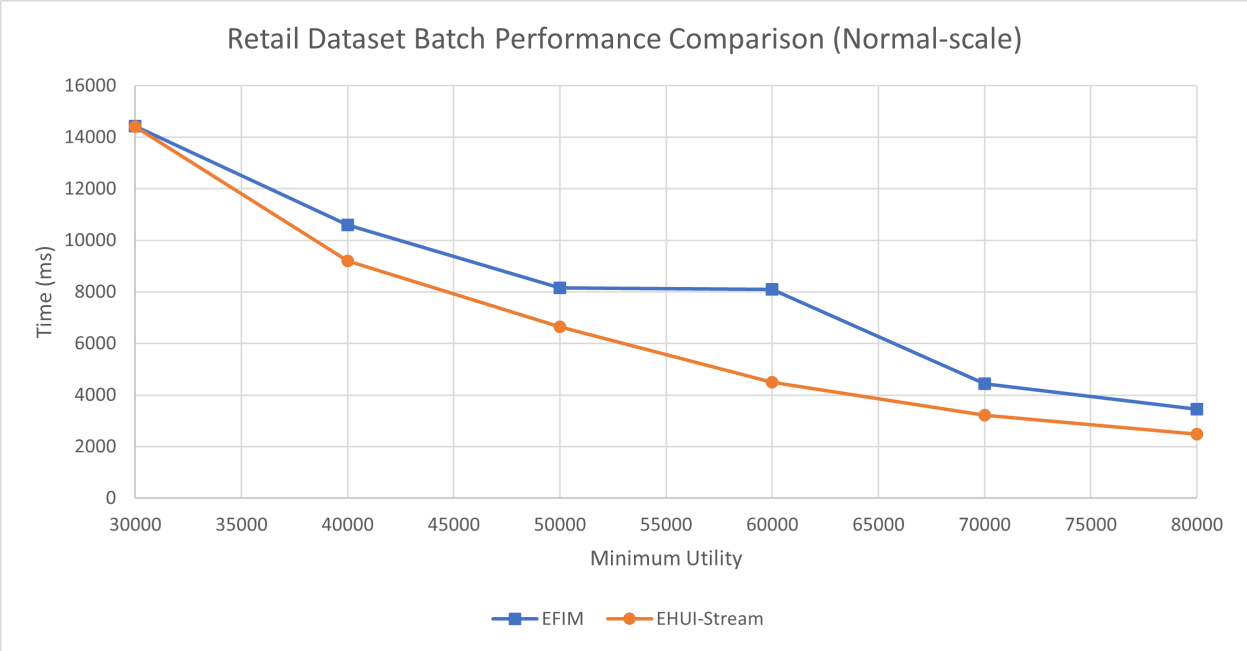

Figure 4.4: Batch runtime performance comparison using the retail dataset

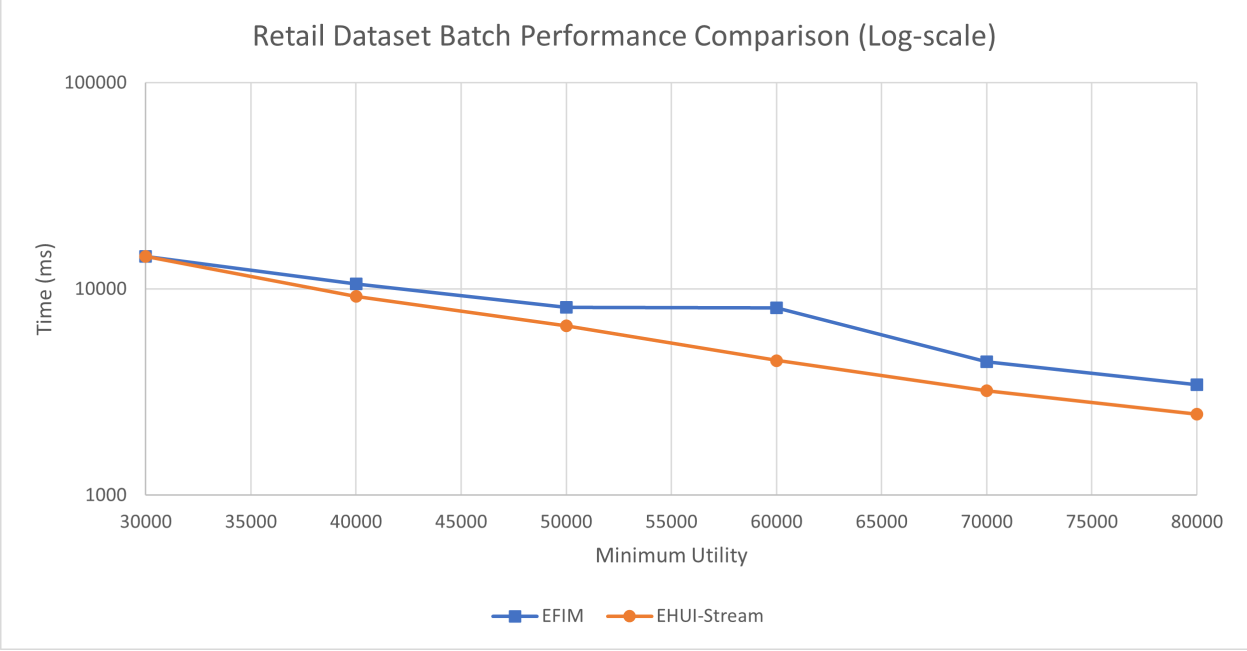

Figure 4.5: Batch runtime performance comparison using the retail dataset (log-scale)

of the size of the data is larger, but because of the amount of long transactions is much higher. In Figure 4.6, we show the memory consumption. Our EHUI-stream algorithm has similar performance as EFIM, but our EHUI-stream performs slightly better than EFIM. An interesting observation is that EFIM has almost identical performance as our EHUI-stream when the minimum utility is 30000 but has slightly worse performance compared to our EHUI-stream for the rest of the utility. As for the memory consumption, both EFIM and EHUI-stream performs fairly even and sometimes EFIM outperforms our EHUI-stream when the minimum utility is around 60000 to 80000 and our EHUI-stream outperforms EFIM when the minimum utility is around 30000 to 40000. Overall, it is even match-up result comparing EFIM and EHUI-stream.

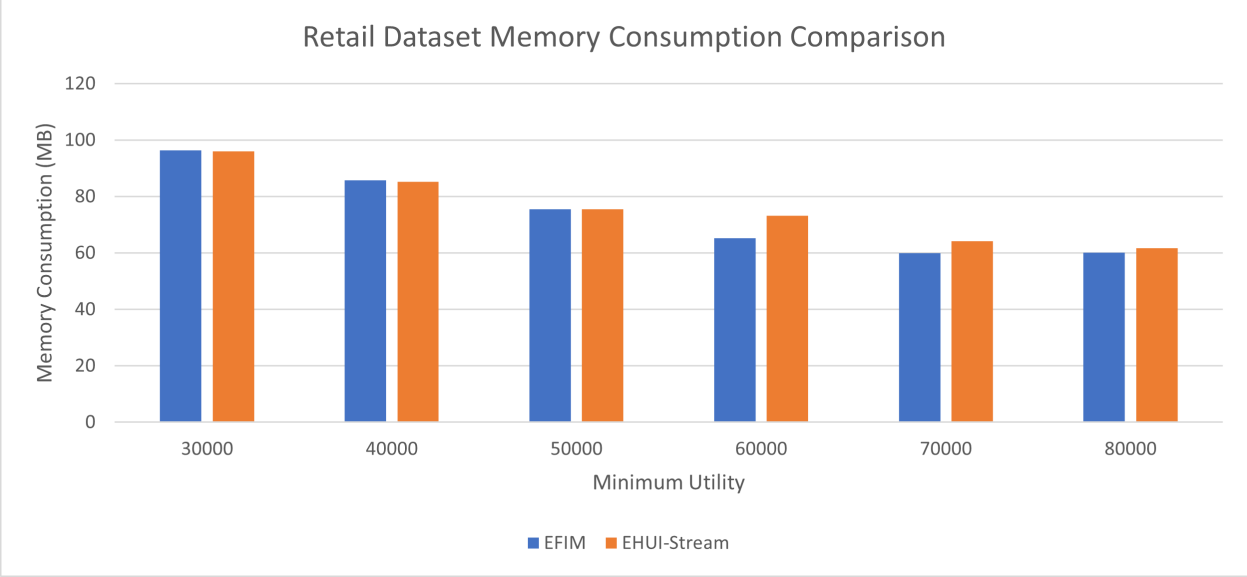

Figure 4.6: Batch memory consumption comparison using the retail dataset

Experiment 4.2.3. In this experiment, we use BMS dataset for evaluation. In Figures 4.7 and 4.8, we show the runtime performance in normal scale and in log scale respectively. In Figure 4.9, we show the memory consumption. For this dataset, our EHUI-stream algorithm

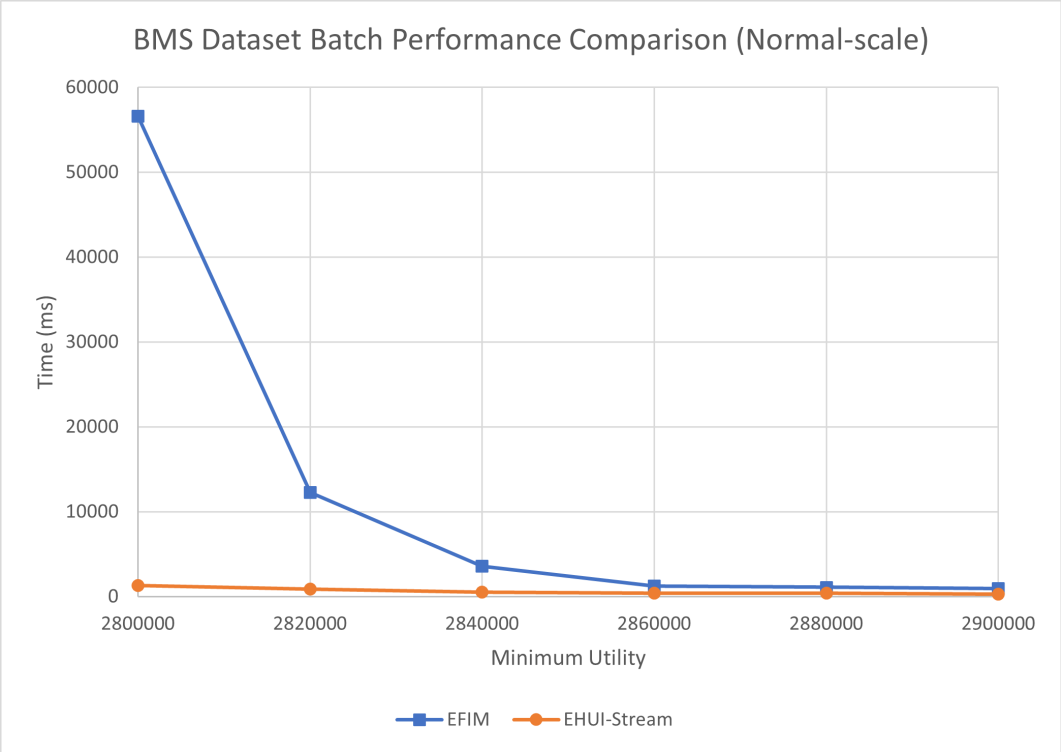

Figure 4.7: Batch runtime performance comparison using the BMS dataset

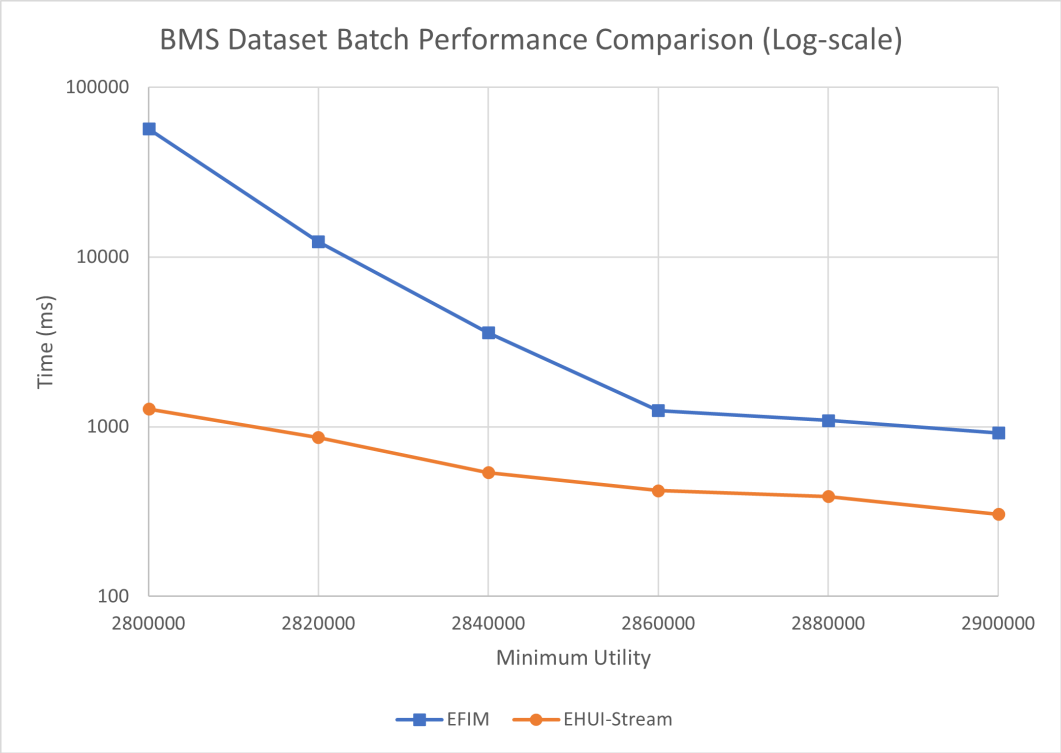

Figure 4.8: Batch runtime performance comparison using the BMS dataset (log-scale)

