VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Efficient incremental high utility itemset mining with Hashtrie

By
Trần Xuân Hiếu

A thesis submitted to the School of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Bachelor of Data Science

Ho Chi Minh City, Vietnam
2022

Efficient incremental high utility itemset mining with Hashtrie

APPROVED BY:

_____,
Nguyen Van A, Ph. D, Chair
(*Type Committee names beneath lines*)

_____
(*Typed Committee name here*)

_____
(*Typed Committee name here*)

_____
(*Typed Committee name here*)

_____
(*Typed Committee name here*)

THESIS COMMITTEE
(Whichever applies)

ACKNOWLEGMENTS

First and foremost, I would like to announce my greatest amount of gratitude and appreciation towards the wisdom and support of Assoc. Prof. Nguyen Thi Thuy Loan. It is with the constant encouragement from Professor Loan that had enabled me to achieve the goals of my research for this thesis.

Additionally, I would like to henceforth establish my thanks to our weekly Seminar group that is organized by Professor Loan and Professor Bay. Their profound knowledge of the related fields has made it possible for me to quickly grasp the core ideals of the study. Once again, my appreciation to the Bay-Loan seminar team and every member participated.

I am grateful to the faculty of the School of Computer Science of the International University, especially Assoc. Prof. Nguyen Thi Thuy Loan for her support ever since the beginning of my studies. Gratitude is also expressed to the members of my reading and examination committee.

Table of Contents

ACKNOWLEGMENTS 3

LIST OF ACRONYMS 6

LIST OF FIGURES 7

LIST OF TABLES 8

ABSTRACT 9

CHAPTER 1 10

INTRODUCTION 10

1.1. Background 10

1.2. Overview about data mining 10

1.3. Pattern mining 10

1.4. Structure of thesis 12

CHAPTER 2 14

LITURATURE REVIEW/RELATED WORK 14

2.1. High Utility itemset mining 14

2.2. Definition in High utility itemset mining 15

2.3. High utility itemset mining algorithm 17

2.3.1. HUI miner [1] 17

2.3.2. FHM miner [2] 21

2.3.3. EIHI Algorithm [3] 23

CHAPTER 3 31

Trie Structure and improvement 31

3.1. Overview 31

3.2. Perquisition analysis 31

3.3. Compare between ListTrie and Hashtrie 32

CHAPTER 4 35

IMPLEMENT AND RESULTS 35
4.1. Experimental database 35
4.2. Experiment on Mushroom Dataset 35
4.3. Experiment on Retail Dataset 37
4.4. Experiment of Chainstore dataset 42
CHAPTER 5 48
DISCUSSION AND EVALUATION 48
5.1. Conclusion 48
5.2. Development 49
REFERENCES 50

LIST OF ACRONYMS

Name	**Meaning**
Item	Single item in the transaction *(eg a, b, c, d, ...)*
Itemset	A combination of single item in the transaction *(eg {abc}, {def}, {bedc}, ...}*
Profit	Profit or utility value of the item
HUI	High-Utility Itemset
Minutil	Minimum utility specified beforehand
iutil	Initial Utility value of the item or Itemset
rutil	Remaining Utility value of the item or Itemset
HUIM	High-Utility Itemset Mining
TWU	Transaction – Weighted Utilization
UL	Utility-list
TID	Transaction Identifier
TU	Transaction Utility
FIM	Frequent Itemset Mining
EUCS	Estimated Utility Co-Occurrence structure
FHM	Fast High-Utility Itemset Mining us Estimated Utility Co occurrence Pruning
Hui-Miner	High Utility Itemset Miner
EIHI	Efficient Incrmental High-utility Itemset miner
HUI-LIST-INS	HUI-LIST INS algorithm [4]
ListTrie	HUI trie structure used in EIHI algorithm
Hashtrie	HUI trie structure proposed

