Mining Top-K High Utility Itemsets in Streaming Data: A Comparative Study

Veronica Sharma

Computer Science

Indraprastha Institute of Information Technology, Delhi (IIIT-D), India

A Thesis Report submitted in partial fulfilment for the degree of

MTech Computer Science

25^{th} October 2016

1. Dr. Vikram Goyal (Thesis Adviser)

2. Dr. Chetan Arora (Internal Examiner)

3. Prof. S K Gupta (External Examiner)

Date of the defense: 25^{th} October 2016

Signature from Post-Graduate Committee (PGC) Chair:

Abstract

High Utility Itemset Mining (HUIM) has gained significant progress in recent years. The HUIM refers to the method of finding most relevant itemsets from a database and it finds its applications in the domain of senosor data analytics, ad-click data analytics and retail stores. The HUIM allows to associate notion of utility with each item which was not possible in in the case of frequent pattern mining (FPM). In FPM, only presence or absence of an item is considered in a transaction itemset and hence the approach does not allow provide flexibility when different items have different importance. The focus of pattern mining work has been limited to mainly static databases. However, with the increase in data and need of timely information requires existing methods to be either scaled to or adapted to the streaming environment. The key concerns for streaming data are high throughput computations with minimum time and space constraints. In this thesis, we implemented and compared the *top-k* streaming version of state-of-the-art algorithms T-HUDS(High Utility Itemset Mining over Data Stream), FHM(Faster High Utility Itemset Mining),EFIM(Efficient High Utility Itemset Mining) on various databases. Our experimental results show that Stream-FHM and Stream-EFIM outperforms tree based T-HUDS algorithm. The Stream-FHM results are better for sparse databases and Stream-EFIM results better for dense databases.

Acknowledgement

I would like to express my special appreciation and thanks to my advisor Dr. Vikram Goyal who have been a tremendous mentor for me. I would like to thank you for encouraging my research and helped me in all the time of research and writing of this thesis. Your advice on both research as well as on my career has been priceless.

Besides my advisor, I would like to thank Siddharth Dawar (Ph.D. Research Scholar, Data Engineering) for supporting me all the time. I would also like to thank you for guiding me at every point when I faced difficulty and for letting my defense be an enjoyable moment. Great thanks for your brilliant comments and suggestions.

I would like to thank all the faculty members of IIIT-Delhi, especially to Data Engineering Department for imparting the best of knowledge and moulding my future. I am also thankful to all my friends especially Ruchita Bansal, for being with me at each step whenever I needed their support.

A special thanks to my family. Words cannot express how grateful I am to my mother, and father for all the sacrifices that they have made on my behalf. Great thanks to my brother for supporting me and guiding me to the right path at all times of my career. Your prayers for me have sustained me so far. I would like to thank all my whole family and my friends who supported me in writing and inspired me to strive towards my goal.

This work was supported in parts by Infosys Center for Artificial Intelligence, IIIT-Delhi.

Declaration

This is to certify that the MTech Thesis Report titled **Mining Top-K High Utility Itemsets in Streaming Data: A comparative Study** submitted by **Veronica Sharma** for the partial fulfilment of the requirements for the degree of MTech in Computer Science is a record of the bonafide work carried out by her under my guidance and supervision at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Vikram Goyal
Indraprastha Institute of Information Technology, New Delhi

Contents

List of Figures

List of Tables

1

Research Motivation and Aim

Frequent Itemset Mining(FIM) [8] has emerged an important area in data mining in recent years. In a transaction database, each transaction consists of different items bought by a customer. An itemset refers to set of items clubbed together. Frequent Itemset Mining aims at finding those Itemsets which are frequently bought by the customers in the database. The study of *association rule mining* [2] helps to define the relationship between two or more items in a database. The percentage of transactions containing the itemset in a database represents the *support* of the item. Associations rule mining proceeds with generation of frequent itemsets having support greater than the minimum support threshold and then finding the rules among those frequent itemsets. Each rule has a *confidence* value of how an itemset is associated with another itemset and with what degree. It not only provides retail businesses with a knowledge of frequently occurring itemsets but also helps to suggest the possible location of each item adjacent to another item is kept in retail stores. However, frequent itemset mining approach suffers a major drawback. It did not include any information about the quantity of items present in a single transaction. It works on the boolean principle of 0 and 1. 1 represents the item is present in the transaction whereas 0 indicates the absence of the item. However, this concept cannot be applied in the real world in order to fetch more business and operations.

To overcome the limitations of frequent itemset Mining, **High Utility Itemset Mining**(HUIM) [10] concept was introduced. *HUIM* aims at finding the itemsets which bring huge profits to retail stores and can give key insights of the data. High

utility itemset Mining finds extensive applications in transaction retail stores, sensor data [1], biomedical domain, and network operations in telecommunication domain. In retail stores, a transaction consists of different items with external and internal value as key characteristic attributes of each item. The Internal value of item depicts the quantity of each item in a transaction whereas external value exhibits the actual price of the item. Thus, the utility of an item is the product of Internal and external values. *HUIM* mine the itemsets having high utility from the database. Consider, we have transaction comprising of three items as flour, curd, and gold with Internal values of each item is 10, 20, 1 respectively. The external utility of each item in rupees is 100.0, 40.0 and 1,00,000.00 respectively. We can see although the quantity of a gold item is less but overall utility of gold is far greater than combined utility of other two items. Thus, a single item brings a huge profit difference to retail stores if we consider its both internal and external values also. The concept helps retail business officials to develop strategies to identify and made available profitable items and to reduce the inventory size. High utility itemset Mining has also been extensively used in query expansion in Information Retrieval domain. Consider, the set of online documents as transactions where each item is represented by the words contained in the document. The TF-IDF score can be used as the external value of the word. If the user enters an incomplete query in the search engine, the *HUIM* can be used to generate high utility itemsets in form of set of recommended query words. The possible retrieved words can be used by the search engine to expand the query and retrieve the most probable list of documents as required by the user.

Finding *High Utility Itemsets* is difficult than *frequent itemsets*. Frequent itemsets were obtained based on downward closure property of itemsets to prune the search space effectively. *Downward Closure* property states if an itemset is a low-frequency itemset having frequency less than the minimum support threshold, the supersets of that itemset will be less frequent and therefore should be pruned out. But downward closure property cannot be applied to high utility itemset since supersets of a high utility itemset can be weighted more frequent than the itemset. Therefore, a new notion of Transaction Weighted Utility *TWU* was introduced to prune the search space. Most high utility itemset mining algorithms work in two phases. In the first phase, the algorithm finds the candidates suitable to be considered as high utility itemsets. In

the second phase, the candidates are verified with another database processing to get the exact utility of each itemset. In order to find *High Utility Itemset*, the user needs to provide the minimum utility threshold for the high utility itemsets to be retrieved. Many algorithms use this pre-defined threshold and increase the threshold to obtain high utility itemsets efficiently. But this concept introduces a problem if the user is not a domain user. Setting a low threshold value can lead to the generation of various candidates which increases processing time and space whereas setting a high threshold value can lead to discovering limited or no high utility itemsets. To overcome the problem, a concept of *top-k* was introduced. Now user just needs to specify how many top high utility itemsets he/she wants to retrieve. Although, *HUI* can be applied to various fields of computing, but recent growth in big data and Internet of things have put limitations on time and space with unbounded size of data.

In recent years, to tackle the unbounded stream of data , the concept of data streams [3] increasingly gained importance. A data stream is an unbounded, continuous stream of data. The data if once processed cannot be retrieved back since data stream focuses on current data. This concept gained high importance since it considers the large volume of data stored in limited space and should be processed in minimum time. Also, data stream incorporates the previous window statistics such as minimum utility threshold for next window. Its applications lie at the heart of big data and Internet of things, where a huge data is gathered through sensors, cellular networks, biomedical data of genes collection, online ad-clicks, e-commerce and much more. A couple of algorithms have been proposed to find high utility itemsets in data stream. In this report, we are using sliding window based data stream model where window contains a portion of data to be processed and return the high utility itemsets for current data. This helps us to gain insights of how with the change of time and incoming data, the trend of high utility itemsets can change. Consider a retail store data stream scenario, where a set of transactions are coupled to form batch and one or more batches are coupled to form a fixed size window. Thus, whole streaming data can be processed in multiple parts with current data to be processed resides in the sliding window. Window returns the high utility itemsets processed for current data . As soon as the current window is processed, the window slides by a single batch to include the next batch and

discards the oldest batch information from the window. The statistics for the next window are drawn from the previous window to efficiently discover the high utility itemsets.

Top-k High Utility Itemset Mining over Data Stream, T-HUDS [18] algorithm is a two-phase algorithm that mines the *top-k* itemsets from a data stream. The algorithm uses a tree structure to include current window containing a batch of transactions. It defines the process of increasing the pruning threshold in order to reduce the count of candidates to be generated. Comparing to the state-of-the-art algorithms, *T-HUDS* suffers from memory and time delays. Also, due to a huge count of generated candidates, *T-HUDS* takes a lot of time for candidate verification as part of the second phase. Static Transaction database algorithms like *Faster High Utility Itemset Mining, FHM* and *Efficient High Utility Itemset Mining, EFIM* provides a better statistics compared to *T-HUDS* which was a streaming algorithm. *FHM* uses utility lists to store items data. It provides an efficient mechanism to reduce the number of join operations of two or more utility lists. *EFIM* uses general array list data structure and proposes a concept of projected database to reduce the database scan time.

Our research contribution can be summarised as follows:

- We conducted a thorough study of various state-of-the-art algorithms for mining high utility itemsets from static and dynamic databases.
- We implemented the state-of-the-art algorithm in the dynamic database, the streaming algorithm *T-HUDS*.
- The *top-k* version of the state-of-the-art algorithms of static databases, *Stream-FHM* and *Stream-EFIM* were implemented. We naively applied the algorithms on each window without reusing the computation from the previous window.
- We conducted extensive experiments on real databases and the results shows that *Stream-FHM* performs better on sparse datasets while *Stream-EFIM* shows better results for dense datasets in terms of execution time. *Stream-EFIM* reduces the memory consumption by 100-1000 times compared to other two algorithms.