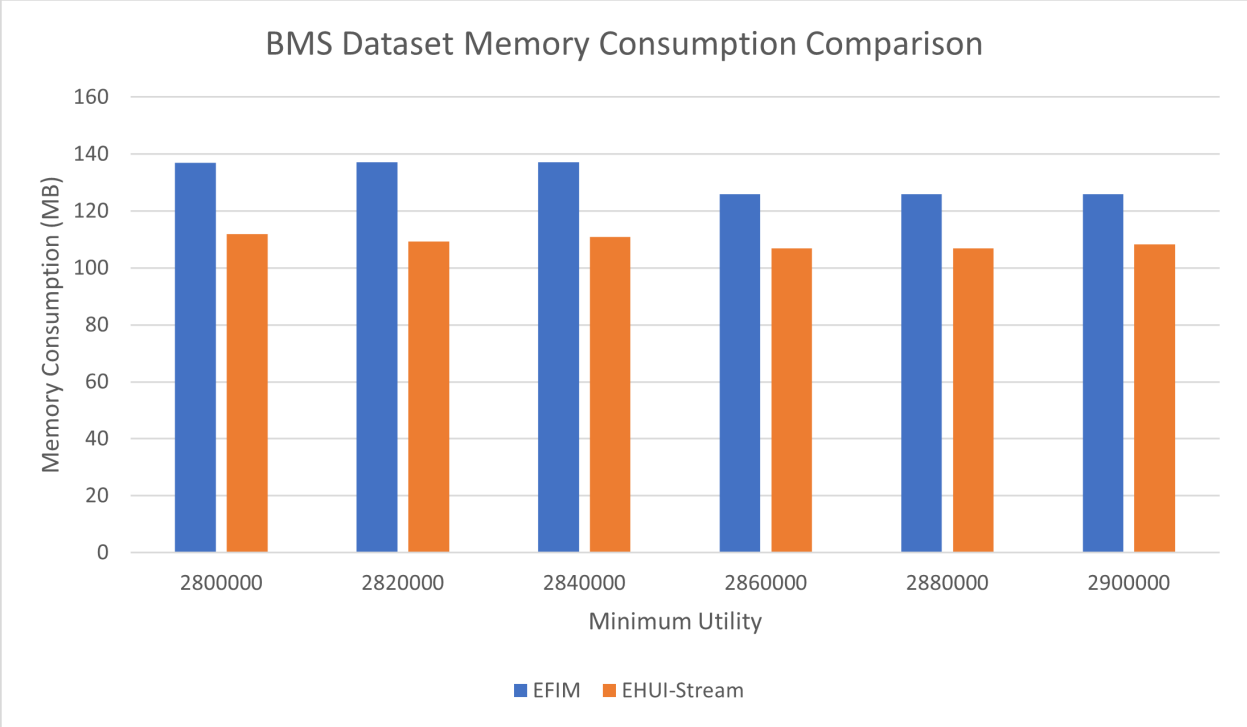

Figure 4.9: Batch memory consumption comparison using the BMS dataset

outperforms EFIM for both runtime performance and memory consumption. The difference is large especially when the minimum utility is less than 2840000.

Experiment 4.2.4. In this experiment, we use Mushroom dataset for evaluation. This is a dense dataset which is different from the previous sparse datasets. In Figures 4.10 and 4.11, we show the runtime performance comparison; in Figure 4.12, we show the memory consumption comparison. We observe that our EHUI-stream has slightly better runtime performance compared to EFIM, and the memory consumption is much lower compared to EFIM.

In **summary**, batch processing performance is not our EHUI-stream mining strength. Behind the scene, our algorithm processes each transaction whenever it is arrived, while the other algorithms treat the entire dataset as a whole and perform optimizations when applica-

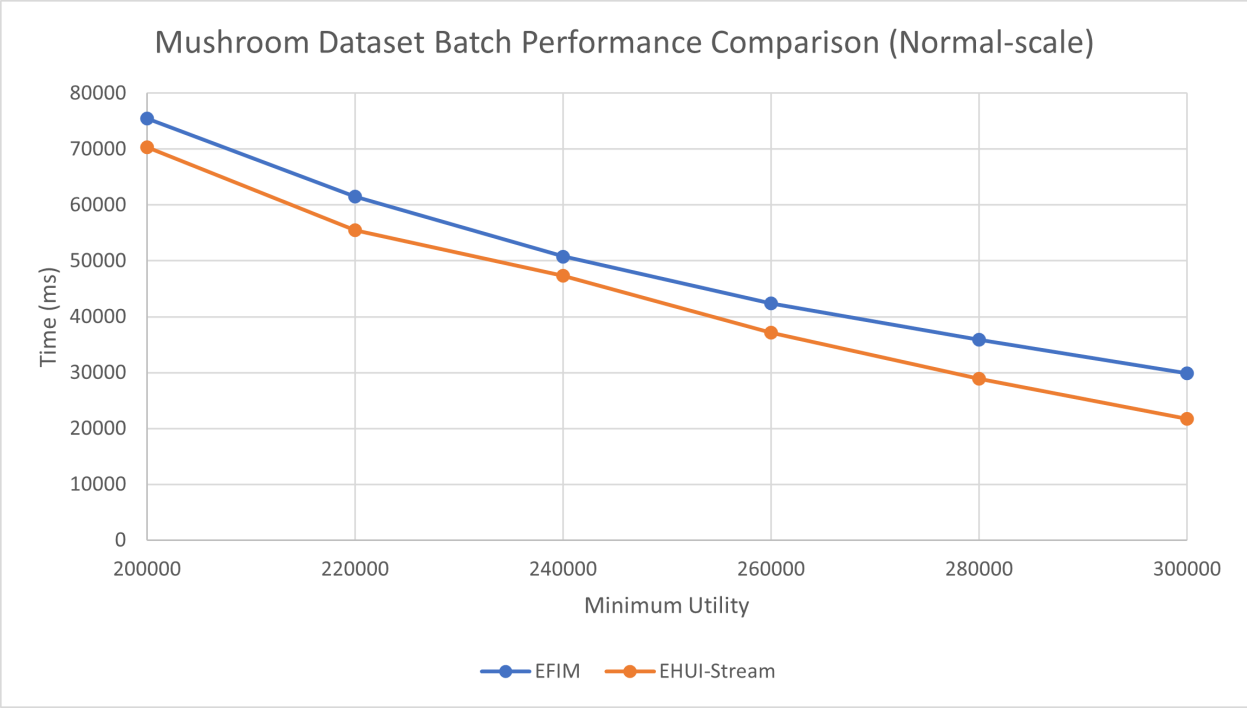

Figure 4.10: Batch runtime performance comparison using the mushroom dataset

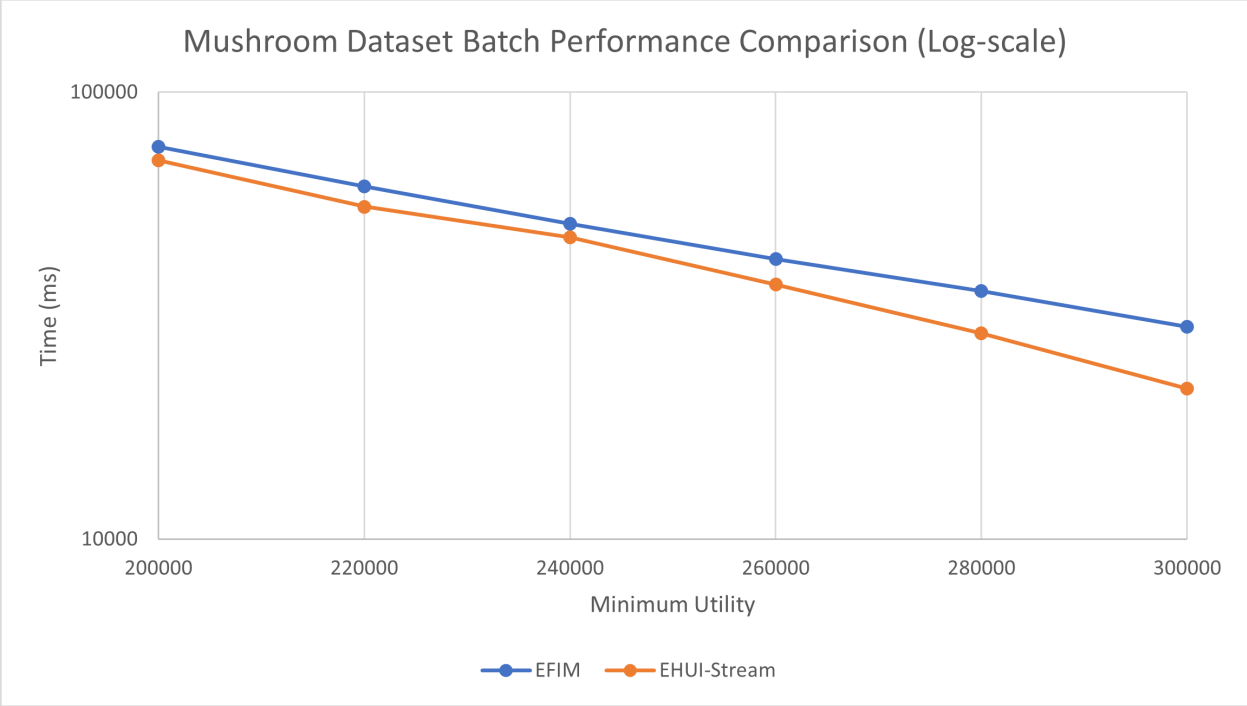

Figure 4.11: Batch runtime performance comparison using the mushroom dataset (log-scale)

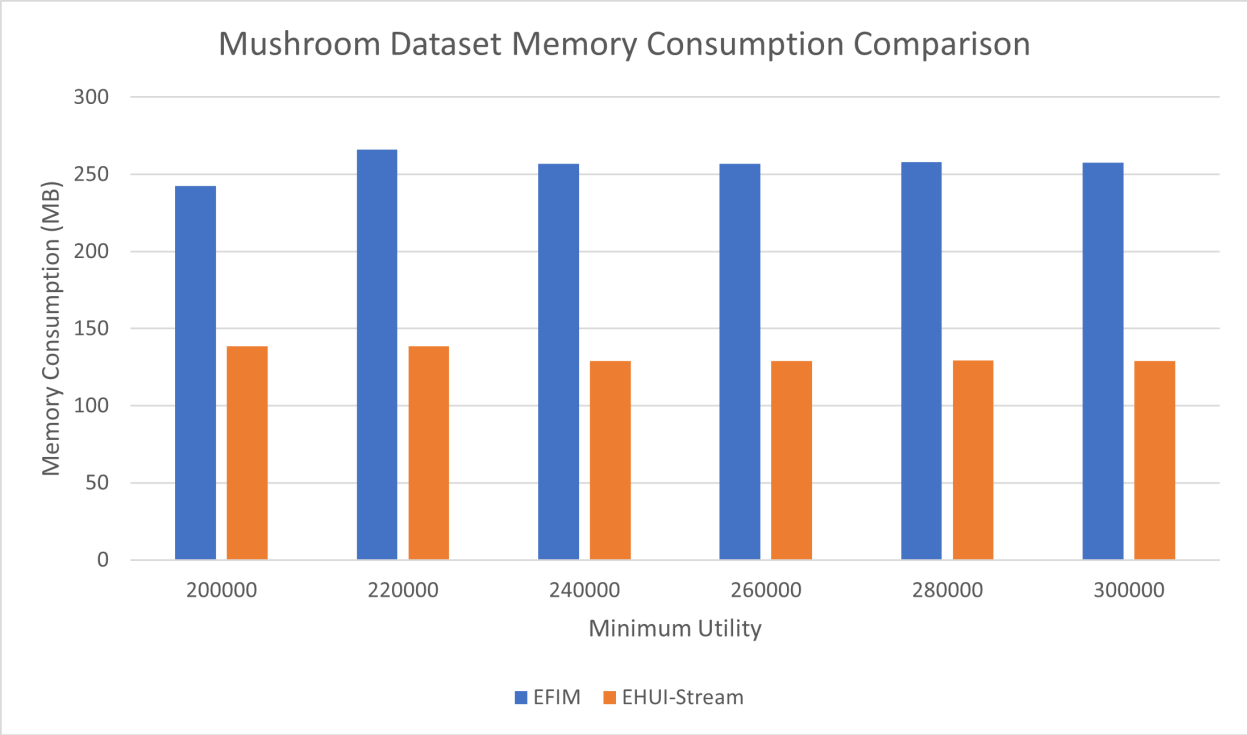

Figure 4.12: Batch memory consumption comparison using the mushroom dataset

ble. The main disadvantage of our algorithm compared to other static algorithm is the lack of database reduction feature. Database reduction feature is especially useful when dealing with dense datasets. For example, when processing mushroom dataset, EFIM database reduction feature would significantly reduce the size of the dataset which is way more efficient than any other algorithms, and stream mining algorithm simply cannot perform that level of optimization due to the nature of processing when arrived. We compared sparse datasets in this experiment section so to be fair. Our EHUI-stream mining algorithm can be used in batch processing and has relatively good performance compared to other static algorithm, but it may not be the best depending on the characteristic of the dataset.