LIST OF FIGURES

Figure 2.1 Increment Utility Structure 24

Figure 2.2 UL after update 25

Figure 2.3 HUI trie 27

Figure 2.4 HUI trie after update update 30

Figure 4.1 Running time between two types of Trie in the Mushroom dataset 36

Figure 4.2 Memory usage between two types of Trie in the Mushroom dataset 36

Figure 4.3 Running time in the Retail dataset experiment 1 38

Figure 4 4 Memory usage in the Retail dataset experiment 1 37

Figure 4.5 Running time in the Retail dataset experiment 2 38

Figure 4.6 Memory usage result in the Retail dataset experiment 2 39

Figure 4 7 Running time result in the Retail dataset experiment 3 40

Figure 4 8 Memory usage in the Retail dataset experiment 3 40

Figure 4.9 Running time result in the Retail dataset experiment 4 41

Figure 4.10 Memory usage result in the Retail dataset experiment 4 41

Figure 4.11 Running time in the chainstore dataset experiment 1 43

Figure 4 12 Memory usage in the chainstore dataset experiment 1 43

Figure 4.13 Running time in the chainstore dataset experiment 2 44

Figure 4.14 Memory usage in the chainstore dataset experiment 2 44

Figure 4.15 Running time in the chainstore dataset experiment 3 45

Figure 4.16 Memory usage in the chainstore dataset experiment 3 46

Figure 4.17 Running time in the chainstore dataset experiment 4 46

Figure 4.18 Memory usage in the chainstore dataset experiment 4..... 47

LIST OF TABLES

Table 2.1 Example database transaction.....15

Table 2.2 External utility of individual 15

Table 2.3 Update database.....15

Table 2.4 utility of 5 transaction.....16

Table 2.5 TWU of each item increasing order 20

Table 2.6 Utility list of each item 20

Table 2.7 EUCS structure of given database.....21

Table 2.8 TWU of D.....23

Table 2.9 TWU of D' 23

Table 4.1 Experiment data.....35

Table 4.2 number of new HUI itemset Mushroom exp 1.....35

Table 4.3 number of new HUI itemset Retail exp 1.....37

Table 4.4 number of new HUI itemset Retail exp 2.....38

Table 4.5 number of new HUI itemset Retail exp 2.....39

Table 4 6 number of new HUI itemset Retail exp 4.....41

Table 4 7 number of new HUI itemset chainstore exp 4.....43

Table 4 8 number of new HUI itemset chainstore exp 2.....44

Table 4 9 number of new HUI itemset chainstore exp 3.....45

Table 4 10 number of new HUI itemset chainstore exp 4.....46

ABSTRACT

High-utility itemset mining (HUIM) has emerged as a crucial study area in recent decades, as both quantity and profit variables are important when mining high-utility itemsets (HUIs). Ever since then, there have been several similar algorithms proposed. In 2012, Liu and Qu presented the HUI-Miner method (High Utility Itemset Miner) to exploit high utility itemsets by storing all valuable information about an itemset in a new structure called a utility list. The HUI-Miner method is believed to be an effective technique for extracting high utility itemsets. In 2014, Phillipe and colleagues proposed the FHM method by employing a utility list structure to minimize search space.

However, the database is always evolving since fresh data is collected regularly, coal... To leverage HUI on a growing database, an efficient technique is required. The EIHI method presented in 2015 is a very efficient item-mining technique applicable to the growing database, the algorithm used a tree-typed data structure to perform storing and updating whenever a new itemset is founded, this process can cause certain drawbacks in the algorithm, especially when dealing with big data. In this paper, I will focus on the procedure of organizing itemset during the mining of the EIHI algorithm and suggest a better data structure to improve the performance of EIHI algorithm.

CHAPTER 1

INTRODUCTION

1.1. Background

It is common knowledge that itemset mining has enabled a multitude of realistic applications and a frequent task in application database. The research regime is composed of many tasks that support the finding of subsets belonging to a currently examined item that could eventually yield high impact in total transaction. High-utility itemset mining (high-utility itemset) [1] is an extended problem of frequent item set mining, which has been interested by many authors for the purpose of evaluating the significance of item sets in association rule mining.