Thesis report has been structured in following way:

Chapter 1: Introduces the topic with recent progress made in the context of data mining. Further, it provides the motivation for conducting this research study and our contributions towards this thesis.

Chapter 2: Talks in brief about the related work done in this field starting from the emergence of the concept of itemset mining to till date state-of-the-art algorithms.

Chapter 3: Describes the problem statement and background of mining high utility itemsets.

Chapter 4: The State-of-the-algorithms on static and dynamic transaction algorithms are discussed in depth.

Chapter 5: Implementation of naive top-k version of static transaction algorithms is discussed.

Chapter 6: Conducted experiments on real datasets and analyzed the results. The results shows how static transaction database based algorithms like *FHM* and *EFIM* when implemented naively on data streams performs far better than data stream based *T-HUDS* algorithm.

Chapter 7: This chapter concludes the thesis with the study conducted. Also, it proposes the possibility of future work.

2

Related Work

Frequent Itemset Mining [8, 16] has been extensively studied in the data mining domain. The concept of association rule mining makes use of frequent itemset mining which originates from the retail domain. Given a retail transaction dataset consisting of a set of transactions with each transaction consists of a set of items. The idea was proposed to mimic the retail store's customer-item relationship. Each customer is a single transaction whereas each item the customer purchases is a transaction item. Since the customer is the key focus for the retail domain, there was a need to develop an algorithm which can state item-item relationship to make the customer buy more items from the retail store. The emergence of the concept of Association Rule Mining greatly helps to achieve the objective. As stated by Zaki et al. [17] (1997) association rule mining defines the concept of support of an item. The *support* of an item is the percentage of transactions in the database that contains the itemset. Association Rule Mining can be stated in two phases. In the first phase, it aims to find all frequent itemsets, the itemsets occurring in the database with a certain user specified frequency called minimum support. In Second phase various rules are formed among the frequent itemsets. The search space for frequent itemsets was pruned out using *downward closure* property. *Downward Closure* property states that if an item/itemset is a non-frequent item/itemset having support less than the minimum support, then all the supersets of that item/itemset will also be non-frequent. The concept of Frequent Itemset helps to gain insights of all the items frequently bought by the customers in retail stores. Also, finding the rules from those frequent itemsets helps to gain insights for locating each item in retail store based on item-item relationship.

Agrawal et al. [2] (1994) proposes an algorithm named Apriori for mining Association rules in retail domain. Apriori works in a breadth first search manner, initially finding the single size frequent itemsets above a minimum support. When single size frequent itemsets are discovered, the algorithm scans the database again and find the itemsets of size two above minimum support. In such manner, apriori generates k size itemsets with k database scans. The algorithm can find all the frequent itemsets but lacks the efficiency as well as it was time and space consuming. Zaki et al. [17] (1997) proposes a faster algorithm named *Eclat* which uses Tid-list data structure. Every item/itemsets contain a list of all transaction id in which item was present. *Eclat* also works in a breadth first search manner as *Apriori*. Support of an item/itemset is indicated by the size of Tid list of each item/itemset. The algorithm first forms Tid list of single items and generates the frequent itemsets above a given minimum support. After generating the single frequent items, it generates frequent itemsets of size two by intersecting the Tid lists of single frequent items. It works for k itemsets by intersecting Tid list of size *k-1*. This algorithm reduces the database scans but results in generation of huge number of candidate itemsets.

Further progress in frequent itemset mining come with increasing the efficiency of the algorithms by reducing the size of generated candidate itemsets and rules. Pei et al. [11](2000) introduces an algorithm for finding the frequent closed itemsets. The algorithm works similar to Apriori but with the reduction in the number of generated association rules. A Major advancement in frequent itemset mining comes with the introduction of tree based mining frequent itemsets. Han et al. [8](2000) proposes a new tree based data structure named *FP-tree* and an algorithm called *FP-growth*. The items in each transaction are arranged in decreasing order of support and are inserted in *FP-tree*. Each node of the tree contains the information about each item in transaction. Each node consists of name, support, link to the child and link to the parent of the current item. The name represents the name of the item node, support defines the number of transactions along the path in which the current node is present. The tree starts with a root node which only contains the link to all its child nodes. A header table is constructed to store the information of unique items present in the database. Header table is used for faster tree traversal and contains each item in decreasing order

of their support value. Header table is composed of the name of the unique item with a pointer to a linked list of the same item name. This linked list contains all the nodes with same item name linked to each other. The *FP-growth* algorithm works with processing the header table from the bottom. The corresponding linked list is scanned to detect the paths in which item is present. Paths are combined to form *local FP-tree* which contains all paths in which a particular item i is present. All frequent itemsets are generated above given minimum support by the recursive traversal of *local FP-tree*. When the processing of item i is done over the whole tree, the algorithm repeats its task to process next item present in header list.

Frequent Pattern Mining consider only the binary presence of items. It does take into account the quantity of each item in a transaction. Thus, if an item is present, implies customer can only purchase one quantity of the item which compliments the real life retail store scenario. To overcome this drawback, the concept of *High Utility Itemset Mining (HUIM)* [3, 4, 5, 10, 15] was introduced. Two new attributes of an item were introduced. An item apart from its name should have an internal and external value associated with it. The internal value represents the quantity of the item whereas the external value represents the price of an item. The product of internal and external value results in *utility* of an item. The main focus lies in increasing the profits earned by a retail store over whole transaction dataset. The retail stores want the in-depth knowledge of the different set of items which can increase their whole business and operations. Also, the *HUIM* concept could be applied to various other fields like transaction retail stores, sensor data through sensors information [1], networking data in telecommunication domain, genes data in biomedical domain and much more. But it was not that much easier to calculate High Utility Itemsets as compared to Frequent Itemsets. The search space of *HUIM* cannot be pruned out using downward closure property since a high utility itemset can have a superset of high utility. To cater this problem, concept of *Transaction Weighted Utility (TWU)* [10] (2005) was introduced. *TWU* is an overestimated utility of an item/itemset which states any item in a transaction should have utility less than or equal to *TWU*. Thus, itemsets having the value of *TWU* less than the minimum threshold passed by user, those itemsets will not be considered as high utility itemsets.

Ahmed et al. [10] were the first to introduce the concept of *HUIM* over tree data structure. A two-phase recursive algorithm *IHUP* was proposed to mine *High Utility Itemsets*. Each node contains the information of node name, support, and *TWU* of each item. Items in each transaction were sorted in decreasing order of *TWU* since this order produces the best result as compared to lexicographic order or frequency order. But the algorithm generates a huge number of candidates which ultimately results in high space and time complexity. Tseng et al. [15] (2010) proposes another algorithm named *UP-growth* and a tree data structure *UP-tree*. *UP-tree* is a similar data structure to *FP-tree* with each node consists of node name, node utility, support, links to child nodes, a link to the parent node. The node utility is a better upper bound than *TWU* used in *IHUP*. The node utility indicates an upper bound value than the exact utility of itemset in the transactions from root to node. *UP-growth* is a pattern growth algorithm which incorporates new strategies like discarding global unpromising item(DGU), discarding global node utility (DGN) for reducing the number of candidates generated in the first phase.

Liu et al. [9] (2012) proposed a vertical mining algorithm called *HUI-Miner*. The authors use a utility-list data structure to store each item-Tid, exact utility in that transaction and remaining utility. Initially, the algorithm scans the database and calculates each item's *TWU*. All items in each transaction are sorted according to increasing the order of *TWU* since this ordering results in the generation of a fewer number of candidates. Another database scan is performed to construct the utility-list of high utility singleton items. These utility lists are then intersected to form high utility itemsets of size two. The algorithms work in a similar manner to generate high utility itemsets of greater length. The intersection of utility lists results in increasing cost in terms of time. Thus, Viger et al. [6] (2014) proposed a data structure *Estimated Utility Co-occurence Pruning (EUCS)* and an algorithm *Fast High-Utility Miner (FHM)* to reduce the number of join operations while mining high utility itemsets from utility list data structure. Zida et al. [13] presented a novel algorithm *EFficient High Utility Itemset Mining (EFIM)* which introduces an array based utility counting approach named *Fast Utility Counting* to calculate the utility upper bounds. Also, to reduce the cost of database scan, EFIM introduces a concept of database projection and transaction

merging in linear time and space.

With the huge growth in big data and internet of things, a continuous and unbounded stream of data is produced at a very high speed. A data stream is a concept of fast processing an unbounded group of data dynamically with system space constraints. Static databases do not incorporate the feature of dynamism. In streaming data, more information needs to be tracked and requires greater complexity to manage. Storage needs to be dynamically adjusted for generated high utility itemsets. Also, the possibility of some non high utility itemsets in some stream data can become high utility in future streams data. To tackle all these complexities, results need to be reused to work for future streams with a concept of *build once mine many*. A couple of algorithms have been developed using data stream models. Three types of models are studied in literature for modelling streaming scenarios. *Landmark Model* which points a time called landmark and mines the high utility patterns from that landmark to current time. *Time Fading Model* is similar to landmark model but gives more importance to recent data compared to historical data using a decay factor. *Sliding Window Model* consists of a fixed size window containing data in form of group of transactions. When a window is processed, the oldest group of transactions is removed and new group of transactions is added for further processing. The window slides over the data once a single window is processed and incorporates the incoming group of transactions.

Ahmed et al. [3] (2011) were the first to propose a tree data structure called *HUS-tree* and an algorithm for mining high utility itemsets from data stream under sliding window model. The proposed tree structure keeps batch wise track of itemsets in the transaction from root to node. The algorithm works with TWU concept to prune out the search space of candidates. Zihayat et al. [18] (2014) proposes another data structure called *HUDS-tree* and an algorithm for mining *top-k* high utility itemsets from data stream under sliding window model. The algorithm uses user-specified *top-k* value to generate k top high utility itemsets. In this concept, the user need not specify the minimum utility threshold to prune out the candidate search space. Instead only k value is specified by the user, the number of high utility itemsets user wants to view. Instead of storing TWU with each node, *HUDS-tree* uses better upper bound estimate called *prefix utility* and prunes the search space better than *HUS-tree*. Several strategies

were proposed by authors to increase the minimum threshold from zero before starting the mining process in order to prune out the candidates.