4.2.2 Experiments on Data Stream Mining Response Time

Experiment 4.2.5. In this experiment, we test the response time of our purposed EHUI-stream algorithm compared to EIHI algorithm using Foodmart dataset. We set the minimum utility threshold for both algorithm be 3250. In Figure 4.13, we show the experimental result. There are 4141 transactions in Foodmart datatset, EHUI-stream achieves almost 0 ms response time for almost 3000 transactions. Most EIHI response time is within the range of 1 to 2ms. No transaction takes more than 4ms to mine for both algorithms. Our EHUI-stream outperforms EIHI in this experiment.

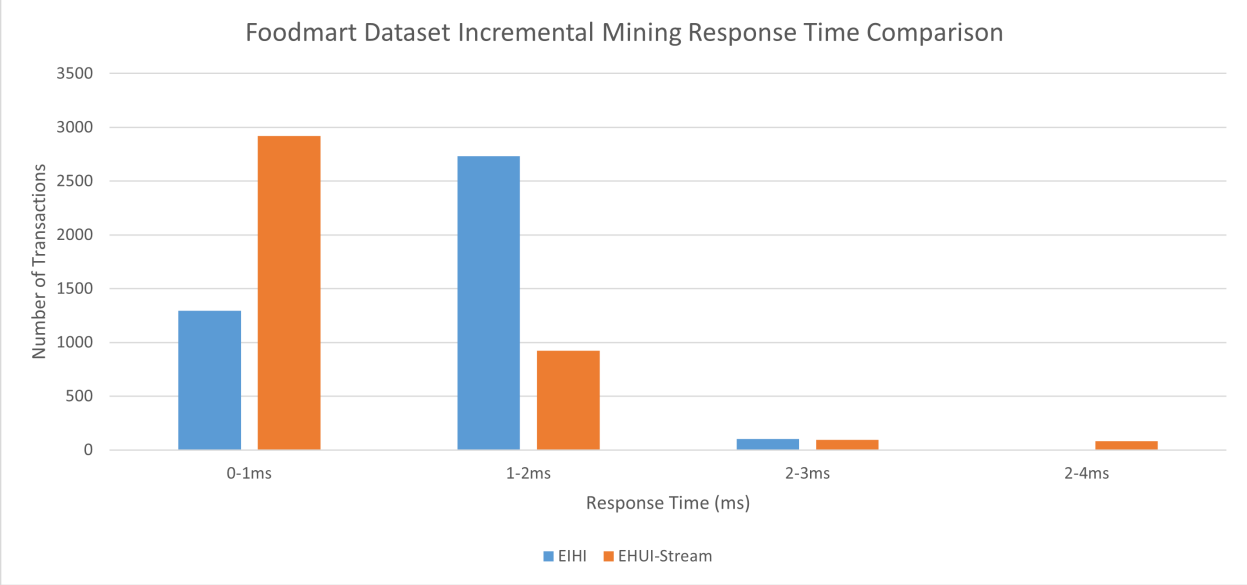

Figure 4.13: Incremental response time comparison using the foodmart dataset

Experiment 4.2.6. In this experiment, we test the response time of our purposed EHUI-stream algorithm compared to EIHI algorithm using Retail dataset. We set the minimum utility threshold for both algorithm be 80000. In Figure 4.14, we show the experimental result. In this dataset, our EHUI-stream contains about 20000 transactions that were mined within 1ms. EHUI-stream has less items to be mined more than 4ms. Overall, our algorithm

is faster in terms of the response time.

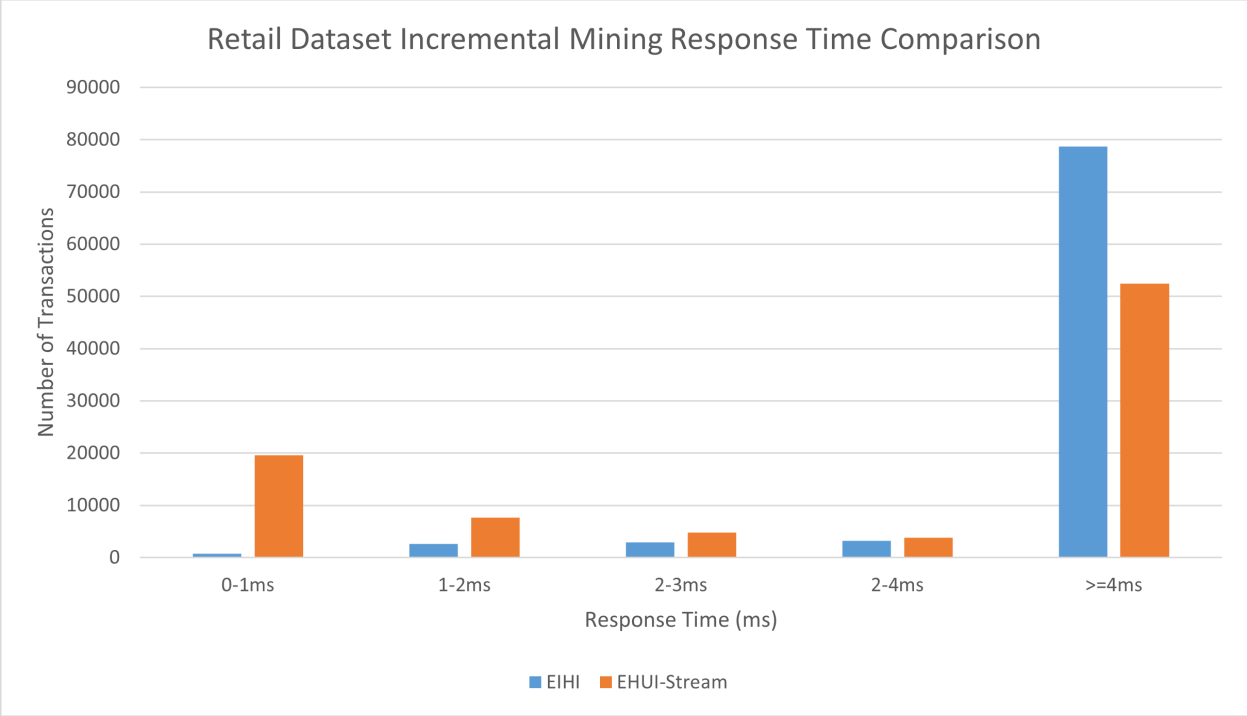

Figure 4.14: Incremental response time comparison using the retail dataset

Experiment 4.2.7. In this experiment, we test the response time of our purposed EHUI-stream algorithm compared to EIHI algorithm using mushroom dataset. In Figure 4.15, we show the experimental result. This is the dense dataset that is challenging to to mine for both EIHI and our EHUI-stream incrementally, so we reduced the sample size to be 2000 instead of the original dataset size of 8124. Our EHUI-stream algorithm has an overall better performance than EIHI in terms of the response time.

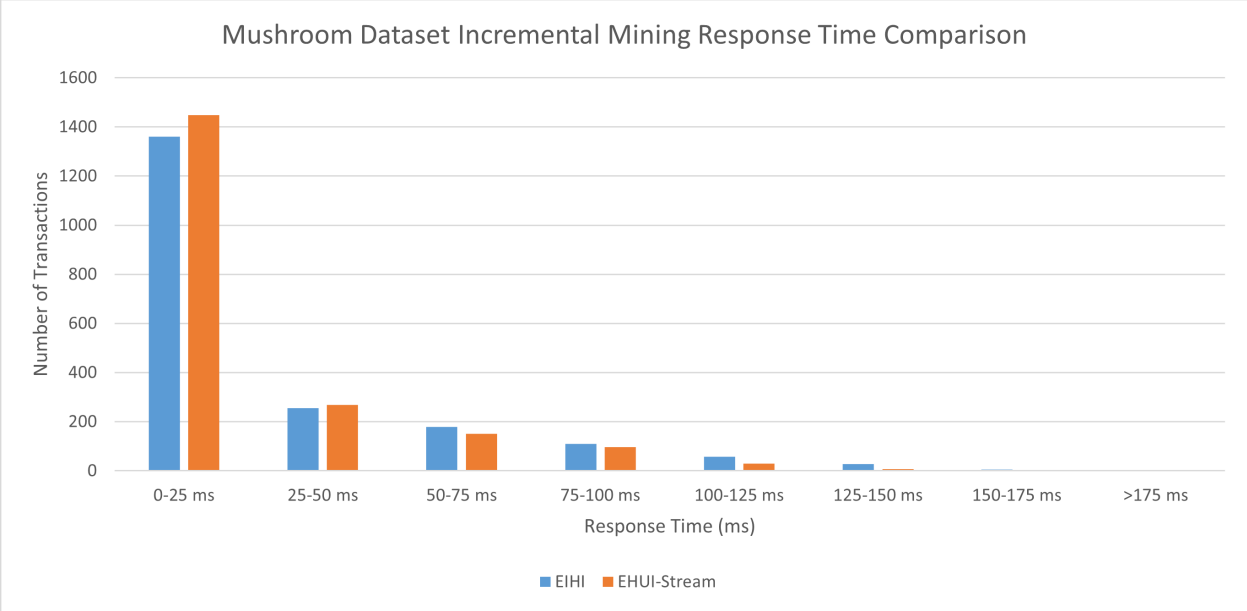

Figure 4.15: Incremental response time comparison using the mushroom dataset

Experiment 4.2.8. In this experiment, we test the response time of our purposed EHUI-stream algorithm compared to EIHI algorithm using BMS dataset. We set the minimum utility threshold be 290000 for EIHI and our EHUI-stream algorithm. In Figure 4.16, we show the experimental result. In this experiment, our EHUI-stream greatly outperforms EIHI in terms of the response time. Most of the transaction is processed within 1 ms for EHUI-stream while EIHI takes more than 4ms to process the majority of the transactions.

In **summary**, our EHUI-stream mining has impressive incremental mining performance

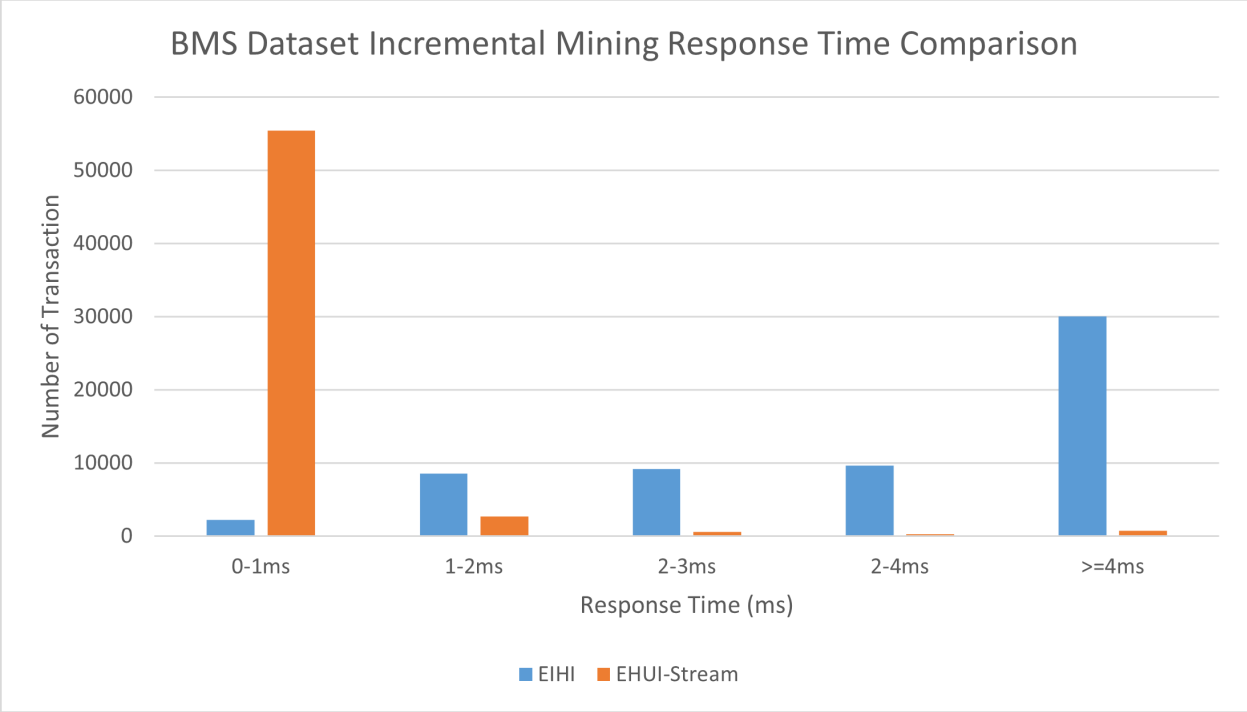

Figure 4.16: Incremental response time comparison using the BMS dataset

compared to EIHI. To the best of our knowledge, our EHUI-stream algorithm can achieve closed to 0 ms response time when the incremental step is small, which is the best algorithm in the HUI incremental mining domain.