However, real life database is usually changed overtime, this proved to become a hindrance for common HUI miner algorithm as well as its extension FHM [2]. Although there have been several algorithms made to adapt dynamic database like HUI-LIST-INS [4]. However, these algorithms are still expensive in terms of execution time. Updating the resulting itemsets when the database changes is also one of the urgent needs. In this paper, I will present a new data structure that can be used for incremental database extended for previous HUI miner and FHM to increase the performance of the current one.

Overview about data mining

a. Definition

From an overview perspective, Data mining can be considered as a family of techniques that are also found in other studies such as machine learning, statistics, database design and so on. Therefore, it could be generally accepted that data mining is in fact an interdisciplinary subject that intertwine the fields in computer science with the overarching objective to eventually extract the invaluable secrets behind every data pattern as well as to generate meaningful messages hidden deep within the vast sea of unintelligible information.

b. Application

Even though the field of data mining has not been established for long, there are definitely many businesses spanning across different areas from the manufacturing of products to the nurture of human's well-being under healthcare that are already utilizing the bountiful products of data mining to capitalize upon their existing storages of information. As many disciplinaries and actual use cases have suggested, data mining has assisted many researchers and scholars in determining crucial information patterns from invalid and unhelpful data that would have otherwise gone unnoticed.

In the field of businesses, data mining can help to reveal various interesting aspects that would eventually support many crucial business decisions that could potentially affect a whole group of connected human resources network. For the agencies working in marketing, data mining can help provide important insights into customers' spending behaviors and would subsequently allow more effective marketing initiatives to be developed, which would then accurately satisfy the customers need and allow the business to profit better from their new-found loyalty

c. Challenge

Although there has been many solutions and method support for data mining but in reality, we still face a lot of challenge include:

+ Big data

+ Dynamic data

+ Invalid attribute

+ Missing value

+ Uncertainty

+ Security

And so on …

1.2. Pattern mining

Pattern mining involves finding rules that characterize certain patterns in data. Although finding these relationships has long been possible and is often easier on smaller datasets data mining but reality shows trivial and faces a certain difficulty when extracting frequent patterns on a large-scale and dynamic database. It is important to identify unexpected hindrances that may open up a new method for research and finding. A major use of pattern mining is to detect sequential patterns and analyze sequence or structure data such as trees, graphs, and networks. Therefore, it would offer a wide range of applications to its user such as clustering, classification, software bug identification, recommendations, and several other challenges.

Anomaly detection:

Anomaly detection [5] is the opposite of clustering, as it helps the users to pinpoint with high certainty of accuracy all the items that are either too exotic or could not meet the requirements to be listed along in any recognized patterns beforehand. An exemplary and solid descendant of anomaly detection that is fraud detection. Although fraud detection may be considered a challenge for predictive modeling, the relative rarity of fraudulent transactions, as well as the speed with which criminals invent new kinds of fraud, means that any prediction model is likely to be inaccurate and out of date.

Real-life examples included detecting a series of failures or warnings before equipment failure or providing insight into design failures.

Market basket analysis:

The field of study referred to as Market basket analysis [6] involves a procedure of steps that are focused on connecting the correlations between things in a commercial environment. However, market basket analysis can also extend beyond the supermarket scenario that inspired its name. Therefore, it can be generalized that Market basket analysis is the examination of any group of commodities to uncover affinities that may be exploited in some way. Some applications of market basket analysis include:

- Product placement
- Physical shelf arrangement
- Up-sell, cross-sell, and bundling opportunities.
- Customer retention.

In this research, we aim to focus on High utility Itemset mining and High utility itemset mining in the incremental database. HUI is a widely studied data mining algorithm used to discover high utility itemset, yet it remains some weakness in data structure to update the new HUI itemset as it can be very large in terms of execution time and memory

1.3. Structure of thesis

The thesis is divided into five chapters, as follows:

- Chapter 1: An overview of the practical setting influencing the thesis's study approach.
- Chapter 2: Introducing the theoretical foundation for Chapter 3. The primary goal of this chapter is to provide two strategies for mining high utility itemset, in detail: HUI-Miner, FHM algorithm, and mining algorithm.
- Chapter 3: The thesis presents the proposed data structure to improve the performance of the mining process and compared with the original algorithm (EIHI).
- Chapter 4: The thesis conducts experiments on the databases to demonstrate the proposed data structure.
- Chapter 5: Presenting the conclusion and development direction of the thesis.