3

Background

In this section, we present some of the definitions formally stated in earlier works. The various state-of-the-art algorithms for mining high utility itemsets and data structures used in static and dynamic transaction databases are discussed.

In a retail domain transaction database, we have set of n distinct items $I = \{i_1, i_2, i_3, ..., i_n\}$, where each item has two attributes associated with it. Each item i_p has internal value as quantity of an item $quant(i\_p)$ and external value depicting the price of an item as $pr(i_p)$. An itemset X of length k is a set of k items, $X = \{i_1, i_2, i_3, ..., i_k\}$. A transaction database of size d consists of d number of transactions as $D = \{T_1, T_2, T_3, ..., T_d\}$ where every transaction has subset of items belonging to I. Every item i_p in each transaction has a quantity $quant(i_p, T_d)$ associated with it and price $pr(i_p)$.

Table 3.1: *Transaction Database*

		TID	Transaction	Transaction utility
SW1	B1	T_1	$(a:1)\,(c:1)\,(d:2)$	24
		T_2	$(a:2)\,(c:6)\,(e:2)\,(f:5)$	59
	B2	T_3	$(a:1)\,(b:2)\,(c:3)\,(d:3)\,(e:1)$	58
SW2		T_4	$(b:4)\,(c:3)\,(d:3)\,(e:2)$	71
		T_5	$(b:2)\,(c:2)\,(e:1)\,(f:2)$	32
	B3	T_6	$(a:2)\,(f:5)$	21

Table 3.2: *Profit Table of Items*

Item	a	b	c	d	e	f
Profit	3	6	5	8	4	3

The utility of an item and itemset can be computed in the following manner:

Definition 1. ***(Utility of an itemset in a database D)*** *The utility of an itemset* $X = \{x_1, x_2, x_3, ..., x_n\}$ *is denoted as* $u(X,D)$ *and defined as the sum of utilities of itemset* X *in the whole database, i.e.* $\sum_{X \subset T_d \wedge T_d \in D} u(X,D)$.

Consider the given example database shown in Table 3.1 and the external price value of each item in profit table 3.2. The utility of an itemset $X = \{A, C\}$ in transaction T_1 is sum of $u(a, T_1) + u(c, T_1)$, i.e, $3 + 5 = 8$. Similarly, the utility of itemset $\{a, c\}$ over example database is $u(ac, T_1) + u(ac, T_2) + u(ac, T_3) = 8 + 36 + 18 = 62$.

Definition 2. ***(High Utility Itemset, HUI)*** *A High Utility Itemset is an itemset* X *in database* D *whose utility is greater than and equal to the minimum utility threshold min_util specified by the user. In the example database, the utility of itemset* $\{c, e\}$ *is* $u(ce, T_2)+u(ce, T_3), +u(ce, T_4)+u(ce, T_5) = 38+19+23+14 = 94$. *If min_util specified by user is* 90*, then the itemset* $\{c, e\}$ *is a high utility itemset whereas if the min_util is* 100 *then* $\{c, e\}$ *is a low utility itemset.*

Definition 3. ***(Utility of a transaction in a database)*** *The utility of a transaction* T_d *is denoted as* $TU(T_d)$ *and defined as* $\sum_{i_p \in T_d} u(i_p, T_d)$.

In the example database, the *TU(T_1) is* $u(a, T_1)+u(c, T_1)+u(d, T_1) = 3+5+16 = 24$.

3.1 Problem Statement

Given a transaction database D consisting of n unique items, $I = \{i\_1, i\_2, ..., i\_n\}$, k user specified value, our aim is to find *top-k* high utility itemsets in streaming environment.

Mining high utility itemsets is challenging since high utility itemsets does not obey the *Anti-Monotone(downward closure)* property. Downward closure property states that an itemset X if is a low utility itemset, then all its supersets will also be a low

utility. We can prune the superset space using downward closure property. But since high utility itemsets do not have downward closure property, pruning search space becomes difficult. To overcome this drawback, the concept of *Transaction Weighted Utility, TWU* was introduced.

Definition 4. ***(Transaction Weighted Utility, TWU)*** *of an itemset X over a dataset is defined as sum of the transaction utilities of those transactions in which itemset X is present. TWU is denoted as : $TWU_D X = \sum_{X \subseteq T_i \wedge T_i \in D}(TU(T_i))$*

We, can clearly see that $u(X, T\_i) \leq TU(T\_i) \leq TWU_D X$. Also, TWU satisfies the downward closure property for high utility itemsets. If any itemset has utility less than minimum utility threshold, that itemset is considered to be low utility itemset. $TWU_D X$ is an overestimated of $u_D X$, therefore it cannot ignore any high utility itemset. It helps to prune out the search space of candidates using overestimation utility.

To tackle huge quantity of incoming data, we should design algorithms which focus on less memory consumption.Data Stream is a viable approach in which we stimulate the incoming data as fast speed streaming data grouped in batches. In this report, we are using *Sliding Window Model* technique to mine high utility itemsets from steaming data.

4

State-of-the-art algorithms

4.1 Top-K High Utility Itemset Mining over Data Streams (T-HUDS)

Here, we first define some of the definitions used in *Top-K High Utility Itemset Mining over Data Streams* paper by Zihayat et al. [18].

Definition 5. ***(Prefix Utility of an Itemset in a Transaction)*** *Assume all items in the transaction are sorted in lexicographic order and an itemset $X \subseteq T$. The prefix set of the X in T denoted as prefixSet(X, T) and is defined as the set of items that do not contain any item from transaction T greater than the last item in the itemset X. Thus, summed utility of items in prefix set results in prefix utility of an itemset X in transaction T.*

$$PrefixUtil(X,T) = \sum_{i \in prefixSet(X,T)} u(i,T)$$

Consider the example database 3.1, we have all transactions sorted in lexicographic order. For an itemset $\{a, e\}$, the prefix set of itemset in transaction T_2 is $\{a, c, e\}$. The Prefix utility is $u(a, T_2) + u(c, T_2) + u(e, T_2) = 6 + 30 + 8 = 44$.

Definition 6. ***(Prefix Utility of an Itemset X in a Database D)*** *is defined as $PrefixUtil_D X = \sum_{X \subseteq T_i} PrefixUtil(X, T_i)$.*

In the example given above, the $PrefixUtil_D\{a, c, e\} = u(ace, T\_2) + u(ace, T\_3) = 44 + 58 = 102$.

The Prefix Utility is an overestimated utility but has better bound over Transaction Weighted Utility(TWU). TWU is a highly estimated utility which results in the generation of a huge number of candidates. Therefore, to reduce the running time for the second phase of tree based algorithms, the concept of Prefix Utility was adopted. We can derive a relationship between TWU, PrefixUtil and exact utility of an itemset X as described below,

$TWU_D X \geq PrefixUtil_D X \geq u_D X$

Definition 7. ***(Minimum Transaction Utility (mtu))*** *of a transaction T is defined the least utility among all the utilities of items present in the transaction T. mtu is defined as: $mtu(T) = min_{i \in T}(u(i, T))$.*

For example, in Fig 3.1:
$mtu(T_1) = min(u(a, T_1), u(c, T_1), u(d, T_1)) = min(3, 5, 16) = 3$

Definition 8. ***(Minimum Transaction Utility (MTU))*** *of an itemset X over a data set D is defined as the sum of the minimum transaction utility(mtu) of the transactions in which itemset X is present.*
$MTU_D X = \sum_{X \subseteq T \wedge T \in D} mtu(T)$
As an example, consider itemset X be $\{a, c\}$, then $MTU(\{ac\}) = mtu(T_1) + mtu(T_2) + mtu(T_3) = 3 + 6 + 3 = 12$

Thus, for any itemset X in a data set $D, MTU_D X \leq u_D X$ relationship holds true, i.e, the minimum transaction utility of an itemset in a dataset is always less than or equal to the utility of the same itemset in dataset.

T-HUDS [18] is a compressed tree based algorithm for mining *top-k* high utility itemsets over data stream without specifying minimum utility threshold. It internally uses *HUDS* algorithm for finding *HUIs*. The *T-HUDS* algorithm processes the current data with three stages: 1: HUDS Construction ,2: HUDS Mining and 3: HUDS Update. Each non-root tree node of tree data structure represents an item in a transaction. The path from a root to the node represents the transactions containing the item. A non-root node contains the following fields: nodeName, nodeCounts, nodePUtils, nodeMTUs, link to the child nodes and link to the parent node. *nodeName* describes the name of the item, *nodeCounts* is an array of size equal to window size and is defined as the number of times the item appeared in the transactions from root to node,

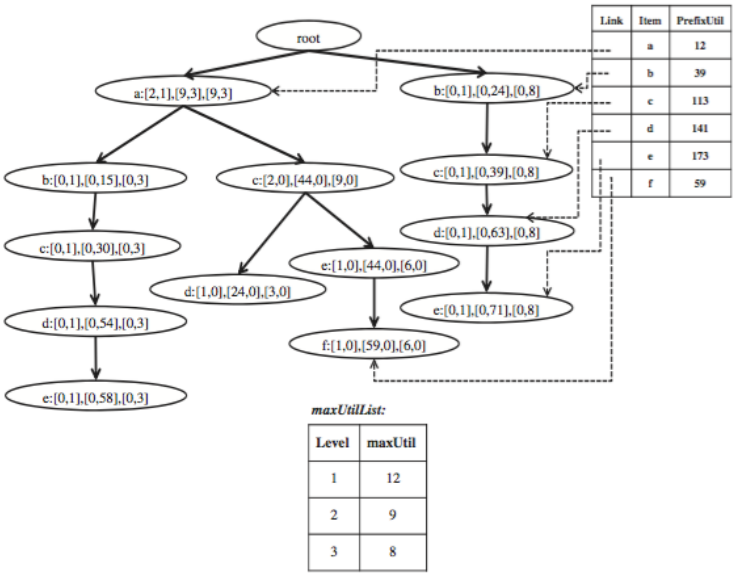

Figure 4.1: HUDS-Tree after inserting transactions in SW_1

nodePUtils is an array which represents the prefix utility of each node falling onto the transaction path, *nodeMTU* represents an array of minimum transaction utility of item node ,link to child nodes are the pointers to child of the item node and link to parent node is the pointer to the parent node. The information in *HUDS* tree is propagated in Batch form since it is easy to add a new batch once a single window is processed and to delete the oldest batch from the window.