4.2.3 Experiments on Stream Mining Performance

Experiment 4.2.9. In this experiment, we compare our EHUI-Stream algorithm to the baseline algorithm HUPMS [ATJC12], we use foodmart database for experimental purposes. Since this is a relatively small dataset, we adjust the window size to the be same size of the dataset and measure the runtime performance. The other reason is that we discovered stream mining on high utility itemset is dependent on the window function so for the first stream mining experiment we would like to reduce the variance. We adjust the minimum utility threshold be within the range of 4000 to 10000 ($0.03\% - 0.08\%$ of the total minimum

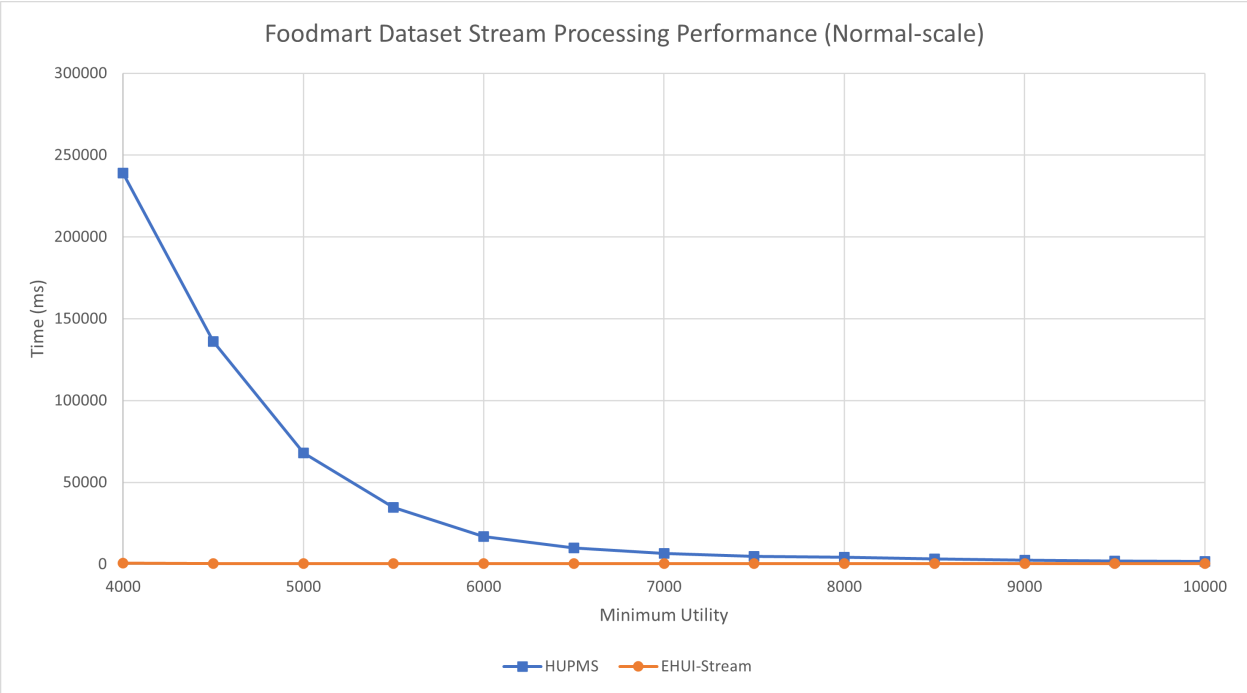

Figure 4.17: The stream mining performance comparison of HUPMS and EHUI-Stream using the foodmart dataset

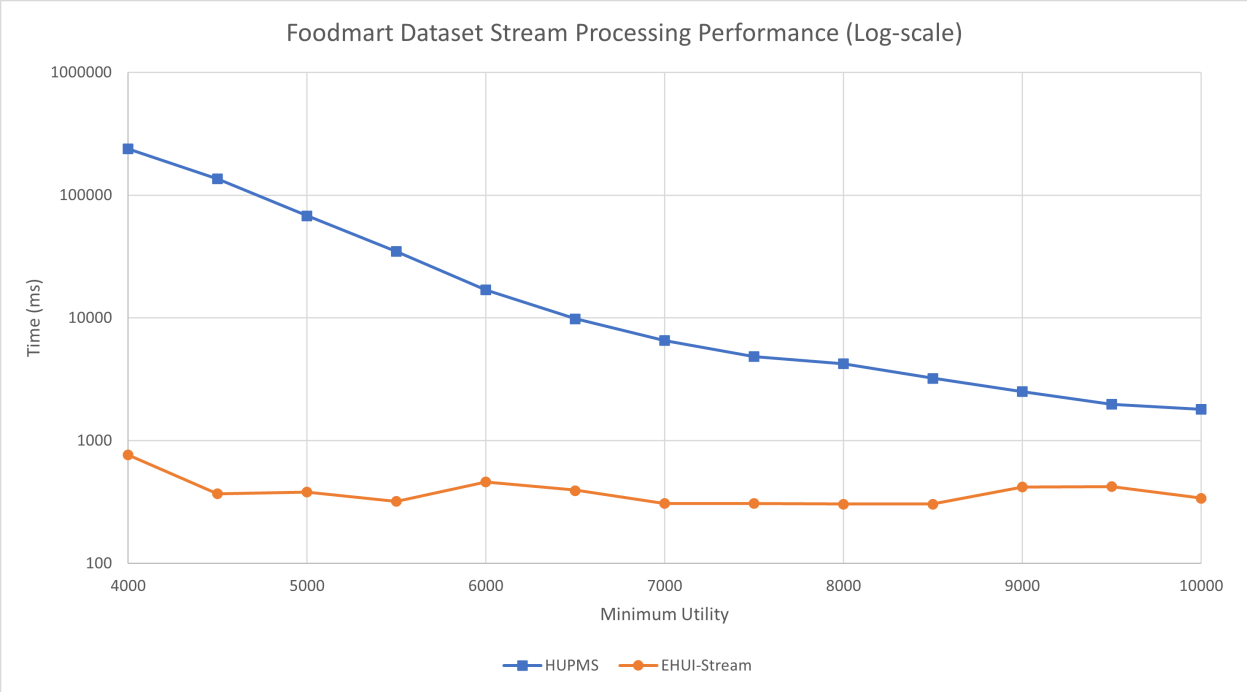

Figure 4.18: The stream mining performance comparison of HUPMS and EHUI-Stream using the foodmart dataset (log-scale graph)

utility threshold). For HUPMS, the number of batch within a window is 1. We show the experimental result in Figures 4.17 and 4.18. We observe that our EHUI-Stream is much faster than the existing HUPMS in terms of the mining performance and it is in a different log-scale. The main drawback of HUPMS is the candidate generation and the database re-scanning which is really time-consuming.

Experiment 4.2.10. In this experiment, we use foodmart database for experimental purposes. We set the batch size be 500 transactions and the window size be 3 batches(1500 transactions). The minimum utility threshold be within the range of 1000 to 2000 with the step size of 125, and the mining process happens after the all the data is consumed and we measure the runtime performance. We show the experimental result in Figures 4.19 and 4.20.

Experiment 4.2.11. In this experiment, we use retail dataset for experimental purposes. For EHUI-stream algorithm, we set the windows size be 20000 transactions and the minimum utility be within the range of 2000 to 20000 with the incremental value of 1000. For HUPMS algorithm, we set the windows size be 20000 and with the number of batch in a window be 4. In Figure 4.21, we show the experimental result, we observe that our EHUI-stream total time is even less than the candidate generation time of HUPMS.

Experiment 4.2.12. In this experiment, we use mushroom dataset for experimental purposes. As the mushroom datatset is a dense dataset, we need to adjust the minimum utility threshold to be higher than the sparse dataset such as retail and foodmart. We set the window size be 2000 and the minimum utility threshold be within the range of 20000 to 40000 with the step size of 1000 and measure the response time.

In **summary**, we demonstrated our EHUI-stream algorithm stream mining performance

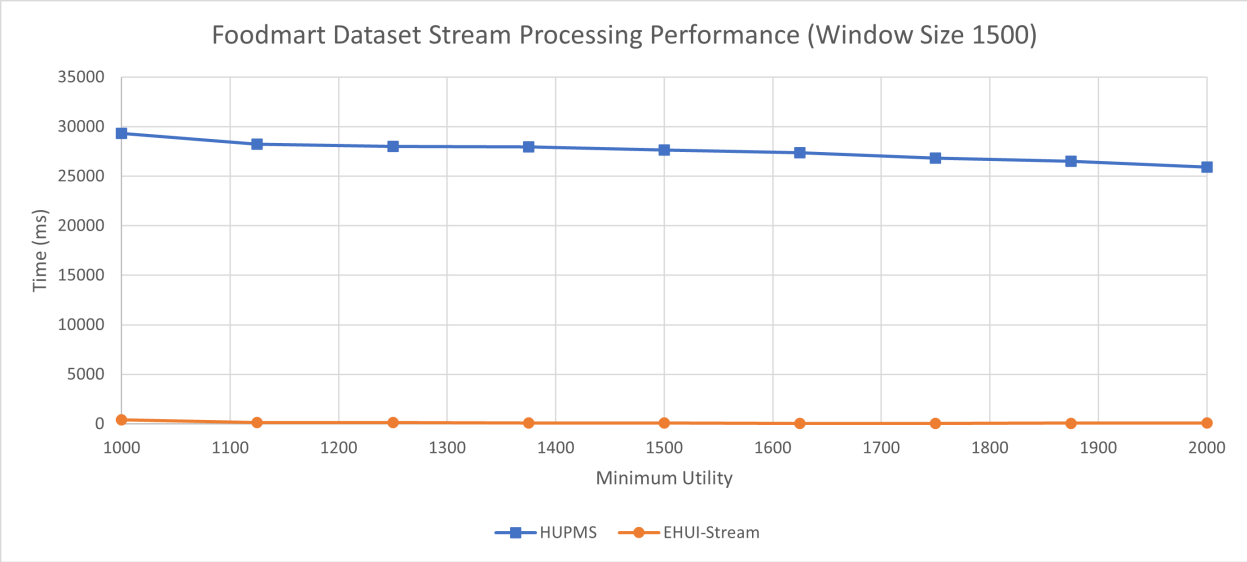

Figure 4.19: The stream mining performance comparison of HUPMS and EHUI-Stream with window size 1500 using the foodmart dataset

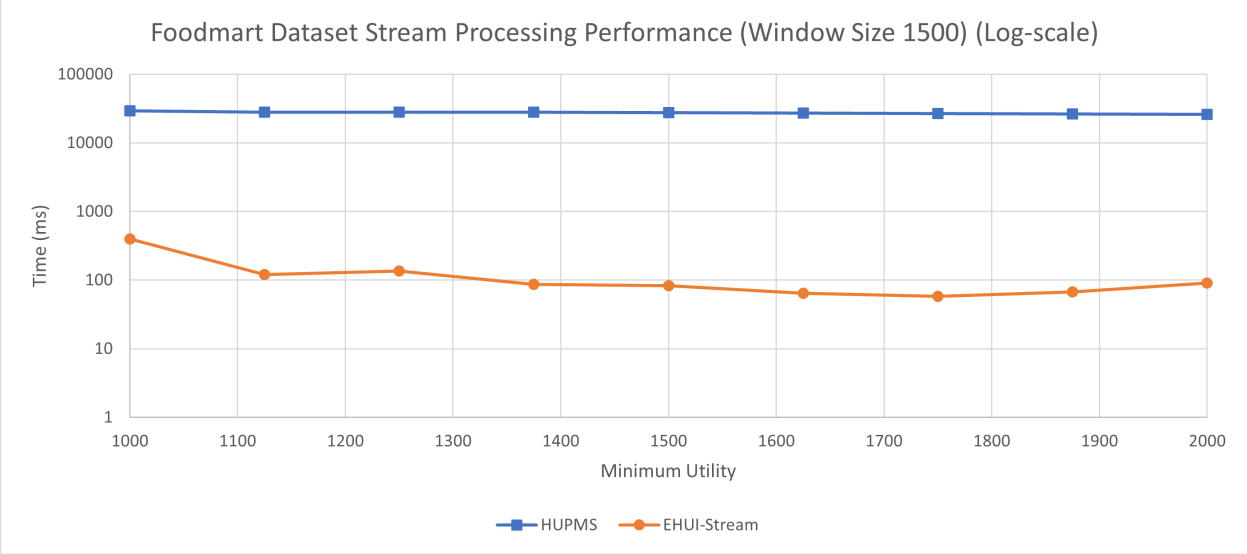

Figure 4.20: The stream mining performance comparison of HUPMS and EHUI-Stream with window size 1500 using the foodmart dataset (log-scale graph)