CHAPTER 2

LITURATURE REVIEW/RELATED WORK

2.1. High Utility itemset mining

High Utility itemset mining (HUIM) is a generalized problem of frequent itemset mining [7]. In this problem, utility of an item is defined as the measurement of how much profit of that item or percentage profit of item in database. As a result, its goal is to find all combinations of a set of items with profit higher than a specified threshold (called minimum utility) and to reduce the dimension of candidates itemset for analyzing and another purpose.

A main approach is to list all itemset from database based on heuristic search which can be represent in form of tree structure. However, as a heuristic search function, the algorithm has large time complexity when facing with a large data or when minimum utility set by user is small. Therefore, it is necessary to have a smart strategy to effectively mining and pruning itemset.

In HUIM, not only frequency but the weight of item and itemset need to be considered. Therefore, even though an itemset appears multiple, if the total weight does not surpass the minimum utility, it is not considered a high utility itemset.

There are many algorithms involved that have been proposed. Typically, Liu and colleagues (2005) proposed a Two-Phase algorithm with concepts of transaction utility (TU) and transaction weighted utility (TWU) to improve the highly useful search space.[8]

In 2012, Liu and Qu came up with the HUI-Miner (High Utility Itemset Miner)[1] algorithm that can derive high utility sets using a new data structure called Utility List in order to encapsulate information about an itemset and to efficiently reduce the search space. The HUI-Miner algorithm outperformed the previous work for extracting high utility itemsets until the

emergence of the FHM algorithm [2], an extension of HUI miner proposed by Phillipe and colleagues in 2014.

2.2. Definition in High utility itemset mining

This part presents some definition, concept about problem in mining high utility itemset [9][10]

Given *X*, *I* is a finite set of items to $X \subseteq I$. A transaction database D is a set of many transaction $D = \{T1, T2, T3..., Tn\}$ each transaction has a unique identifier (Tid) and is a subset of I. Each item in the transaction has a following number indicated quantity of that item in the transaction, thus multiply with external value of individual item to create internal item value. Value, weight, or utility is general term to express the importance of item in total transaction,

Table 2.1 Example database transaction

Tid	Transaction	TU
T_1	(a,1), (c,1), (d,1)	8
T_2	(a,2), (c,6), (e,2), (g,5)	27
T_3	(a,1), (b,2), (c,1), (d,6), (e,1), (f,5)	30
T_4	(b,4), (c,3), (d,3), (e,1)	20
T_5	(b,2), (c,2), (e,1), (g,2)	11

Table 2.2 External utility of individual

Item	a	b	c	d	e	f	g
Utility	5	2	1	2	3	1	1

Table 2.3 Update database

T_6	(b,2), (d,5), (f,2)	16
T7	(a,1), (c,2), (d,1), (e,1)	12

Table 2.1 describes a standard database transaction to this problem. The left most column (Tid) is transaction identifier. The middle is detail of a transaction for example, (a,1), (c,1), (d,1) is one item **a,** one item **c** and one item **d.** Finally, we have TU columns stand for Transaction Utility which is the sum of item utility in the transaction. Database transaction D contains five transactions (*T1, T2, T3, T4, T5)* with a set of item I = *{a, b, c, d, e, f, g}* and table 2.2 presents each item external utility respectively.

Definition 1. *The external utility of item i*, denoted as *eu(i),* is the utility value of i in the utility table.

Definition 2. *The internal utility of item i in transaction T*, denoted as *iu (i, T),* is the count value associated with i in T in the transactional table.

Definition 3: *Item's utility:* Utility of an item i_j in transaction T_d is defined as $u(i_j, T_d)$ and is calculated as:

$$u(i_j, T_d) = iu(i_j, T_d) \times eu(i_j)$$

$u(i_j, T_d)$ show the profit of selling item i in transaction T_d

For example: Item a in table T_2 has the utility of u(a, T_2) = 2 x 5 =10, whereas item c in T_1 has the utility of u(c, T_1) = 1 x 1 = 1.