Definition 9. ***(Maximum Utility List, (MaxUtilList))*** *is a list of length d where, d is the depth of the tree.*

MaxUtilList is defined by the formula:
$maxUtilList = \{maxUtil_1, maxUtil_2, maxUtil_3, ..., maxUtil_d\}$, *where* $MaxUtil_i$ *is the maximum utility among nodes at level i of the HUDS tree and is defined as:*

$$MaxUtil_i = max_j\{max(minProfit(node_{i,j}) * nodeCount(node_{i,j}), nodeMTU(node_{i,j}))\}$$

, where $node_{i,j}$ *denotes the* j^{th} *in* i^{th} *level of the HUDS tree. Also,* $nodeCount(node_{i,j})$ *denotes the sum of the count of items in all the batches falling in path from root to node in different transactions, whereas,* $minProfit(node_{i,j})$ *depicts the least profit value(external value) from root to* $node_j$*.* $nodeMTU(node_{i,j})$ *denotes the sum of the*

minimum transaction utility for all the batches of $node_j$ *formed by different transactions in which* $item_j$ *is present along the path from root to node. Although the definition of maxUtil fetches the maximum value, but it represents the maximum value among the minimum values among the nodes at* $level_i$*. Therefore,* $maxUtilList_i$ *value at level i can be considered as the candidate utility to represent minimum utility threshold which need to be raised from zero in order to prune search space during mining process.*

Consider the example database, in figure 4.1, the maxUtil of level 1 will be:
$max\_a = max_a(minProfit(node_{1,a}) * nodeCount(node_{1,a}), nodeMTU(node_{1,a}))$ *and*
$max\_b = max_b(minProfit(node_{1,b}) * nodeCount(node_{1,b}), nodeMTU(node_{1,b}))$.
$MaxUtil_1 = max\{max\_a, max\_b\}$.
Implies, $MaxUtil_1 = maxmax_a(3 * 3, 12), max_b(1 * 6, 8) = max(12, 8) = 12$.
Similarly, maxUtil for all the levels can be calculated and is shown upto level 3 in 4.1.

Definition 10. ***(Minimum Item Utility of an item)*** *x in a database D is defined as:* $miu_D x = \{u(x, T_q) \mid x \in T_q and x \neq \exists T_p such that u(x, T_p) < u(x, T_q)\}$.

The definition implies that the utility of x should be least among all the transactions in consideration. In the example database, the $miu_D a = 3$.

Definition 11. ***(Minimum Itemset Utility)*** *of an itemset X is the product of the sum of the minimum utility of the itemset X over the whole database and the support count of itemset X. The support count indicates the count of itemset X appearing in the database.*

$MIU_d X = \sum_{a_i \in D} MIU_D a_i * SC_D X$ *where,* $SC_d X$ *is the support count of itemset X in database D.*

Definition 12. ***(Minimum Itemset Utility List)*** *denoted by MIUList is the top-k list of Minimum Itemset Utilities of itemsets sorted in descending order.*

$Thus, MIUList = \{MIU_1, MIU_2, ..., MIU_k\}$ *where,* $MIU_1 > MIU_2 > MIU_3, ..., MIU_k$.
In HUDS implementation, initially, for the first window, MIU of single items will be inserted in the MIUList. With the formation of itemsets of greater size, MIUList of size k will be updated.

The three phases of *HUDS* algorithm are described below:

The *T-HUDS* algorithm proposed by Zihayat et al. [18] is described in Algorithm 4.2. Below we describe how the HUDS-tree constructed, mined and updated in streaming scenario.

HUDS-Construction: Figure [18] shows the *HUDS* tree after inserting transactions in the first window. Each transaction in the data set is sorted in lexicographic order. Each item of the transaction is a node in the HUDS-tree with the path from the root to node denotes the transaction or lexicographic set in the transaction. The transaction data is added to the tree in batch form. Different transactions are grouped to form *batch* of transactions and different batches grouped to form *window* of batches. When the first window is filled with $window\_size$ number of batches, the algorithm forms the HUDS-Tree and proceeds for mining high utility itemsets. Other windows are updated with every new batch entry and deletion of oldest batch from window. For a parent node p and child node c, if c is already an existing child of p, the following fields are updated for batch x in window: nodeCount for batch x is increased by 1, nodePUtil for batch x is increased by current node prefixUtil, and nodeMTU for batch x is increased by current transaction *mtu*. Otherwise, child c is added to parent with nodeCount set to 1, nodePUtil set to child's prefix util and nodeMTU set to current child's *mtu*. Also, *maxUtilList* is updated if the current node's utility at *Kth* depth of the tree is maximum. *MIU-List* of size k is updated with *miu* values of single size items. A *Header Table* is constructed to keep track of each item node globally in lexicographic order by keeping node's global prefixUtil and link to the list of other nodes with the same name. After processing the first window, every time windows are processed to mine *top-k HUIs* with the insertion of each new batch and deletion of the oldest batch from the window. The utility of *Kth* itemset denoted as *minTopKUtil* is used as the *minimum utility threshold* for the next window. From second window onwards, the tree is not built from scratch instead it is updated with current batch values. Also, the nodes which do not occur in the current window, are deleted and their entries are also updated in header table. The process is repeated until the whole data is not explored.

Figure 4.1 shows the HUDS-Tree, maxUtilList and header table construction. For each node with a node name, we have the first array describing the item count per batch, the second array describes the item prefix utility along the path from the root

Algorithm 1 $T - HUDS$ algorithm

SubRoutine1: Top-K HUI Mining
SubRoutine2: HUDS-tree-Update
Input: Batch B_i, k, *window_size*, *HUDS-tree*
Output: Complete set of top-k high-utility itemsets.

1: **if** HUDS-tree is empty (i.e., B_i is the very first batch B_1) then **then**
2: $minTopKUtil\_0 < -0$
3: Construct a HUDS-tree based on B_i (i.e., B_1)
4: Construct the *maxUtilList* based on the information in the $HUDS - tree$
5: Initialize the *MIU-List* using the *top-k* miu values of the items
6: **else**
7: Call **HUDS-tree-update** to update HUDS-tree, maxUtilList and MIUList using B_i and *window_size*
8: **end if**
9: **if** batch ID $i \geq$ *window-size* then **then**
10: Call **Top-K HUI Mining** to compute *top-k* HUIs on the current sliding window with the *HUDS-tree, maxUtilList, MIUList,* $minTopKUtil_{i-1}$
11: **end if**
12: **return** $Top - k$ HUIs

Figure 4.2: $T - HUDS$ algorithm

to node, whereas the third array describes the item minimum transaction utility along the path from the root to node. The header list contains each item in lexicographic order with prefix utility information over the whole window. For each item, header table contains a pointer to the node first arrived in the window which in turn linked to other nodes with the same item name. MaxUtilList of size d contains the maxUtil for each level i of the HUDS-Tree with depth d.

HUDS-Mining: To mine the high utility itemsets, since the user only provides the information of a number of high utility itemsets to explore and not the minimum utility threshold, a mechanism is described to increase the minimum utility threshold from zero at the start of the algorithm to the utility of *Kth* high utility itemset. This is done to avoid generation of a huge number of candidates. Initially, the *minimum threshold* is kept to 0. Before mining, the minimum of $maxUtil_k$, $minTopKUtil$ and $MIUList_k$ values is updated as *minimum threshold* value. The *MIU-List* contains the *top-k* utilities of single size items sorted in decreasing order. If the new itemset appears to have more utility than the *Kth* value of the *MIU-List*, that itemset is said to be a high utility itemset and utility is updated in *MIU-List* with deletion of lowest utility present. The header list is traversed from bottom to top and each item's pointer to linked list is explored. A local tree for each item is formed depicting all the transactions

in which item is present. Recursively, the local tree is explored and high utility itemsets are mined. After processing the item, the global header table is processed similarly for other items in the bottom to up direction.

HUDS-Updation: Since *HUDS-Tree* is a two-phase algorithm with phase one describing the construction and mining high utility candidates. Phase two starts with the verification of mined high utility candidates. The data set is scanned again to verify the high utility candidates by calculating their exact utility in the whole dataset. The *top-k* high utility itemsets are identified and *Kth MIU-List* utility is set as value of *minTopKUtil*. The tree is updated with a new batch and the oldest batch information is deleted. Also, all those nodes which become null and are not present in current window data, are deleted.

4.2 Faster High Utility Itemset Mining (FHM)

Fournier-Viger et al. [6] proposed a vertical mining algorithm *Faster High Utility Itemset Mining* as an optimisation to *HUI-Miner* [9] algorithm to reduce the number of join operations of different utility lists. It identifies the item co-occurrences to avoid the costly join operations. It defines a novel data structure *Estimated Utility Co-Occurrence Structure, EUCS*. It is a hashmap of hashmap of float values of the form $\{a, b, c\}$ where c is a float value and $c \neq 0$. This structure stores the items co-occurring with other items and this structure greatly reduce the memory size.

Before switching to algorithm, below we define terms relevant to the algorithm:

Definition 13. ***(Utility-List)*** *Let $\succ$ be the total order defined on items from I. The utility list of an itemset x is the information stored as tuple of the form $(tid, iUtil, rUtil)$ where T_{tid} is the transaction id in which itemset x appears, $iUtil$ is the exact utility of the itemset in current tid as $x_{iUtil} = u(x, T_{tid})$. $rUtil$ is the remaining utility of all the itemsets after x such that $x_{rUtil} = \sum_{\{i|i \in T_{tid} \wedge i \succ x\}} u(i, T_{tid})$.*

Consider the example database, if the current itemset is d, the utility list for item $utilList(d) = \{(T_1, 16, 0)(T_3, 24, 4)(T_4, 24, 8)\}$. The utility list for itemset $\{d, e\}$ is $utilList(d, e) = \{(T_3, 28, 0)(T_4, 32, 0)\}$.