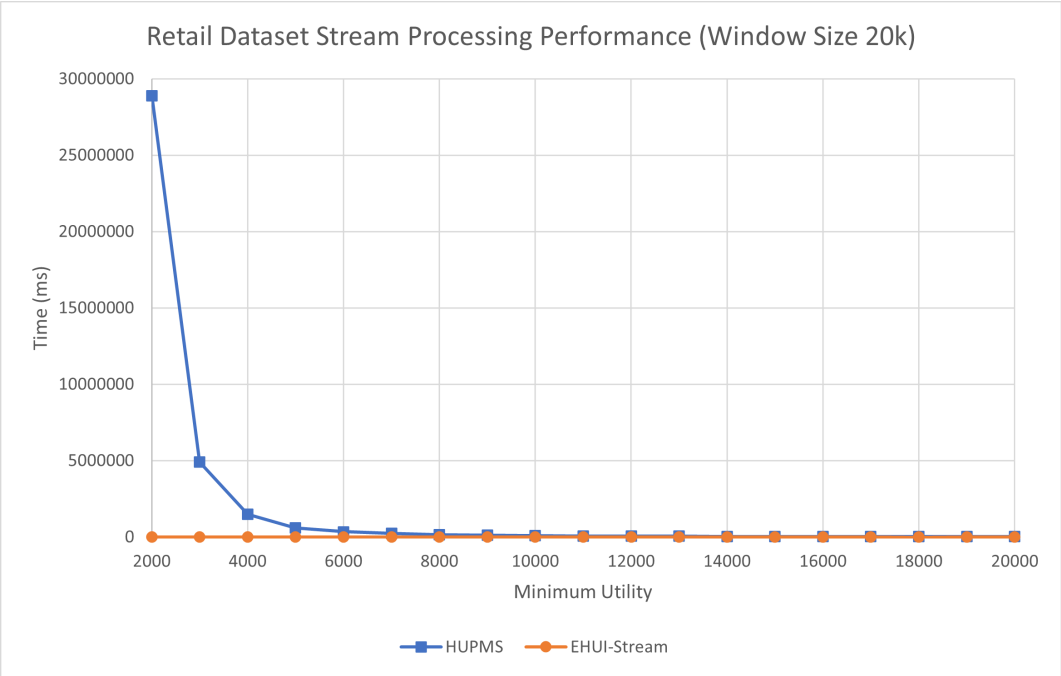

Figure 4.21: The stream mining performance comparison of EHUI-Stream and HUPMS with window size 20k using the retail dataset

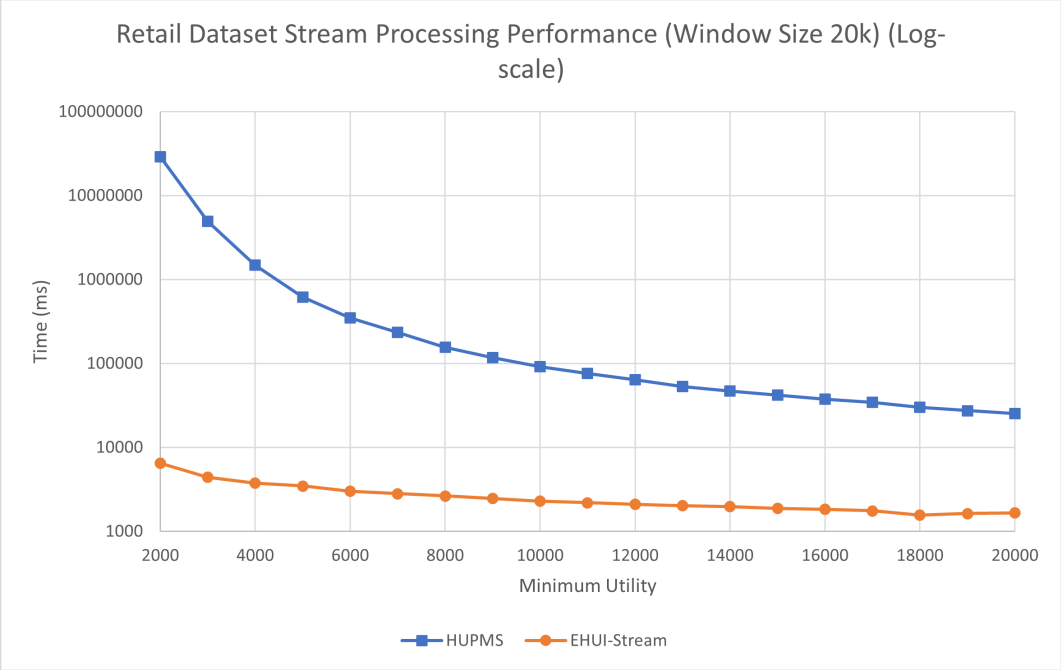

Figure 4.22: The stream mining performance comparison of EHUI-Stream and HUPMS with window size 20k using the retail dataset (log-scale graph)

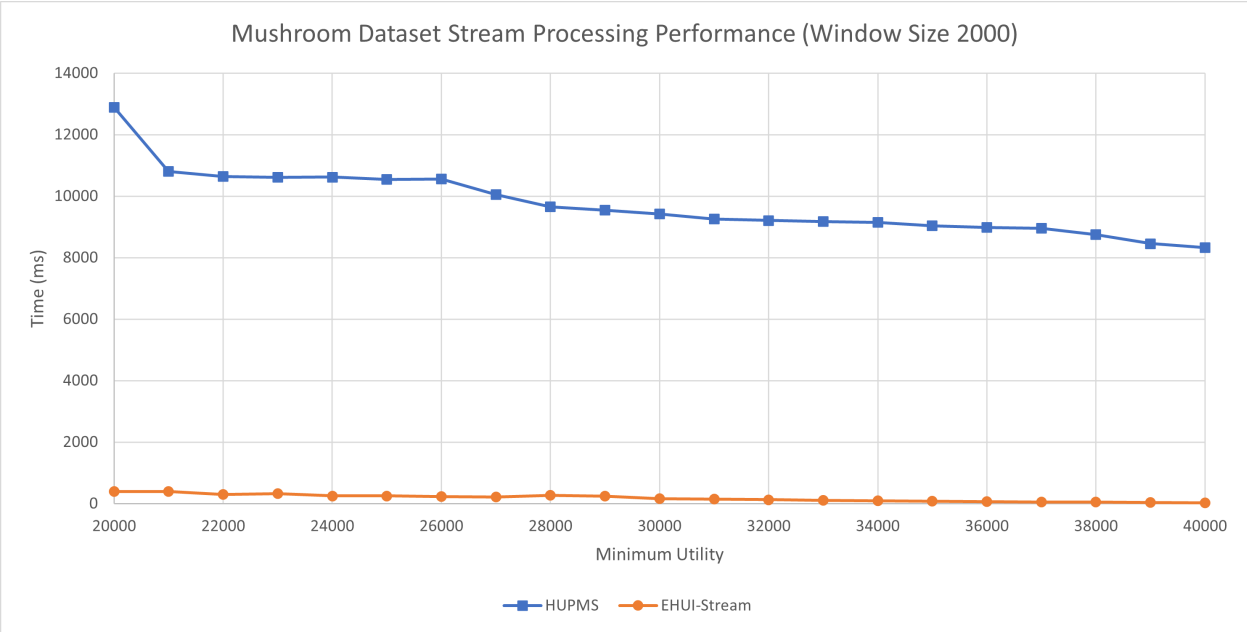

Figure 4.23: The stream mining performance comparison of EHUI-Stream and HUPMS with window size 2k using the mushroom dataset

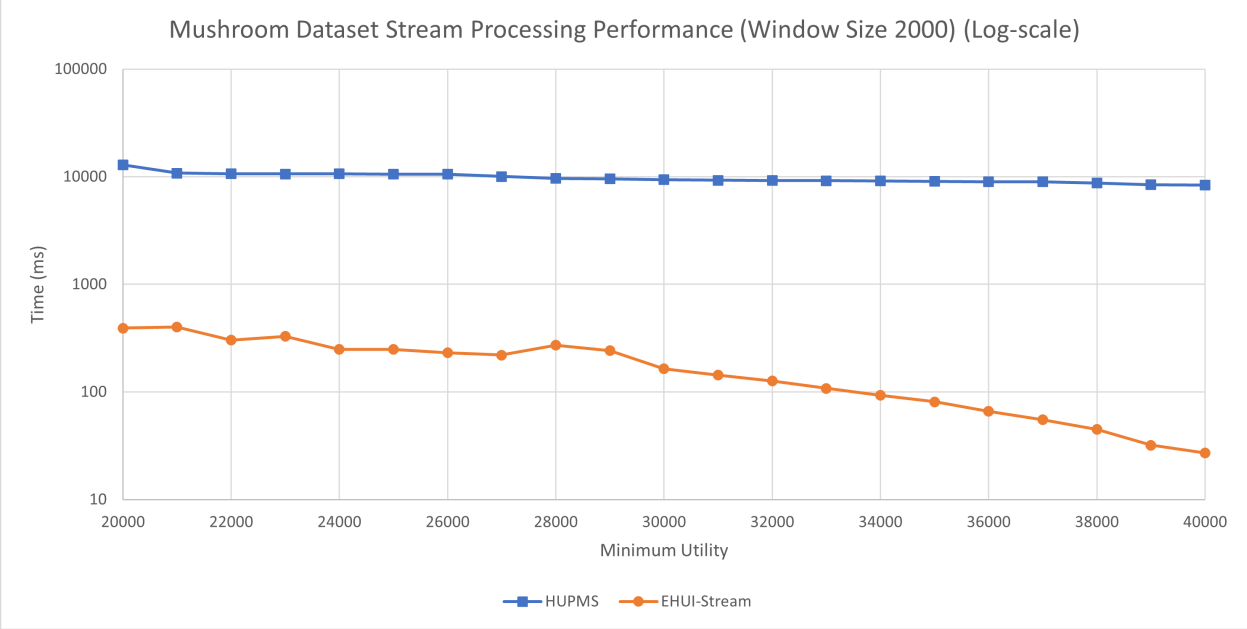

Figure 4.24: The stream mining performance comparison of EHUI-Stream and HUPMS with window size 2k using the mushroom dataset (log-scale graph)

compared to HUPMS using both sparse and dense datasets, we conclude our EHUI-stream is order of magnitudes faster than HUPMS when mining sparse dataset and in the same order of performance when mining dense dataset. We did not include more recent algorithm SOHUPDS into the experiment due to the part of their procedure is unclear for us to replicate, but SOHUPDS algorithm is not that much faster than HUPMS compared to our result. We conclude that our EHUI-stream algorithm has better stream mining performance when compared to the existing baselines especially when dealing with sparse datasets.

4.2.4 Experiments on Response Time after Data are Modified

Experiment 4.2.13. In this experiment, we simulate the scenario to re-mine the HUI after the unit price changes. We use foodmart dataset for evaluation. This is a unique feature of our proposed algorithm so there is no baseline to compare with. Here is our experimental design. We set the minimum utility threshold to be 3250. For every iteration, we add 1000 to a unique item in the foodmart dataset, we record the new high utility itemsets related to the price change item and record the response time. In Figures 4.25 and 4.26, we show the number of TID under modification when the transaction is modified and the re-mining response time respectively. We observe that most of the iteration has around 10 TIDs under modification and the re-mining response time would be less than 1ms.