Definition 4: *Utility of itemset X in transaction T*, denoted as u(X, T), is the total utility of all the item in X in transaction table T

$$u(X, T) = \sum_{i \in X \wedge X \subseteq T} u(i, T)$$

Definition 5: *Utility of itemset X*, denoted as u(X), is the sum of all utilities of that item in all transactions containingit, where $u(X) = \sum_{T \in DB \wedge X \subseteq T} u(X, T)$.

For example, in table 2.1, u({ae}, T2) = u(a, T2) + u(e, T2) = 2 × 5 + 2 × 3 = 16 , and u({ae}) = u({ae}, T2) + u({ae}, T3) = 16 + 8 = 24.

Definition 6: *The utility of a transaction T*, denoted as TU (T), is the sum of the utilities of all items in T, where TU(T) = $\sum_{i \in T} u(i, T)$, and the total utility of database is the sum of the utilities of all the transactions in transactional database.

Table 2.4 utility of 5 transaction

Tid	1	2	3	4	5
TU	8	27	30	20	11

Definition 7: *High utility itemset.* An itemset X is a high utility itemset (HUI) if its utility surpasses a specified minimum utility (minutil).

Definition 8: *Transaction Weighted Utility* denoted TWU(X). The weight of the transaction utility of an item X in database D is the sum of transactions containing X

$$TWU\ (X) = \sum_{T_{d \epsilon D} \wedge x \subseteq T_d} TU(T_d)$$

Definition 9: *Utility List* denoted as UL (X). Utility List is data structure consist of Tid, iutil and rutil of each itemset, where rutil is remaining util defined by:

$$\sum_{i \in T_{tid} \wedge i > x \forall x \in X} u(i, T_{tid})$$

2.3. High utility itemset mining algorithm

2.3.1. HUI miner [1]

To identify high utility itemset, most algorithm will generate candidate itemset from how high utility they are then calculate exact utility. These algorithms face the problem of generating a large number of candidates set, but most of the candidates are found to be not high utility after the utilities are correctly calculated. HUI miner uses a structure called Utility List (UL) to contain utility information of itemset as well as effectively prune the search space. By avoiding generate unnecessary itemset, HUI-Miner is more efficient than previous predecessor because it can extract high utility set from utility list without consuming too much resources.

Construct Algorithm

Input: $P.UL$: Utility list of itemset P

$Px.UL$: Utility list of itemset Px

$Py.UL$: Utility list of itemset Py

Output: $Pxy.UL$: Utility list of itemset Pxy

1. $Pxy.UL = NULL$

2. Foreach element $E_x \in P_{x.}.UL$ **do**

3. **If** $\exists E_y \in P_{y.}UL$ and $E_{x.tid} == E_{y.tid}$ **then**

4. **If** $P.UL \neq null$ **then**

5. search $E \in P.UL$ that $E_{tid} == E_{tid}$

6. $E_{xy} =< E_{x.tid,}\ E_{x.iutil} + E_{y.iutil} - E_{.iutil}\ , E_{y.rutil} >$;

7. **Else**

8. $E_{xy} =< E_{x.tid,} E_{x.iutil} + E_{y.iutil,} E_{y.rutil>}$;

9. **End**

10. Append E_{xy} to Pxy

11. **End**

12. End

13. Return $Pxy.UL$

HUI-Miner Algorithm

Input:

P.UL: Utility List of itemset P, initially empty

ULs: the set of utility-lists of all P's 1-extensions.

minutil: the minimum utility threshold.