Algorithm 1 *FHM* algorithm

SubRoutine: Search(an itemset, set of extension of itemset, minUtil, EUCS structure)
Input: D: a transaction database, minutil: a user specified threshold
Output: the set of high-utility itemsets

1: Scan D to calculate the TWU of single items;
2: $I* <-$ each item i such that $TWU(i) > minutil$;
3: Let $\succ$ be the total order of TWU ascending values on $I*$;
4: Scan D to built the utility-list of each item $i \in I*$ and build the EUCS structure;
5: Search$(\emptyset, I*, minUtil, EUCS)$ to recusively search for all the extensions of $I*$ in EUCS structure. It prunes out the extension of $I*$ with low utility than minUtil.

Algorithm 1 shows the *FHM* algorithm as stated in paper [6].The algorithm works with two database scans. In the first database scan, the algorithm calculates the *Transaction Weighted Utility, TWU* of each item present in the database. The minimum utility is provided by the user. The items having TWU less than the minimum user threshold, are not considered. The potential high utility items are sorted by ascending order of TWU. The FHM algorithm works in a breadth first search manner by generating pair, triplets, quads etc. The algorithm again scans the database to form utility list of single items having *twu* greater than the minimum utility. Thus, items with low *twu* values will be pruned out in the first step. For an item, total *iUtil* and total *rUtil* in the database along with utility list of the item containing tuples is encapsulated. The utility lists of single items are joined to form utility lists of items of size two. Similarly, the algorithm proceeds to form utility list of items of greater length. The naive method to form utility lists of greater size is to explore the whole search space but this method is time and space consuming. Therefore, in order to prune out the search space initially itemsets *iUtil* for all tuples was checked w.r.t minimum utility passed by the user. If itemset *iUtil* is less than minimum utility, that itemset will not be considered as high utility itemset and will be pruned out. Also, itemset's sum of *iUtil* and *rUtil* for all tuples was compared with minimum utility and if found to be less than minimum utility, that itemset's extensions will not be considered as high utility itemsets. The extension of an item is determined by EUCS structure. The EUCS structure is built using a hashmap of hashmap of float values. It contains the items having a set of possible extended items with their *twu* values.The search procedure is called with the current value of $I*$ as *null*. The search procedure finds out the extensions of itemsets

in EUCS recursively.

The itemsets are pruned based on two pruning techniques:

1: $Sum(X.iUtil \geq minUtil)$: If an itemset X has sum of $iUtil$ values of all tuples in the utility list is less than the $minUtil$, that itemset is said to be a low utility itemset else the itemset is said to be a high utility itemset.
2: $Sum((X.iUtil + X.rUtil) \geq minUtil)$: If itemset X has sum of $(iUtil, rUtil)$ of all tuples in the utility list is less than $minUtil$, that itemset will not be considered for further exploration since the utility of the item along with remaining utility is not enough to make the itemset and its descendants a high utility itemsets. Else the itemset X is further explored using EUCS to get the high utility itemsets.

4.3 Efficient High Utility Itemset Mining (EFIM)

Zida et al. [13] proposed an efficient one phase algorithm based on static transaction database. It is efficient in terms of memory and time. *EFIM* defines the two upper bounds named *sub-tree utility* and *local utility* to effectively prune the search space. To calculate the upper bounds in linear space and time, it also made use of an array based utility counting technique named *Fast Utility Counting*. In order to reduce the size of the dataset, it comes up with the strategy of *database projection* and *transaction merging* to reduce the cost of multiple database scans.

Below we define few terms used in the *EFIM* paper:

For a set of items I, the possible search space of itemsets that can be produced is 2^I and can be represented by *set-enumeration* tree. The itemsets are ordered in total order $\succ$ which is represented by lexicographic order or order of increasing TWU. Figure 4.3 shows the set-enumeration tree of itemset $\{a, b, c, d\}$ where total order $\succ$ is in lexicographic order. Starting from the root, an itemset α is explored and each time a single item is added to α in $\succ$ order.

Definition 14. ***(Items that can extend an itemset)*** *Let α be an itemset and $E(\alpha)$ represents the set of all items that can be used to extend α in depth first search, i,e,*

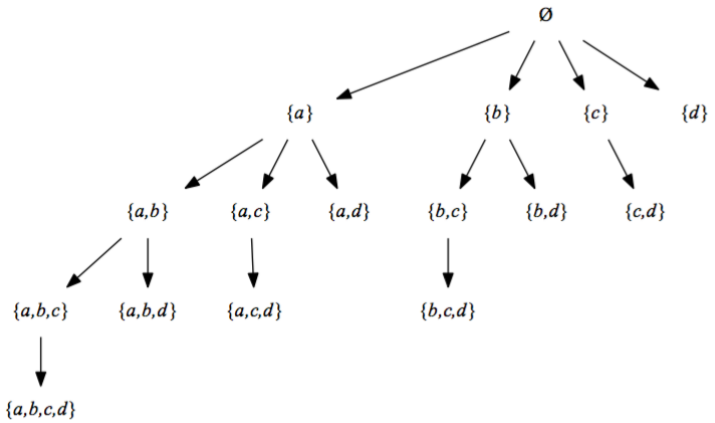

Figure 4.3: Set-enumeration tree for $I = \{a, b, c, d\}$

$E(\alpha) = \{z \mid z \in I \wedge z \succ x, \forall x \in \alpha\}$

Consider the example database in Figure 3.1, we have all the transactions ordered in lexicographic order. Also assume total order $\succ$ is according to lexicographic ordering. Therefore, if c is an itemset then, the set of items for extension of c are $E(c) = \{d, e, f\}$.

Definition 15. ***(Extension of an Itemset)*** *Let α be an itemset and Z be an extension to α which appears to be the sub-tree of α in set-enumeration tree. If $W \in 2^{E(\alpha)}$ and $W \neq 0$, then extension of α is $Z = \alpha \cup W$. If W is a single item to be appended to α, then extension of α is $Z = \alpha \cup \{z\}$ for an item $z \in E(\alpha)$.*

Consider, the same example dataset, for transaction T_2 and let c be the itemset to be extended, the possible extensions of itemset c is $\{c, e\}$, $\{c, f\}$, $\{c, e, f\}$.

In order to reduce the cost of database scans, there is a need to reduce the size of the database. *EFIM* defines a new concept of database projection, i.e, all those items which are not the extensions of an itemset, should not be considered and should be removed from the database. This greatly reduces the size of the database.

Definition 16. ***(Projected Database)*** *Projection of a Transaction T using an itemset α is denoted by: $\alpha - T = \{i \mid i \in T \wedge i \in E(\alpha)\}$. Similarly, the Projection of a Database D using an itemset α is defined as: $\alpha - D = \{\alpha - T \mid \alpha - T \in D \wedge \alpha - T \neq \emptyset\}$.*

Consider the example database, for an itemset b the projected transactions $\alpha - T$ *are* $T_3 = \{c, d, e, f\}$, $T_4 = \{c, d, e\}$, $T_5 = \{c, e, g\}$ *and* $T_6 = \{c, e, g\}$.

Some transactions in a dataset are similar to each other and thus those transactions can be combined in order to reduce the size of the data set. Transactions are similar to each other if every element of both transactions is same but their internal value can be different. Thus, identical transactions can be merged to form a single new transaction having an internal value of each item equivalent to the sum of internal values of corresponding items in different identical transactions.

Definition 17. ***(Transaction Merging)*** *Transaction Merging implies the merging of identical transactions* $Tr_1, Tr_2, Tr_3, ..., Tr_m$ *in a database D by a single transaction* $T_M = T_1 = T_2 = T_3 ... = T_m$ *with the quantity of each identical item* $i \in T_M$ *is defined as* $quantity(i, T_M) = \sum_{k=1...m} quantity(i, Tr_k)$.

Projected transaction Merging implies merging the projected transactions to form a merged projected database. Since transaction merging reduces the size of the database, merging the projected transactions greatly reduces the size of transactions since projected transactions are themselves smaller than original transactions.

Definition 18. ***(Projected Transaction Merging)*** *implies merging the set of identical projected transactions* $Tr_1, Tr_2, Tr_3, ..., Tr_m$ *into a single new merged transaction* $Tr_M = Tr_1 = Tr_2 = ... = Tr_m$ *in a database* $\alpha - D$. *The internal value of each item* $i \in T_M$ *is defined as* $quantity(i, T_M) = \sum_{k=1...m} quantity(i, Tr_k)$.

Definition 19. ***(Sub-tree utility)*** *If* α *be the itemset and z be an extension to* α *in depth first search manner, the subtree utility of* $z = su(\alpha, z) = \sum_{T \in \{\alpha, z\}} u(\alpha, T) + u(z, T) + rem_{i \in T \wedge i \succ z} u(i, T)$.

Consider the example database, for transaction Tr_5 if b is the itemset and e is the extension to b, $su(b, e) = u(b, Tr_5) + u(e, Tr_5) + u(f, Tr_5) = 12 + 4 + 6 = 22$

Definition 20. ***(Local Utility)*** *If* α *be the itemset and z be an extension to* α, *the local utility of* $z = lu(\alpha, z) = \sum_{T \in \{\alpha, z\}} u(\alpha, T) + rem_{i \in T \wedge i \succ \alpha} u(i, T)$.

Consider the example database, for transaction Tr_5 if b is the itemset and e is the extension to b, $lu(b, e) = u(b, Tr_5) + u(c, Tr_5) + u(e, Tr_5) + u(f, Tr_5) = 12 + 10 + 4 + 6 = 32$.