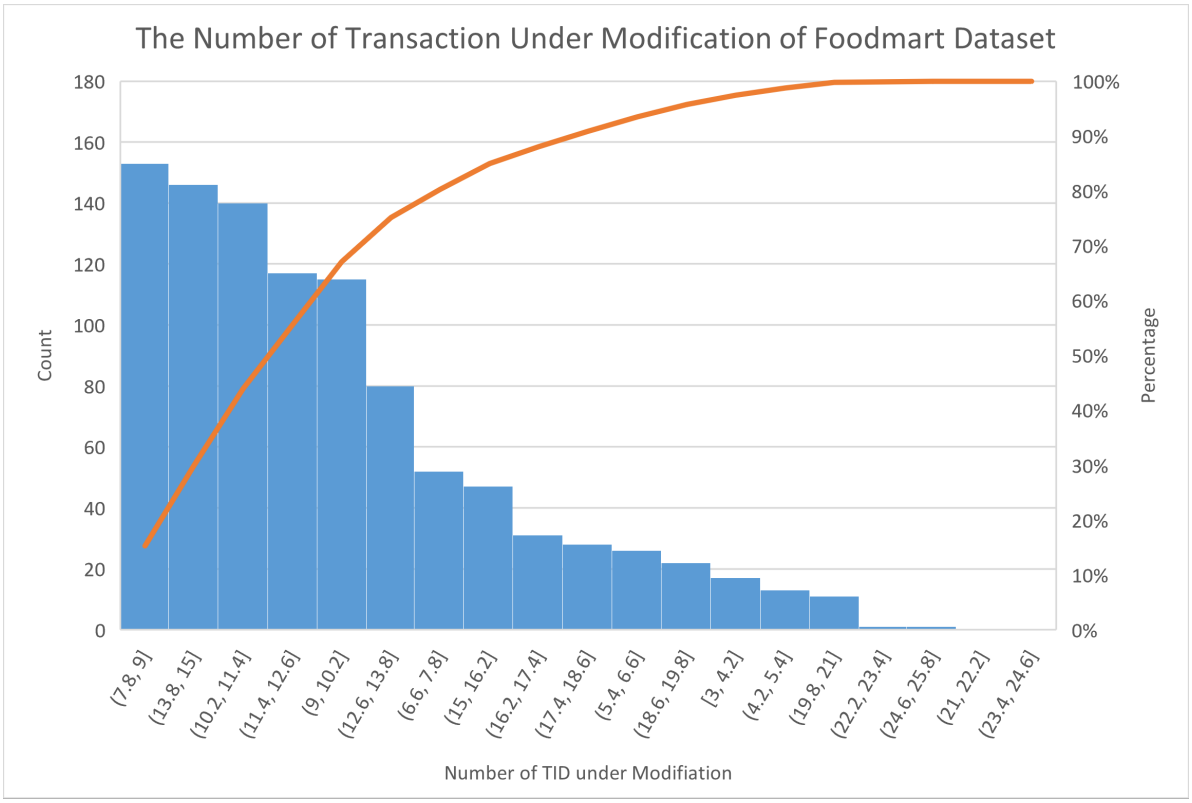

Figure 4.25: The number of TID under modification of the foodmart dataset

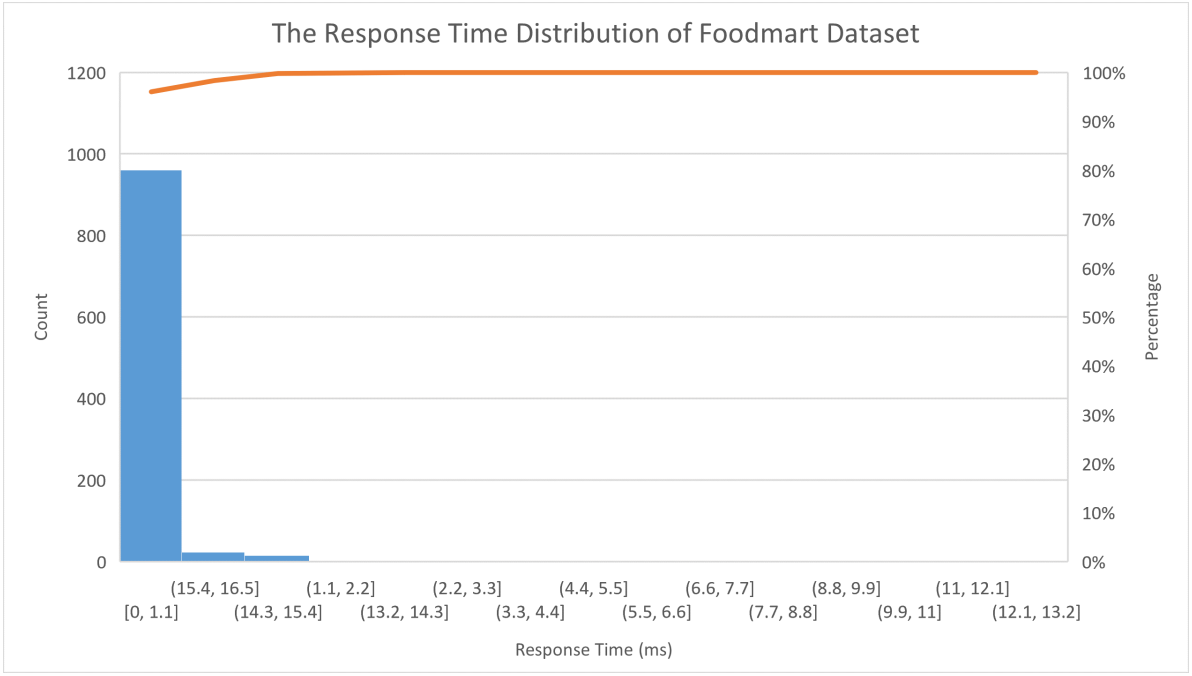

Figure 4.26: The number of TID under modification of the foodmart dataset (zoom-in view)

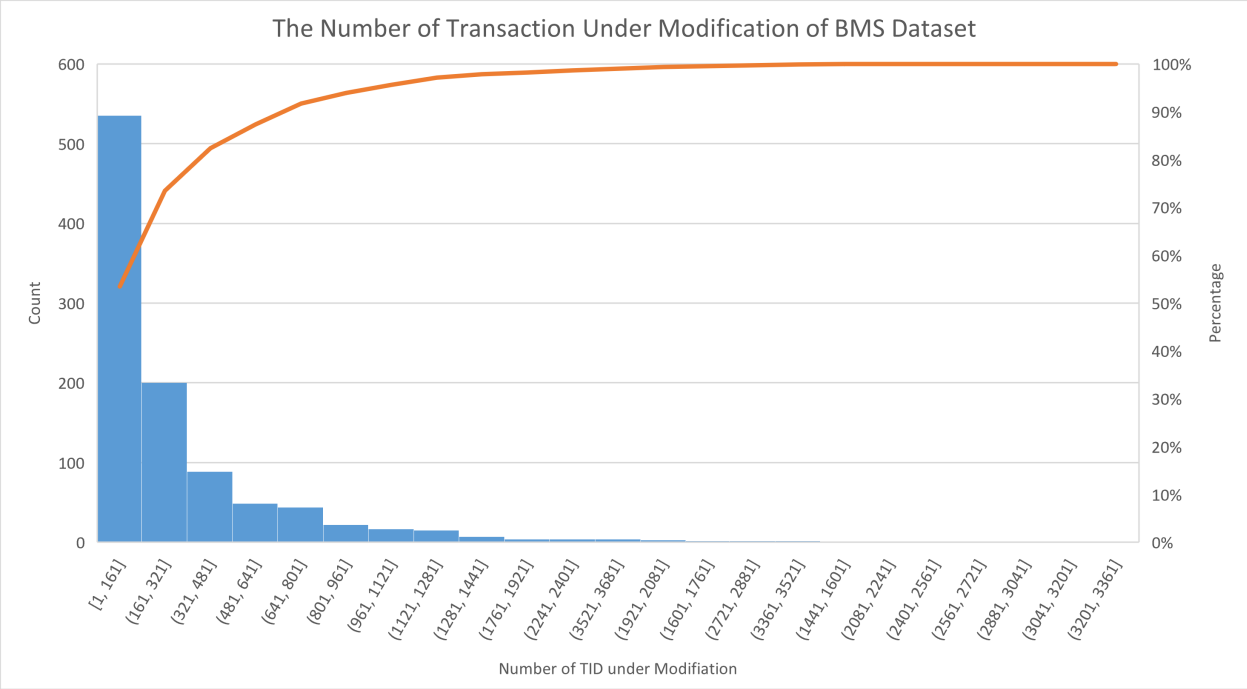

Figure 4.27: The number of TID under modification of the BMS dataset

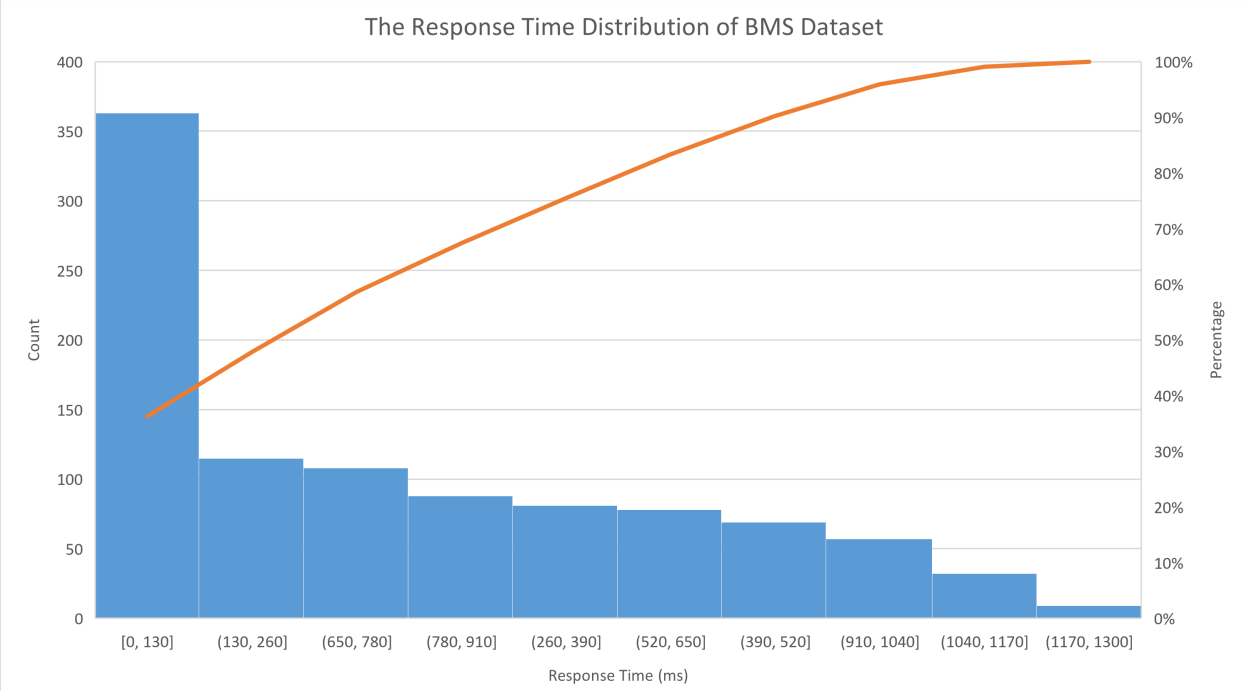

Figure 4.28: The number of TID under modification of the BMS dataset (zoom-in view)

Experiment 4.2.14. In this experiment, we use the BMS dataset for evaluation. The minimum utility threshold for BMS dataset is set to 290000. We randomly select 1000 itemsets to change the price and perform the re-mining process. Figures 4.27 and 4.28 show the number of tid under modifications when the price is changed for an item and the re-mining execution time respectively. The number of TID after modification is mostly within the range of 0 to 321, and the re-mining response time is mostly within 1 second. This shows the effectiveness of our EHUI data structure in terms of re-mining without re-scanning datasets.

In **summary**, this is the unique feature of our EHUI-stream algorithm, we show the re-mining performance after the price of a product is changed. Our EHUI-stream algorithm efficiently looks for the TID under modification and perform re-mining operation efficiently.

4.2.5 Experiments on Scalability

Experiment 4.2.15. In this experiment, we use foodmart dataset for evaluation. We keep doubling the size of the foodmart dataset from 4141 to $265,024$ while keeping the same percentage of the minimum utility to measure the execution time. Note that we set the minimum utility threshold be 0.03%. In Figure 4.29, we show that our algorithm has linear complexity with respect to the size of the data.

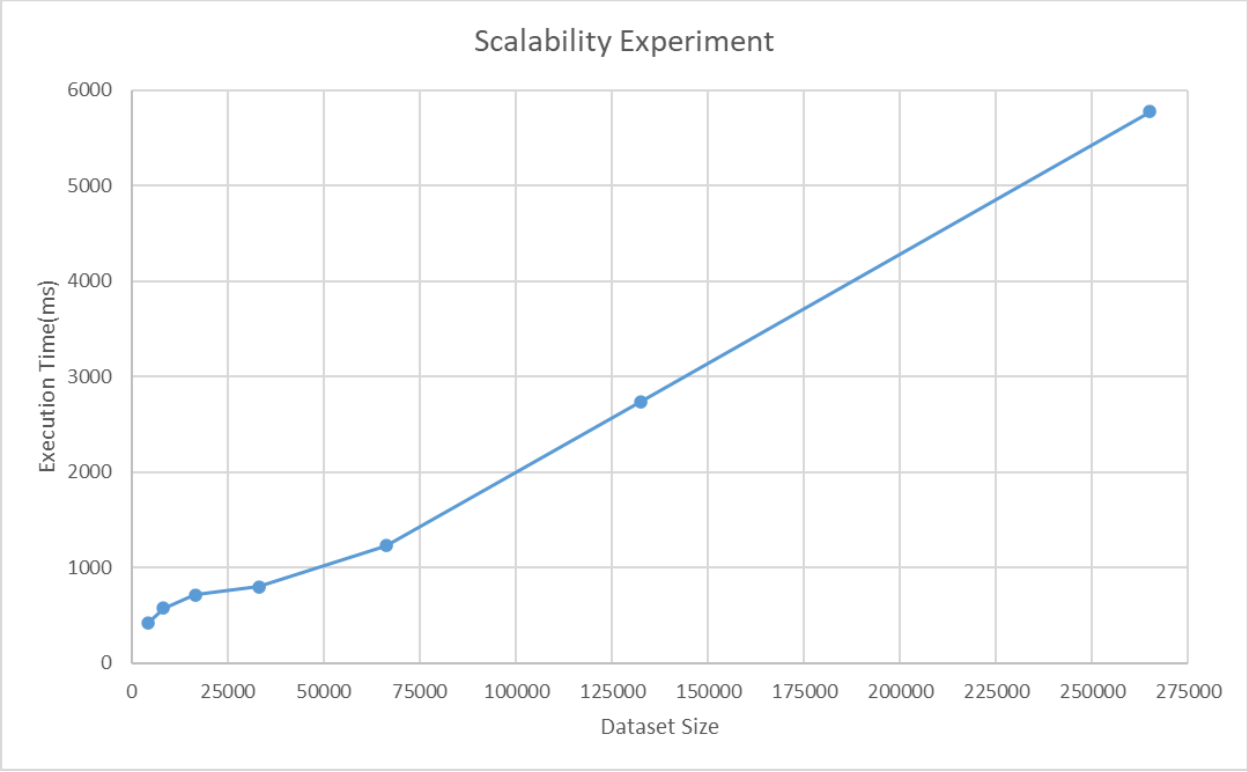

Figure 4.29: The scalability experiment using foodmart dataset

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Although improving performance on interactive stream mining is a challenging yet crucial task, it is also important to solve real-world problems. In this project, we mainly focus on several aspects of the problem and attempt to build a scalable HUI stream mining technique. Our strategy is to focus on filling the gaps to further enhance the performance of HUI stream mining algorithms along with enhancing the feature. In this thesis, I explored and answered the following two questions:

Q1. Can we design a high utility stream mining algorithm that is more efficient than existing algorithms?