The above definitions of *sub-tree utility* and *local utility* clearly shows the relationship between them and *transaction weighted utiilty, TWU*. For an itemset $X = \alpha \cup \{z\}$, $TWU(X) \geq lu(\alpha, z) \geq su(\alpha, z)$

Definition 21. ***(Primary and Secondary Itemsets)*** *: For an itemset* $Y = \{\alpha \cup z\}$*.Primary Itemsets are those items denoted by* $Primary(\alpha) = \{z \mid z \in E(\alpha) \wedge su(\alpha, z) \geq min\_util\}$*. Whereas, secondary itemsets are denoted as* $Secondary(\alpha) = \{z \mid z \in E(\alpha) \wedge lu(\alpha, z) \geq min\_util\}$*.*

Again, we can depict the relationship between Primary and Secondary itemsets. Primary itemset is a subset of secondary itemset and is denoted as: $Primary(\alpha) \subseteq Secondary(\alpha)$*.*

Algorithm 2 shows the *EFIM* working proposed by Zida et al. [13]. *EFIM* works with storing each item in an array and calculating the TWU of each item i. Initially, the local utility $lu(\alpha, i)$ of an item i is equals to the TWU of the item in the database. Initially, the value of α is $\emptyset$. The *TWU*, *local utility* and *sub-tree utility* of item i is stored in array of size I at an index i. This is done to fetch the item data in constant time. The items having local utility greater than the minimum utility threshold provided by the user are considered as potential high utility itemsets. Database projection occurs by removing all low utility itemsets from the database identified in the previous step. All transactions are sorted in order by size. The empty transactions with size zero are identified in the beginning of the data set and thus can be removed easily. All the items in the transactions are sorted in ascending TWU order and the items are explored in depth first search manner. The algorithm proceeds with calculating the subtree utility, $su(\alpha, i)$ of potential high utility itemsets. If the $su(\alpha, i) \leq minutil$, those itemsets are considered as low utility itemsets and can be pruned out. The itemsets which passed the criteria of subtree utility are considered high utility itemsets and are further explored in *Seach* method. The Search method works again with calculating the local utility, subtree utility with database projection. The procedure is recursed in above manner till all the high utility itemsets are not explored.

Algorithm 2 *EFIM* algorithm

SubRoutine: Search

Input: D: a transaction database, minutil: a user specified threshold

Output: the set of high-utility itemsets

1: $\alpha = \emptyset$
2: Calculate $lu(\alpha, i)$ for all items $i \in I$ by scanning D, using a utility-bin array;
3: $Secondary(\alpha) = \{i \mid i \in I \wedge lu(\alpha, i) \geq minutil\}$;
4: Let $\succ$ be the total order of TWU ascending values on $Secondary(\alpha)$;
5: Scan D to remove each item $i \notin Secondary(\alpha)$ from the transactions, and delete empty transactions;
6: Sort transactions in D according to $\succ_T$;
7: Calculate the sub-tree utility $su(\alpha, i)$ of each item $i \in Secondary(\alpha)$ by scanning D, using a utility-bin array;
8: $Primary(\alpha) = \{i \mid i \in Secondary(\alpha) \wedge su(\alpha, i) \geq minutil\}$;
9: $Search(\alpha, D, Primary(\alpha), Secondary(\alpha), minutil)$ to recursively explore bigger size itemsets;

5

Implementation of State-of-the-art algorithms in Streaming Environment

In this section we defined our variants - *Stream-FHM* and *Stream-EFIM*, the top-k versions of state-of-the-art algorithms *FHM* and *EFIM* respectively. The static transaction based state-of-the-algorithms *Stream-FHM* and *Stream-EFIM* were compared with the dynamic transaction dataset based algorithm *T-HUDS*. Keeping in view the heavy requirement of streaming algorithms in today's world, we studied the effect of streaming environment over these algorithms. The performance of trio is recorded and is discussed in chapter 6.

5.1 Top-K Fast High Utility Itemset Mining over Data Stream ($Stream - FHM$)

We applied the top-k version of *FHM* algorithm naively on each window. Unlike in *T-HUDS*, where information of *minTopKUtil* is propagated to next window and data structure was retained with minimal updation of batches, the *minTopKUtil* will not be propagated to next window since we tried to study the effect of applying each static transaction based algorithms in streaming environment only. Also, all the utility lists in FHM algorithm will be formed from the scratch.

Algorithm 3 shows the working of *Stream-FHM* algorithm. The algorithm works with the limited size of data equivalent to the size of the window. The input provided to the algorithm is the set of values $window_j, K - value, win\_size$. Each window contains batches whereas each batch contains a group of sequenced transactions. Once the window is filled with win_size number of batches, the first window will be processed. The window is scanned and the TWU of each item is calculated. Figure 5.3 shows the TWU of items in window SW_1. Items having the TWU less than the *minTopKUtil* are low utility itemsets and will be pruned out. Initially, the *minTopKUtil* value is zero, therefore all items will be potential high utility itemsets. The items are arranged in ascending TWU order. We can use lexicographic ordering also.

Table 5.1: *UtilityList of item a in* SW_1

TID	EU	RU
1	3	21
2	6	53
3	3	55

Table 5.2: *UtilityList of item b in* SW_1

TID	EU	RU
3	12	43
4	24	47

Table 5.3: *TWU of items in* SW_1

Item	a	b	c	d	e	f
TWU	162	182	244	153	220	112

Table 5.4: *UtilityList of itemset ab in* SW_1

TID	EU	RU
3	15	43

The window is again scanned to create the utility-list structure for all the batches of each item. Utility-List of an item is the information stored for each item in the form

Algorithm 3 $Stream - FHM$ algorithm

SubRoutine: FHM

Input: $Window_j$, user defined k value, win_size

Output: the set of high-utility itemsets

1: Set the $minTopKUtil < -0$
2: Scan $Window_j$ to calculate the TWU of single items;
3: $I* < -$ each item i such that $TWU(i) > minTopKUtil$;
4: Let $\succ$ be the total order of TWU ascending values on $I*$, transactions are sorted in $\succ$ order.
5: Scan the window to build the utility-list of each item for every $batch_i$ in $window_j$ having $u(i) \geq minTopKUtil$.
6: Build the EUCS structure containing for every item i, its possible extensions along with their twu values.
7: Intersect the utility lists of different items occurring in same batches and generate a single utility list.
8: Call FHM method to prune the itemsets having sum of utility less than the *minTopKUtil*. Add the itemset to *top-k* hui set and raise *minTopKUtil* if required.
9: Itemsets having $Sum(iUtil, rUtil) \geq minTopKUtil$ are further explored by calling FHM recursively.
10: **return** TopKHUI

of tuple $< tid, iUtil, rUtil >$ where tid is the transaction id in which item is present. $iUtil$ represents the exact utility of the item in the transaction and $rUtil$ represents the remaining utility of the item. Remaining utility of item x is the sum of the utility of all the items i where the order of items i is represented as $i \succ x$. The utility-list for each single item for every batch is formed. If we have a batch size of two and window size of two, then the utility-list structure for item a can be shown in 5.1. For batch B1, item a is present in two transactions tid_1 and tid_2 with tid_1 having exact utility as 3 and remaining utility as 21. Similarly, the utility-list of item b is shown in figure 5.2. Thus, for every item, the utility-list will be formed to begin the mining process. But to start with mining process, the value of $minTopKUtil$ should be increased from zero. Thus, single items having *top-k* TWU will be inserted in the *top-k* buffer and the buffer is sorted in descending order. The $minTopKUtil$ threshold is set to k^{th} value in *top-k* buffer. After raising the threshold, mining process can begin by calling *FHM* in line *8*.

FHM explores the search space in depth first search manner. If we consider the naive method, then whole search space should be explored with the generation of every

possible itemset. But this method has a very high space and time complexity. Thus, to provide efficiency to the algorithm, we need to prune out some itemsets which will be low utility itemsets and are not of importance to us. Therefore, the strategy of non- duplication of single items as depicted in [9] will be adopted. Since single items are already present, we will avoid re-insertion of single itemsets. The algorithm first generates the pair by intersecting the utility-lists of single items. Consider the figure 5.4, shows the intersection of utility list of a and *b*. When the itemset is generated, the algorithm inserts the itemset in the *top-k* buffer if the sum of the utility of itemset in all tuples is no less than the utility of Kth item in the *top-k* buffer. If the utility of Kth item in *top-k* buffer is greater than the $minTopKUtil$, then the $minTopKUtil$ is updated to Kth item utility. Also, the algorithm prunes out the itemset and its superset if the sum of exact utility and remaining utility of itemset in all tuples containing the itemset is less than *minTopKUtil*. The algorithm recursively works in a similar manner and returns the set of *top-k* high utility itemsets for current window.

After processing the single window, the algorithm does not consider any input from the previous window. The oldest batch is removed from the window and a single new batch is added to the window. For every new window, the algorithm runs from scratch with the creation of utility-lists for every item present in the database.

5.2 Top-K Efficient High Utility Itemset Mining over Data Stream (Steam-EFIM)

We implemented the state-of-the-art EFIM algorithm in streaming environment without passing user defined threshold. The *top-k* version of EFIM was implemented naively by applying algorithm in streaming context on every window. Every time the window updated, algorithm builds the array data structure from the scratch. We do not make use of *minTopKUtil* to raise the initial minimum threshold from second window onwards since we focus on the performance measured by state-of-the-art algorithms in streaming context.

Algorithm 4 shows the working of the $Stream - EFIM$ algorithm. The algorithm starts by filling window and processing the data to mine high utility itemsets. When

Algorithm 4 $Stream - EFIM$ algorithm

SubRoutine: EFIM
Input: $Window_j$, user defined k value, win_size
Output: the set of top-k high-utility itemsets

1: Set the $minTopKUtil < -0$
2: $\alpha = \emptyset$
3: Scan the $window_j$ to calculate $lu(\alpha, i)$ for all items $i \in I$ using a utility-bin array;
4: $minTopKUtil$ = utility value of K^th item in $lu\_utility$ bin array;
5: $Secondary(\alpha) = \{i \mid i \in I \wedge lu(\alpha, i) \geq minutil\}$;
6: Let $\succ$ be the total order of TWU ascending values on $Secondary(\alpha)$ for $window_j$;
7: Scan the window to remove each item $i \notin Secondary(\alpha)$ from the transactions, and delete empty transactions;
8: Sort the transactions in $window_j$ according to $\succ_T$;
9: Calculate the sub-tree utility $su(\alpha, i)$ of each item $i \in Secondary(\alpha)$ by scanning $window_j$, using a utility-bin array;
10: $Primary(\alpha) = \{i \mid i \in Secondary(\alpha) \wedge su(\alpha, i) \geq minutil\}$;
11: Call EFIM recursively to explore bigger size itemsets; Put *top-k* itemsets in $topKHui$ buffer;
12: Raise the $minTopKUtil$ if K^{th} utility value in topKHui buffer is greater than minTopKUtil;
13: **return** TopKHUI

the first window is filled by $window\_size$ number of batches, the algorithm proceeds to calculate the *twu* of each item present in database D. Value of $minTopKUtil$ is set to zero. For the first window since we have $\alpha = \emptyset$, local utility bin arraylist data structure is filled with the TWU of each item x at index x while exact utility of each item in window is stored in exact utility bin array at index x. Exact utility bin is sorted by descending TWU values and the value of $minTopKUtil$ is set to the utility of K^{th} item in exact utility bin array. All items of local utility bin array are traversed and are pruned out if local utility or *twu* is less than the minTopKUtil. The potential high utility itemsets are stored in an array and are sorted in increasing *twu* order. Low utility itemsets which are pruned in previous steps are removed from the window data. After removing the low utility itemsets, there can be a possibility of finding empty transactions in the window data. The window database is sorted in $\succ_T$ order so that transactions can be easily merged. If *transaction merging* is activated, the transactions having same items but the different quantity of each item can be merged by a single transaction having a quantity of each item as the sum of the quantity of same items in different transactions. If even a single item differs in two or more transactions, those

transactions cannot be merged. The empty transactions are removed from the window data. Again the window data is scanned to calculate the sub-tree utility of each item x and if the subtree utility of item appears to be less than the $minTopKUtil$, then the item x will not be considered for further exploration since the sum of utility of item x with the utility of its extension z and remaining utility of extension z does not overcome the minTopKUtil.

The *Search* procedure takes α as its input along with $Primary(\alpha)$, $Secondary(\alpha)$ items and minTopKUtil threshold. The search procedure explores the expansion of α given by $\beta = \alpha \cup z$. It proceeds with calculating the local utility for expansion z and thereby projecting the database for z. If itemset's local utility appears to be more than minTopKUtil and itemset's size is greater than zero that itemset is considered to be high utility itemset. The itemset is added in *TopKHUI* buffer. If the utility of K^{th} itemset in *TopKHUI* buffer is greater than the *minTopKUtil*, minTopKUtil will be set to value of K^{th} itemset of TopKHUI buffer. Also, transaction merging is taken into account for projected database of z. Again the empty transactions are merged and sub-tree utility of the expansion z is calculated to determine the expansions of z. The itemsets having sub-tree utility no less than minTopKUtil will be considered for further exploration. The algorithm recursively calls *search* procedure until further items are not explored. The TopKHUI buffer containing *top-k* high utility itemsets are returned for current window.

5.3 Top-K High Utility Itemset Mining, (T-HUDS)

We implemented the *T-HUDS* state-of-the-art algorithm. Since original code was not available, we implemented the T-HUDS code from the scratch on same parameters as mentioned in [18] paper. Experiments have been conducted on sparse and dense datasets similar to conduction done by authors in the paper.

6

Experiments and Results

In this section, we compared the results of *T-HUDS*, *Stream-FHM* and *Stream-EFIM* algorithms. We conducted experiments on an Intel Xeon(R) CPU=26500@2.00 GHz processor with 16 GB free RAM and has JDK 1.6 installed on a Window 8.1 operating system. The datasets used for our experiments are listed in Table 6.1. All real datasets except ChainStore were obtained from FiMi repository Goethals and Zaki (2012) [7]. ChainStore dataset was obtained from NU-Minebench 2.0 repository Pisharath et al (2005) [12].

Table 6.1: *Characteristics of Real Datasets*

Dataset	**#trans**	**#items**	**Avg. length**	**Batch size**	**Window size**	**Type**
ChainStore	11,12,949	46,086	7.2	1,00,000	6	Sparse
Retail	88,162	16,470	10.3	10,000	3	Sparse
Accidents	3,40,183	468	33.8	50,000	3	Dense
Connect	67,557	129	43	10,000	3	Dense
Mushroom	8,412	119	23	1,000	3	Dense

Each item in the ChainStore data set has a quantity and price value associated with it whereas for other datasets quantity values are experimentally generated between 1 to 10 and price values follows a log-normal distribution in the range of 1 to 100.

The following sets of experiments were conducted:

1. Comparison among $T-HUDS$, $Stream-FHM$ and $Stream-EFIM$ for varying k
2. Comparison among $T-HUDS$, $Stream-FHM$ and $Stream-EFIM$ for varying *window size*

6.1 Varying k

In this section, we perform experiments by varying the user defined value K and compare the performance of *T-HUDS*, *Stream-FHM* and *Stream-EFIM* on different dense and sparse datasets. The value of K is varied between 100 to 900.

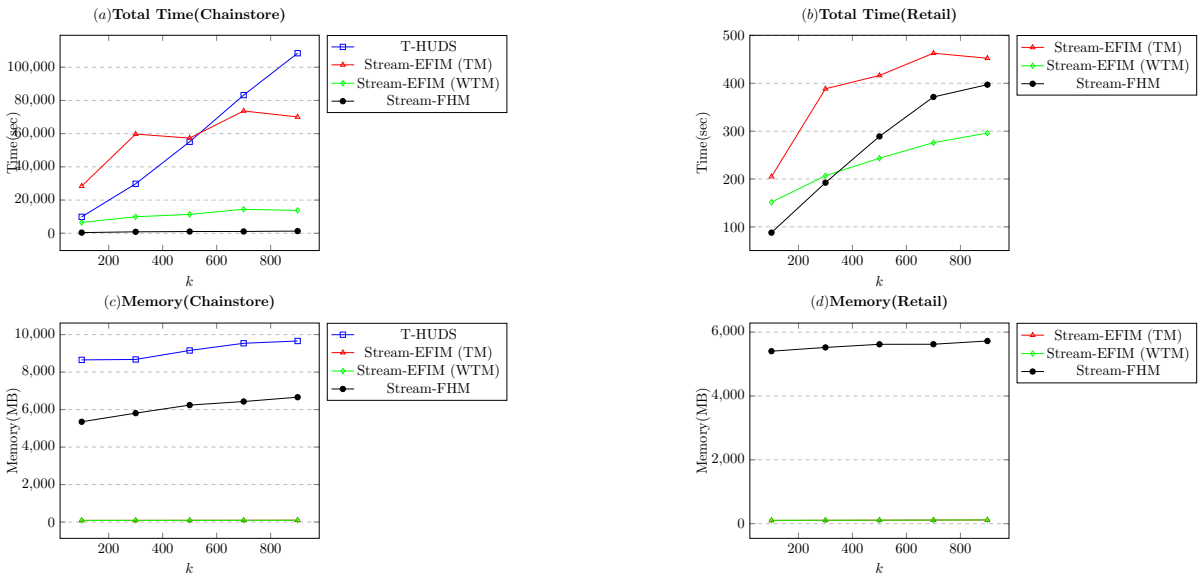

Figure 6.1: Performance evaluation on sparse datasets

Experimental results are shown in Figure 6.1 for Sparse Datasets. We have shown two versions of our variant $Stream-EFIM$ algorithm. One with transaction merging activated denoted by short hand TM and other without transaction merging activated denoted by WTM. The results on chainstore shows that streaming variants $Stream-EFIM$ (WTM) and $Stream-FHM$ perform 10 to 100 times faster than T-HUDS algorithm as both the variants do not generate any candidates. $Stream-FHM$ performs exceptionally well to T-HUDS whereas $Stream-EFIM$ (TM) improves its performance with an increase in the value of k since number of generated candidates in T-HUDS increases with value of k. $Stream-EFIM(WTM)$ is faster than its

complimentary $Stream - EFIM$ since time taken by transaction merging is not taken into account. Since $T - HUDS$ algorithm runs out of memory on the retail dataset, therefore we have shown the performance comparison between our variants only for the retail dataset. For Sparse dataset, $Stream - FHM$ and $Stream - EFIM(WTM)$ performs the best followed by $Stream - EFIM$ (TM) and T-HUDS. $Stream - FHM$ takes into account the utility-list structure and performs better since it did not engage itself in every time projecting database for every itemset and then perform transaction merging. Also, $Stream - EFIM$ (WTM) is faster than $Stream - EFIM$ (TM) since the time taken in transaction merging is reduced and thus greatly effects the running time of the algorithm.

Memory consumption shows that $Stream - EFIM$ takes the least memory followed by $Stream - FHM$ and then T-HUDS. $Stream - EFIM$ since works only on array data structure, do not consume much memory and produces a huge difference in memory consumption by other two algorithms. Also, $Stream - EFIM$ (WTM) takes almost similar but more memory than $Stream - EFIM$ (TM) since it do not take transaction merging into account. In sparse dataset, since most of the transactions are dissimilar to each other, therefore both versions of $Stream - EFIM$ have almost same space complexity. Since, $Stream - FHM$ also do not generate a huge number of intermediate itemsets and uses utility list data structure which does not consume much space as compared to the tree data structure, the memory consumed by $Stream - FHM$ is less than a T-HUDS algorithm.