A1. Yes, our EHUI-stream miner has the potential to compete with static batch processing algorithm in terms of the performance despite lacking the optimization compared to static algorithm. The extension of EHUI-stream allows it to mine high utility itemsets incrementally and has a faster response time compared to EIHI algorithm. Its stream

mining performance is faster than the existing algorithms in the HUI stream domain. To the best of our knowledge, our algorithm is the first one to remove the concept of batch in the high utility stream mining domain. All the previous work uses mini batches instead of stream but our algorithm treats each transaction as individual instead of an entity in a batch. This allows us to have more way to prune transaction when the window slides.

Q2. Can we design a stream mining algorithm to perform high utility mining on dynamic external profit data?

A2. Yes, this is the unique design of our algorithm. As it turns out, our algorithm has ways to handle re-mining after the transaction is added, deleted and updated. When the profit data is changed, our algorithm would treat the transaction is updated and look for those EUL lists that have been changed and perform re-mining for the updated EULs only. The flexible design allows our EULs to perform re-mining efficiently.

5.2 Future Work

In the future, we would like to figure out the potential way to improve our algorithm based on the hint of the existing algorithm. Since there is not much extensive research on exploring high utility stream mining algorithms, not to mention dynamic profit stream algorithms. we would like to explore the potential challenges and solutions. Potentially, we can use the new algorithm to perform interactive stream mining to solve real-world problems. During our experiment, we notice our EHUI-stream algorithm can be executed in parallel to further speed up the execution time, it would be the next step to refine our algorithm to be even more efficient.

The other potential enhancement is that there is rare but long itemset that generates a lot of utility patterns. It contributes to the spike of execution time when the long transaction is updated. Behind the scene, all the EUL list would need to be examined when a long transaction is modified. There are ways to limit the maximum length of patterns to reduce the spike the execution time. It would need to be consider when the algorithm is applied to the real-world scenario since a long pattern would slow down the response time a lot.

Moreover, we would like to integrate our best algorithm with the right platform to achieve better performance and scalability.

Bibliography

[AEH+11] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Mapreduce and PACT - comparing data parallel programming models. In Härder et al. [HLM+11], pages 25–44.

[AL99] Yonatan Aumann and Yehuda Lindell. A statistical theory for quantitative association rules. In Usama M. Fayyad, Surajit Chaudhuri, and David Madigan, editors, *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 15-18, 1999*, pages 261–270. ACM, 1999.

[AR18] K Anusha and K Usha Rani. Comparative evaluation of big data frameworks on batch processing. *International Journal of Pure and Applied Mathematics*, 119(16):937–948, 2018.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.

[ATJC12] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Ho-Jin Choi. Interactive mining of high utility patterns over data streams. *Expert Syst. Appl.*, 39(15):11979–11991, 2012.

[ATJL09a] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee. Efficient tree structures for high utility pattern mining in incremental databases. *IEEE Transactions on Knowledge and Data Engineering*, 21(12):1708–1721, 2009.

[ATJL09b] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee. Efficient tree structures for high utility pattern mining in incremental databases. *IEEE Trans. Knowl. Data Eng.*, 21(12):1708–1721, 2009.

[BJZ94] Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors. *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Morgan Kaufmann, 1994.

[Buc69] Werner Buchholz. A synthetic job for measuring system performance. *IBM Systems Journal*, 8(4):309–318, 1969.

[BYK+21] Yoonji Baek, Unil Yun, Heonho Kim, Hyoju Nam, Hyunsoo Kim, Jerry Chun-Wei Lin, Bay Vo, and Witold Pedrycz. RHUPS: mining recent high utility patterns with sliding window-based arrival time control over data streams. *ACM Trans. Intell. Syst. Technol.*, 12(2):16:1–16:27, 2021.

[CDE+16] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In Sun [Sun16], pages 1789–1792.

[CHZ+21] Haodong Cheng, Meng Han, Ni Zhang, Le Wang, and Xiaojuan Li. ETKDS: an efficient algorithm of top-k high utility itemsets mining over data streams under sliding window model. *J. Intell. Fuzzy Syst.*, 41(2):3317–3338, 2021.

[CJL+15] Alfredo Cuzzocrea, Fan Jiang, Carson Kai-Sang Leung, Dacheng Liu, Aaron M. Peddle, and Syed Khairuzzaman Tanbeer. Mining popular patterns: A novel mining problem and its application to static transactional databases and dynamic data streams. In *Trans. Large-Scale Data- and Knowledge-Centered Systems* [HKW+15], pages 115–139.

[CTL08] Chun-Jung Chu, Vincent S Tseng, and Tyne Liang. An efficient algorithm for mining temporal high utility itemsets from data streams. *Journal of Systems and Software*, 81(7):1105–1117, 2008.

[D+16] Jack Dongarra et al., editors. *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*. IEEE Computer Society, 2016.

[DBD+17] Kusum Deep, Jagdish Chand Bansal, Kedar Nath Das, Arvind Kumar Lal, Harish Garg, Atulya K. Nagar, and Millie Pant, editors. *Proceedings of Sixth International Conference on Soft Computing for Problem Solving - SocProS 2016, Volume 2, Thapar University, Patiala, India, December 23-24, 2016*, volume 547 of *Advances in Intelligent Systems and Computing*, 2017.

[DG17] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[EGA07] Alva Erwin, Raj P. Gopalan, and N. R. Achuthan. A bottom-up projection based algorithm for mining high utility itemsets. In Kok-Leong Ong, Wenyuan Li, and

Junbin Gao, editors, *Integrating Artificial Intelligence and Data Mining - Proceedings of the 2nd International Workshop on Integrating Artificial Intelligence and Data Mining (AIDM 2007), Gold Coast, Queensland, Australia, December 2007. Proceedings*, volume 84 of *CRPIT*, pages 3–10. Australian Computer Society, 2007.

[FGG+14] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S. Tseng. SPMF: a java open-source pattern mining library. *J. Mach. Learn. Res.*, 15(1):3389–3393, 2014.

[FWZT14] Philippe Fournier-Viger, Cheng-Wei Wu, Souleymane Zida, and Vincent S. Tseng. FHM: faster high-utility itemset mining using estimated utility cooccurrence pruning. In Troels Andreasen, Henning Christiansen, Juan Carlos Cubero Talavera, and Zbigniew W. Ras, editors, *Foundations of Intelligent Systems - 21st International Symposium, ISMIS 2014, Roskilde, Denmark, June 25-27, 2014. Proceedings*, volume 8502 of *Lecture Notes in Computer Science*, pages 83–92. Springer, 2014.

[FZL+19] Philippe Fournier-Viger, Yimin Zhang, Jerry Chun-Wei Lin, Hamido Fujita, and Yun Sing Koh. Mining local and peak high utility itemsets. *Inf. Sci.*, 481:344–367, 2019.

[GCM19] Melissa Gehring, Marcela Charfuelan, and Volker Markl. A comparison of distributed stream processing systems for time series analysis. In Meyer et al. [MRT+19], pages 205–214.

[GG16] Manu Goel and Kanu Goel. Fp-growth implementation using tries for association rule mining. In Deep et al. [DBD+17], pages 21–29.

[GHRL03] Chris Giannella, Jiawei Han, Edward Robertson, and Chao Liu. Mining frequent itemsets over arbitrary time intervals in data streams. *Bericht, Indiana University*, 2003.

[GSDF03] Lise Getoor, Ted E. Senator, Pedro M. Domingos, and Christos Faloutsos, editors. *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*. ACM, 2003.

[HKW+15] Abdelkader Hameurlain, Josef Küng, Roland R. Wagner, Alfredo Cuzzocrea, and Umeshwar Dayal, editors. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXI - Selected Papers from DaWaK 2012*, volume 9260 of *Lecture Notes in Computer Science*. Springer, 2015.

[HLM+11] Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz, editors. *Datenbanksysteme für Business, Technologie und Web*

(BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany, volume 180 of *LNI*. GI, 2011.

[HLW09] Tzung-Pei Hong, Cho-Han Lee, and Shyue-Liang Wang. An incremental mining algorithm for high average-utility itemsets. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 421–425, 2009.

[HPYM04] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, 8(1):53–87, 2004.

[HW19] Fabian Hueske and Timo Walther. Apache Flink. In Sakr and Zomaya [SZ19].

[JH20] Bijay Prasad Jaysawal and Jen-Wei Huang. SOHUPDS: a single-pass one-phase algorithm for mining high utility patterns over a data stream. In Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini, editors, *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, pages 490–497. ACM, 2020.

[JS16] Kevin Jacobs and Kacper Surdy. Apache Flink: Distributed stream data processing, 2016.

[KHF10] Mohammed G. Khatib, Xubin He, and Michael Factor, editors. *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*. IEEE Computer Society, 2010.

[Kri15] Srikumar Krishnamoorthy. Pruning strategies for mining high utility itemsets. *Expert Syst. Appl.*, 42(5):2371–2381, 2015.

[KYB+21] Heonho Kim, Unil Yun, Yoonji Baek, Hyunsoo Kim, Hyoju Nam, Jerry Chun-Wei Lin, and Philippe Fournier-Viger. Damped sliding based utility oriented pattern mining over stream data. *Knowl. Based Syst.*, 213:106653, 2021.

[KYK+21] Jongseong Kim, Unil Yun, Hyunsoo Kim, Taewoong Ryu, Jerry Chun-Wei Lin, Philippe Fournier-Viger, and Witold Pedrycz. Average utility driven data analytics on damped windows for intelligent systems with data streams. *Int. J. Intell. Syst.*, 36(10):5741–5769, 2021.

[LGHP14] Jerry Chun-Wei Lin, Wensheng Gan, Tzung-Pei Hong, and Jeng-Shyang Pan. Incrementally updating high-utility itemsets with transaction insertion. In Xudong Luo, Jeffrey Xu Yu, and Zhi Li, editors, *Advanced Data Mining and Applications - 10th International Conference, ADMA 2014, Guilin, China, December 19-21, 2014. Proceedings*, volume 8933 of *Lecture Notes in Computer Science*, pages 44–56. Springer, 2014.

[LHL11a] Hua-Fu Li, Hsin-Yun Huang, and Suh-Yin Lee. Fast and memory efficient mining of high-utility itemsets from data streams: with and without negative item profits. *Knowledge and information systems*, 28(3):495–522, 2011.

[LHL11b] Chun-Wei Lin, Tzung-Pei Hong, and Wen-Hsiang Lu. An effective tree structure for mining high utility itemsets. *Expert Syst. Appl.*, 38(6):7419–7424, 2011.

[LK06] Carson Kai-Sang Leung and Quamrul I. Khan. DSTree: A tree structure for the mining of frequent sets from data streams. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China*, pages 928–932. IEEE Computer Society, 2006.

[LKLH07] Carson Kai-Sang Leung, Quamrul I. Khan, Zhan Li, and Tariqul Hoque. CanTree: a canonical-order tree for incremental frequent-pattern mining. *Knowl. Inf. Syst.*, 11(3):287–311, 2007.

[LLC05] Ying Liu, Wei-keng Liao, and Alok N. Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In Tu Bao Ho, David Wai-Lok Cheung, and Huan Liu, editors, *Advances in Knowledge Discovery and Data Mining, 9th Pacific-Asia Conference, PAKDD 2005, Hanoi, Vietnam, May 18-20, 2005, Proceedings*, volume 3518 of *Lecture Notes in Computer Science*, pages 689–695. Springer, 2005.

[LWF16] Junqiang Liu, Ke Wang, and Benjamin C. M. Fung. Mining high utility patterns in one phase without generating candidates. *IEEE Trans. Knowl. Data Eng.*, 28(5):1245–1257, 2016.

[Mar10] I Marketing. key marketing trends for 2017. Technical report, Technical report, IBM. URL: https://www-01. ibm. com/common/ssi/cgi-bin/ssialias, 10.

[MCAP16] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S. Pérez-Hernández. Spark versus flink: Understanding performance in big data analytics frameworks. In Dongarra et al. [D$^+$16], pages 433–442.

[MMCR22] Anirban Mondal, Raghav Mittal, Parul Chaudhary, and Polepalli Krishna Reddy. A framework for itemset placement with diversification for retail businesses. *Applied Intelligence*, pages 1–19, 2022.

[MRT$^+$19] Holger Meyer, Norbert Ritter, Andreas Thor, Daniela Nicklas, Andreas Heuer, and Meike Klettke, editors. *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs ,,Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*, volume P-290 of *LNI*. Gesellschaft für Informatik, Bonn, 2019.

[NIY19] Takumi Nishina, Koji Iwanuma, and Yoshitaka Yamamoto. A skipping fp-tree for incrementally intersecting closed itemsets in on-line stream mining. In Yoshikawa et al. [Y$^+$19], pages 1–4.