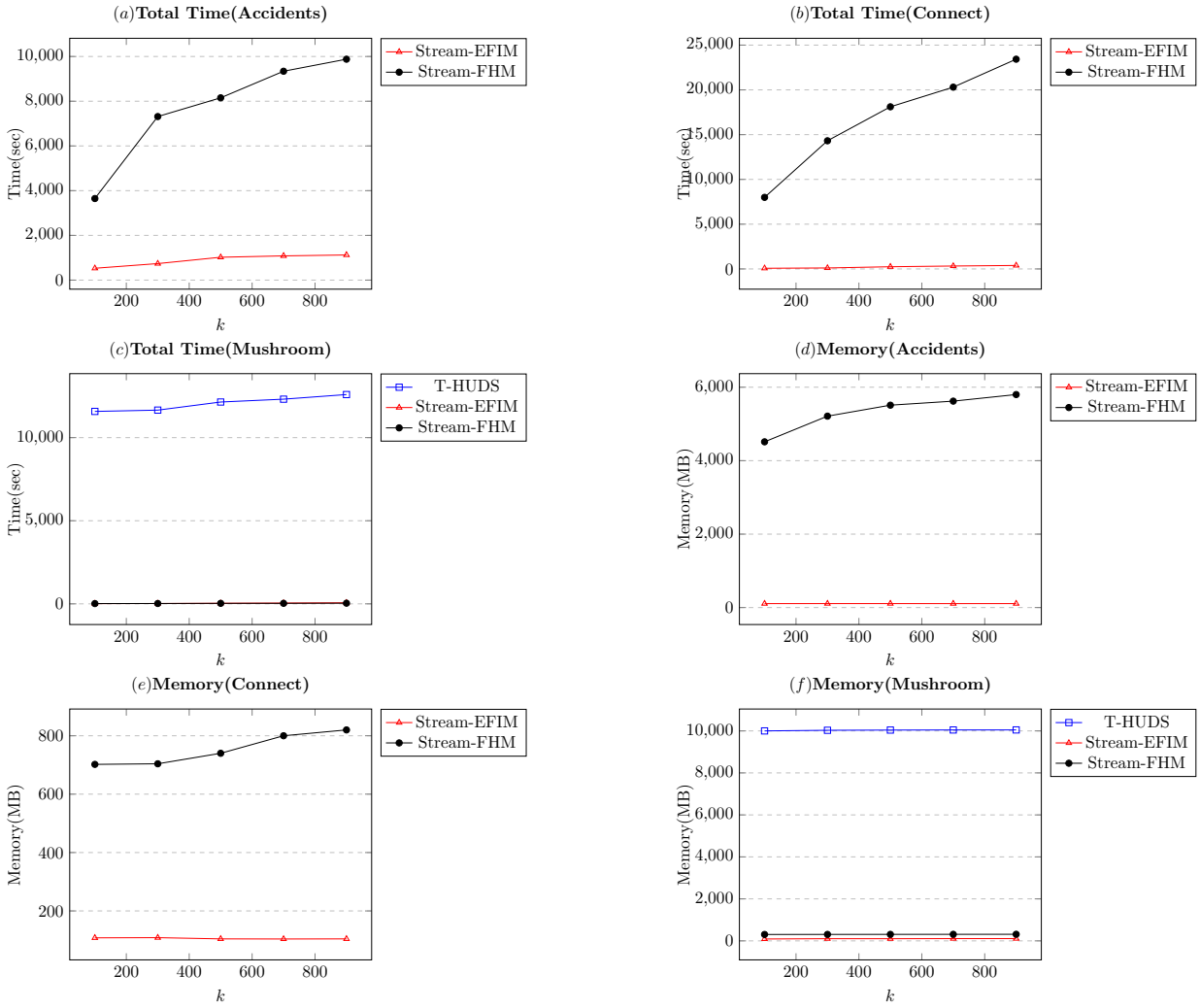

Figure 6.2: Performance evaluation on dense datasets

Experimental results for Dense dataset are shown in Figure 6.2. Except Mushroom, T-HUDS ran out of memory for all other dense datasets. $Stream - EFIM$ performs exceptionally faster than both the other algorithms. The overall result shows that our variants perform 4 to 20 times faster than T-HUDS algorithm since they do not generate huge number of candidates. Since, having huge number of similar transactions, $Stream - EFIM$ takes advantage of transaction merging thereby improving the performance over $Stream - FHM$. We do not consider two versions of EFIM since, EFIM with transaction merging performs exceptionally well for dense datasets.

Table 6.2: *Number of candidates generated by T − HUDS for ChainStore dataset*

Window size	Number of candidates
2	9,73,093
4	8,38,590
6	6,46,312
8	4,55,094

6.2 Varying window size (k=500)

In this subsection, we study the effect of varying the window size, i.e, varying the number of batches that can be accommodated in a window. We fix the value of k to 500 for our experiments. The results of Chainstore and Accidents datasets are shown in Figure 6.3. Experimental results show that the time taken by our variants decreases with increase in a number of batches in a window. This is because with the increase in window size, implies more number of batches can be accommodated in a window but the number of windows formed is less as we increase the window size. The number of intermediate itemsets generated per window will be more as we increase the window size but due to a decrease in the number of windows, the overall time taken will be less.The number of intermediate itemsets generated by $Stream - EFIM$ and $Stream - FHM$ can be shown in Table 6.3.

The decrease in a number of potential high utility itemsets in tree based T-HUDS algorithm for different window size can be shown in Table 6.2. The huge number of generated candidates dominates the performance of T-HUDS algorithm.

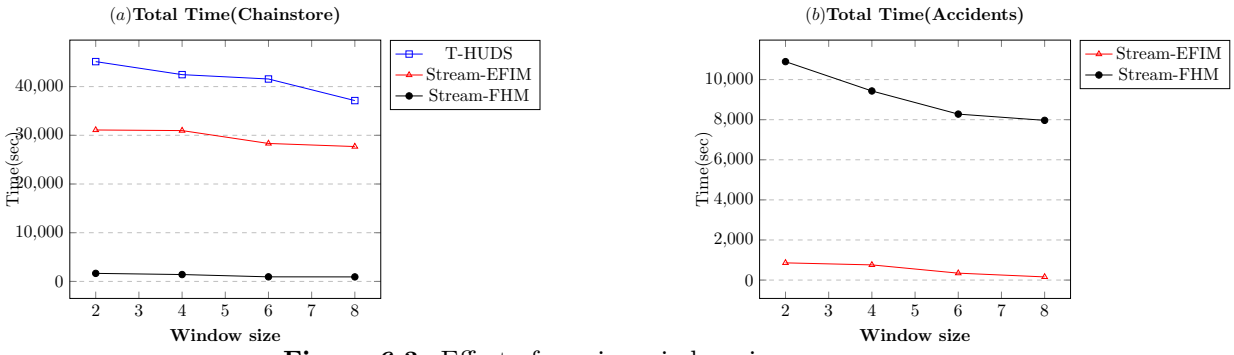

Figure 6.3: Effect of varying window size

Table 6.3: *Number of intermediate itemsets generated by our variants for varying window size*

Dataset	Window size	Stream-FHM	Stream-EFIM
ChainStore	2	10,75,15,642	1,12,496
	4	10,69,49,551	92,458
	6	10,58,67,820	72,921
	8	10,55,96,199	52,507
Accidents	2	74,15,303	6,65,014
	4	71,30,037	4,64,360
	6	66,45,200	2,38,414
	8	63,50,020	1,21,276

7

Conclusions and Future Work

In this thesis, we studied the state-of-the-art algorithms on dynamic transaction databases. We implemented the *top-k* streaming version of state-of-the-art static transaction database algorithms $Stream - EFIM$ and $Stream - FHM$ and compared them with the dynamic transaction database T-HUDS algorithm which is based on the data stream. Experiments are conducted on real datasets which shows the superior performance of defined streaming version algorithms over T-HUDS algorithm due to the generation of a huge number of candidates in T-HUDS. Both EFIM and FHM algorithms are applied naively on every window without reusing the data structure from the previous window. $Stream - EFIM$ exceptionally reduces the space complexity in both sparse and dense datasets. For Dense datasets, $Stream - EFIM$ outperforms the other two algorithms both in terms of memory and time due to transaction merging and database projection. For sparse dataset, $Stream - FHM$ performs better than the other two in terms of running time since $Stream - FHM$ do not apply transaction merging thus reducing the running time of the algorithm.

References

[1] Charu C Aggarwal. *Managing and mining sensor data*. Springer Science & Business Media, 2013. 2, 8

[2] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994. 1, 7

[3] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Ho-Jin Choi. Interactive mining of high utility patterns over data streams. *Expert Systems with Applications*, 39(15):11979–11991, 2012. 3, 8, 10

[4] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee. Efficient tree structures for high utility pattern mining in incremental databases. *Knowledge and Data Engineering, IEEE Transactions on*, 21(12):1708–1721, 2009. 8

[5] Raymond Chan Chan, Qiang Yang, and Yi-Dang Shen. Mining high utility itemsets. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 19–26. IEEE, 2003. 8

[6] Philippe Fournier-Viger, Cheng-Wei Wu, Souleymane Zida, and Vincent S Tseng. Fhm: faster high-utility itemset mining using estimated utility co-occurrence pruning. In *Foundations of intelligent systems*, pages 83–92. Springer, 2014. 9, 21, 22

[7] B Goethals and MJ Zaki. the fimi repository, 2012. 34

[8] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000. 1, 6, 7

[9] Mengchi Liu and Junfeng Qu. Mining high utility itemsets without candidate generation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 55–64. ACM, 2012. 9, 21, 31

[10] Ying Liu, Wei-keng Liao, and Alok Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In *Advances in Knowledge Discovery and Data Mining*, pages 689–695. Springer, 2005. 1, 8, 9

[11] Jian Pei, Jiawei Han, Runying Mao, et al. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, volume 4, pages 21–30, 2000. 7

[12] Jayaprakash Pisharath, Ying Liu, Wei-keng Liao, Alok Choudhary, Gokhan Memik, and Janaki Parhi. Nu-minebench 2.0. *Department of Electrical and Computer Engineering, Northwestern University, Tech. Rep*, 2005. 34

[13] Jerry Chun-Wei Lin Cheng-Wei Wu Vincent S. Tseng Souleymane Zida, Philippe Fournier-Viger. Efim: A highly efficient algorithm for high-utility itemset mining. In *Advances in Artificial Intelligence and Soft Computing*, pages 530–546. Springer, 2015. 9, 23, 26

[14] Vincent S Tseng, Cheng-Wei Wu, Philippe Fournier-Viger, and Philip S Yu. Efficient algorithms for mining top-k high utility itemsets. *Knowledge and Data Engineering, IEEE Transactions on*, 28(1):54–67, 2016.

[15] Vincent S Tseng, Cheng-Wei Wu, Bai-En Shie, and Philip S Yu. Up-growth: an efficient algorithm for high utility itemset mining. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 253–262. ACM, 2010. 8, 9

[16] Mohammed J Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 326–335. ACM, 2003. 6

[17] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, Wei Li, et al. New algorithms for fast discovery of association rules. In *KDD*, volume 97, pages 283–286, 1997. 6, 7

[18] Morteza Zihayat and Aijun An. Mining top-k high utility patterns over data streams. *Information Sciences*, 285:138–161, 2014. 4, 10, 15, 16, 19, 33