[NMSM11] Mohammad-Hossein Nadimi-Shahraki, Norwati Mustapha, Md Nasir Sulaiman, and Ali Mamat. Efficient prime-based method for interactive mining of frequent patterns. *Expert Syst. Appl.*, 38(10):12654–12670, 2011.

[NNN+19] Loan T. T. Nguyen, Phuc Nguyen, Trinh D. D. Nguyen, Bay Vo, Philippe Fournier-Viger, and Vincent S. Tseng. Mining high-utility itemsets in dynamic profit databases. *Knowl. Based Syst.*, 175:130–144, 2019.

[NNV+21] Trinh D. D. Nguyen, Loan T. T. Nguyen, Lung Vu, Bay Vo, and Witold Pedrycz. Efficient algorithms for mining closed high utility itemsets in dynamic profit databases. *Expert Syst. Appl.*, 186:115741, 2021.

[NX10] Erich M. Nahum and Dongyan Xu, editors. *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[NYYL20] Hyoju Nam, Unil Yun, Eunchul Yoon, and Jerry Chun-Wei Lin. Efficient approach of recent high utility stream pattern mining with indexed list structure and pruning strategy considering arrival times of transactions. *Inf. Sci.*, 529:1–27, 2020.

[RD20] J. Ragaventhiran and M. K. Kavitha Devi. Map-optimize-reduce: CAN tree assisted FP-growth algorithm for clusters based FP mining on Hadoop. *Future Generation Comp. Syst.*, 103:111–122, 2020.

[RS19] Ravi Ranjan and Aditi Sharma. Evaluation of frequent itemset mining platforms using apriori and fp-growth algorithm. *CoRR*, abs/1902.10999, 2019.

[RY16] Heungmo Ryang and Unil Yun. High utility pattern mining over data streams with sliding window technique. *Expert Systems with Applications*, 57:214–231, 2016.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Khatib et al. [KHF10], pages 1–10.

[Sun16] Xian-He Sun, editor. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016.

[SZ19] Sherif Sakr and Albert Y. Zomaya, editors. *Encyclopedia of Big Data Technologies*. Springer, 2019.

[TAJL08] Syed Khairuzzaman Tanbeer, Chowdhury Farhan Ahmed, Byeong-Soo Jeong, and Young-Koo Lee. Efficient frequent pattern mining over data streams. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, pages 1447–1448, 2008.

[TMF03] Feng Tao, Fionn Murtagh, and Mohsen M. Farid. Weighted association rule mining using weighted support and significance framework. In Lise Getoor, Ted E. Senator, Pedro M. Domingos, and Christos Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*, pages 661–666. ACM, 2003.

[TWSY10] Vincent S. Tseng, Cheng-Wei Wu, Bai-En Shie, and Philip S. Yu. Up-growth: an efficient algorithm for high utility itemset mining. In Bharat Rao, Balaji Krishnapuram, Andrew Tomkins, and Qiang Yang, editors, *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010*, pages 253–262. ACM, 2010.

[WFGT15] Cheng-Wei Wu, Philippe Fournier-Viger, Jia-Yuan Gu, and Vincent S. Tseng. Mining closed+ high utility itemsets without candidate generation. In *Conference on Technologies and Applications of Artificial Intelligence, TAAI 2015, Tainan, Taiwan, November 20-22, 2015*, pages 187–194. IEEE, 2015.

[WYY00] Wei Wang, Jiong Yang, and Philip S. Yu. Efficient mining of weighted association rules (WAR). In Raghu Ramakrishnan, Salvatore J. Stolfo, Roberto J. Bayardo, and Ismail Parsa, editors, *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*, pages 270–274. ACM, 2000.

[Y+19] Masatoshi Yoshikawa et al., editors. *IEEE International Conference on Big Data and Smart Computing, BigComp 2019, Kyoto, Japan, February 27 - March 2, 2019*. IEEE, 2019.

[YRLF17] Unil Yun, Heungmo Ryang, Gangin Lee, and Hamido Fujita. An efficient algorithm for mining high utility patterns from incremental databases with one database scan. *Knowl. Based Syst.*, 124:188–206, 2017.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Nahum and Xu [NX10].

[ZFL+17] Souleymane Zida, Philippe Fournier-Viger, Jerry Chun-Wei Lin, Cheng-Wei Wu, and Vincent S. Tseng. EFIM: a fast and memory efficient algorithm for high-utility itemset mining. *Knowl. Inf. Syst.*, 51(2):595–625, 2017.

Appendix A

Implementation Details of the EFIM Algorithm

EFIM is the state of the art algorithm for mining HUI from static datasets. We dive deep into the implementation and see what makes EFIM better than the previous HUI algorithms, and the reason such algorithm is not suitable for stream mining. Let us consider the database in Table 2.6 for demonstration purposes.

A.1 Init Database Phase

Using SPMF format, the sample database looks like below.

3 5 1 2 4 6:30:1 3 5 10 6 5

Each item name is converted to a number, where item a becomes 1, item b becomes 2 etc. The transaction is on the left, the total utility value is in the middle and they combine the internal utility and the external utility together and place them to the right.

A.2 Calculate TWU and Filter Singleton by TWU

TWU means Transaction Weighted Utilization as mentioned in Definition 8. It serves as the upper bound on the utility measure and it is the most common optimization for mining high utility itemsets. EFIM converts the dataset to the following internal DB.

3[1] 5[3] 1[5] 2[10] 4[6] 6[5] Remaining Utility:30 Prefix Utility:0

3[3] 5[3] 2[8] 4[6] Remaining Utility:20 Prefix Utility:0

3[1] 1[5] 4[2] Remaining Utility:8 Prefix Utility:0

3[6] 5[6] 1[10] 7[5] Remaining Utility:27 Prefix Utility:0

3[2] 5[3] 2[4] 7[2] Remaining Utility:11 Prefix Utility:0

Let us define the minimum utility threshold be 31. Item 6 only appears in transaction 1. Even we combine all the utils where item 6 is involved, the maximum utility can only be 30 which is less than 31. It would be impossible that any itemset with item 6 can satisfy the minutils. The problem is that the TWU upper bound is loose and few candidates can be pruned away.

Using the sample DB, the TWU are computed as the map below.

Item 1 $\rightarrow$ TWU: 65

Item 2 $\rightarrow$ TWU: 61

Item 3 $\rightarrow$ TWU: 96

Item 4 $\rightarrow$ TWU:58

Item 5 $\rightarrow$ TWU:88

Item 6 $\rightarrow$ TWU:30

Item 7 $\rightarrow$ TWU:38

Based on this hashmap, EFIM can prune away items where TWU less than the minU-

tils. EFIM builds an primitive integer array called utilityBinArrayLU (LU stands for local utility, which is the same value as the TWU other than the name). It tries to enhance the performance via array index lookup rather than map lookup since the simple array index lookup is faster than map lookup.

A.3 Sort Items by TWU Ascend

After the filtering by TWU operation, EFIM sorts items based on TWU by ascending order. Assuming minUtils is 30 and itemsToKeep is [1,2,3,4,5,6,7] since none of the item has minUtil less than 30. After sorting based on TWU, the itemsToKeeps becomes [6,7,4,2,1,5,3] as item 6 has the lowest TWU which is 30, and item 7 has the second lowest TWU which is 38.

A.4 Covert Old Names to New Names (EFIM Specified)

Normally we cannot tell which item has higher TWU based on its item name. Using the sample data, the list has been re-arranged by TWU and out of order.

[6,7,4,2,1,5,3]

EFIM introduces a concept that converts old names to new names and vice versa. Based on new names, the smaller the number has lower TWU. For example, old names [6,7,4,2,1,5,3] can be converted to new names [1,2,3,4,5,6,7]. This can be achieved by two maps lookup as shown below.

MapOldToNew:

Old Item 6 → new item 1

Old Item 7 → new item 2

Old Item 4 → new item 3

Old Item 2 → new item 4

Old Item 1 → new item 5

Old Item 5 → new Item 6

Old Item 3 → new Item 7

MapNewToOld:

New Item 1 → Old Item 6

New Item 2 → Old Item 7

New Item 3 → Old Item 4

New Item 4 → Old Item 2

New Item 5 → Old Item 1

New Item 6 → Old Item 5

New Item 7 → Old Item 3

A.5 Update Dataset Transaction by New Names and Prune Transactions

From the sample database, recall that the original transactions at this point.

3[1] 5[3] 1[5] 2[10] 4[6] 6[5] Remaining Utility:30 Prefix Utility:0

3[3] 5[3] 2[8] 4[6] Remaining Utility:20 Prefix Utility:0

3[1] 1[5] 4[2] Remaining Utility:8 Prefix Utility:0

3[6] 5[6] 1[10] 7[5] Remaining Utility:27 Prefix Utility:0

3[2] 5[3] 2[4] 7[2] Remaining Utility:11 Prefix Utility:0

Those item names need to be updated based on oldToNewArray. Notice that from the third step: filterSingletonByTWU has removed unqualified items. During the conversion process, any oldToNewArray[unqalifiedItem] = 0 as it had never been filled previously. In other words, any old item that cannot covert to a new name is not qualified (this can be seen as a pruning trick).

As a result, the original transactions after the name conversion:

7[1] 6[3] 5[5] 4[10] 3[6] 1[5] Remaining Utility:30 Prefix Utility:0

7[3] 6[3] 4[8] 3[6] Remaining Utility:20 Prefix Utility:0

7[1] 5[5] 3[2] Remaining Utility:8 Prefix Utility:0

7[6] 6[6] 5[10] 2[5] Remaining Utility:27 Prefix Utility:0

7[2] 6[3] 4[4] 2[2] Remaining Utility:11 Prefix Utility:0

A.6 Sort Items in Transaction by TWU in Ascending Order (Contains EFIM Specified Optimization)

Recall that after the name conversion, the item name is sorted by its TWU. This process will be as simple as sorting the item without extra TWU lookup.

1[5] 3[6] 4[10] 5[5] 6[3] 7[1] Remaining Utility:30 Prefix Utility:0

3[6] 4[8] 6[3] 7[3] Remaining Utility:20 Prefix Utility:0

3[2] 5[5] 7[1] Remaining Utility:8 Prefix Utility:0

2[5] 5[10] 6[6] 7[6] Remaining Utility:27 Prefix Utility:0

2[2] 4[4] 6[3] 7[2] Remaining Utility:11 Prefix Utility:0

A.7 Rearrange Transactions for Transaction Merging and Pruning

The key idea is to sort transactions based on two rules:

Rule 1: Sort transactions by length in ascending order under the condition of rule 2

Rule 2: Rule 2 can override rule 1, more important transaction will be placed at the top.

For example, compare T1 and T2.

T1: 1[5] 3[6] 4[10] 5[5] 6[3] 7[1] Remaining Utility:30 Prefix Utility:0

T2: 3[6] 4[8] 6[3] 7[3] Remaining Utility:20 Prefix Utility:0

Although T2 is shorter than T1 and supposedly T2 should be before T1. However, EFIM compares util item by item. In this case, both T2 and T1 has item 6 and item 7, but T1 has item 5 and T2 has a lower TWU item 4 only, which makes T1 is potentially more important than T2, so T1 will be before T2.

Before rearranging the transactions: T1: 1[5] 3[6] 4[10] 5[5] 6[3] 7[1] Remaining Utility:30 Prefix Utility:0

T2: 3[6] 4[8] 6[3] 7[3] Remaining Utility:20 Prefix Utility:0

T3: 3[2] 5[5] 7[1] Remaining Utility:8 Prefix Utility:0

T4: 2[5] 5[10] 6[6] 7[6] Remaining Utility:27 Prefix Utility:0

T5: 2[2] 4[4] 6[3] 7[2] Remaining Utility:11 Prefix Utility:0

After rearranging the transactions:

T3: 3[2] 5[5] 7[1] Remaining Utility:8 Prefix Utility:0

T5: 2[2] 4[4] 6[3] 7[2] Remaining Utility:11 Prefix Utility:0

T2: 3[6] 4[8] 6[3] 7[3] Remaining Utility:20 Prefix Utility:0

T4: 2[5] 5[10] 6[6] 7[6] Remaining Utility:27 Prefix Utility:0

T1: 1[5] 3[6] 4[10] 5[5] 6[3] 7[1] Remaining Utility:30 Prefix Utility:0

A.8 Prune Empty Transactions

Pruning empty transaction is the trick by EFIM. Since the transactions are sorted by the length in ascending order, it can prune away all the empty transactions at the beginning of the list. Although it can rarely prune away any transaction unless the entire transaction is made of pruned items with TWU less than the minutils which is rarely the case.

A.9 Optimize SubTreeUtility

EFIM pruned away items based on TWU, sorted the items based on TWU and sorted the transactions based on the rules mentioned previously. The sub-tree utility is an overestimation of the item utility based on the tail utility.

A.10 Backtracking-Mine High Utilities Items

This is the final stage and the most complex stage of EFIM. For EFIM, there are 4 parameters as the input: transactionOfP, itemsToKeep, itemsToExplore and prefixLength. Using the sample, the transactionOfP is all the transactions in the dataset as a transaction list, itemsToKeep is the items filtered by TWU, and itemsToExplore is the items filtered by subTree Utils. This process is done recursively until all the items are found.