Output: all the high utility itemsets with P as prefix

```
1  foreach UL( X) ∈ Uls do
2        if SUM(X. iutil) ≥ minutil then
3              output the extension associated with X.
4        end
5        if  SUM(X. iutils) + SUM(X. rutils) ≥ minutil then
6              ex. UL_s = NULL
7              foreach utility − list Y after  X in  Uls do
8                    exULs = exULs + Construct(P. UL, X, Y);
9              end
10             HUI − Miner (X, exULs , minutil);
11       end
12 ends
```

For each Utility List, if sum of all iutil of an itemset is equal or greater than minutil then that itemset along with all its subset extension is also high util. If sum of iutil and rutil greater or equal minutil then it needs to be explored. All Utility list is constructed and arrange in transaction weighted utility (TWU) increasing order to ensure all the itemset extension is explored logically. For example, let X be the Utility List of item Px, Y is the Utility List of Py, we call Construct algorithm to return all extension of Pxy for HUI miner to extract all high utility itemset.

Demonstration:

Using database tin table 2.1 and table 2.2 with minutil =30

Step 1: Scan through Database

Step 2: Compute TWU

Step 3: Exclude any itemset has TWU < minutil and sort the remain increasing order

Table 2.5 TWU of each item increasing order

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

$I^* = (\{f, g, d, b, a, e, c\}$

Step 4: sort the order of item set in database in TWU increasing order then construct Utility List for each item

Table 2.6 Utility list of each item

{f}		
T3	5	25

{g}		
T2	5	22
T5	2	9

{d}		
T1	2	6
T3	12	13
T4	6	14

{b}		
T3	4	9
T4	8	6
T5	4	5

{a}		
T1	5	1
T2	10	12
T3	5	4

{e}		
T2	6	6
T3	3	1
T4	3	3
T5	3	2

{c}		
T1	1	0
T2	6	0
T3	1	0
T4	3	0
T5	2	0

We then combined item {f} with the remain to create its further extension

{fg}		
T3	0	0

{fd}		
T3	17	13

{fb}		
T3	9	9

{fa}		
T3	10	4

{fe}		
T3	8	1

{fc}		
T3	6	0

{fdb}		
T3	21	9

{fda}		
T3	22	4

{fde}		
T3	20	1

{fdc}		
T3	18	0

{fdba}		
T3	26	4

{fdbe}		
T3	24	1

{fdbc}		
T3	22	0

{fdbae}		
T3	29	1

{fdbac}		
T3	27	0

{fdbaec}		
T3	30	0

2.3.2. FHM miner [2]

FHM stand for Faster High-Utility Itemset Mining is an extension of HUI-miner with a new data structure called EUCS (Estimated Utility Co-Occurrence Structure) to reduce number of candidates. FHM help reduce 95% search space and run 6 times faster than normal HUI miner.

EUCS structure is set with three mains elements *(a, b, TWU (a, b).* We present in form of HashMap with 2 keys is *a, b* and its corresponding value *TWU (a, b)* to reduce memory if the number of items is large. We can think EUCS is a 2-dimension transaction weighted utility table, given table 2.1 and 2.2 we construct EUCS below:

Table 2.7 EUCS structure of given database

	f	g	d	b	a	e	c
f							
g	0						
d	30	0					
b	30	11	50				
a	30	27	38	30			
e	30	38	50	61	57		
c	30	38	58	61	65	88	

With EUCS, we can prune combination of item that lower than minimum utility. Since if TWU (a, b) < *minutil* then all its open subset is also low utility. In table 2.7 we can see {fg}, {gd}, {gb}… is a low itemset therefore we can ignore at these subsets. HUI miner is already an optimizing algorithm, however joining between Utility List is a costly operation in both memory and time, EUCS helps predict and early remove itemset that does not yield enough utility without joining, create an effective way to improve HUI miner

FHM Algorithm

Input: D: A transaction database

$minutil$: user-specified threshold

Output: set of high-utility item

1. Scan D to compute TWU for single item.
2. $I* <-$ each item i such that $TWU(i) \geq minutil$;
3. Let $\succ$ be the total order of TWU ascending values on $I*$;
4. Scan D to build for single item $i \in I*$ and compute value for EUCS
5. Search (Ø, I *, minutil, EUCS);

Search Procedure

Input: P: an itemset

ExtensionsOfP: a set of extensions of P,

Minutil: threshold

EUCS structure

Output: Set of high utility itemset

```
1  Foreach itemset Px ∈ Extensions of P do:
2      If SUM(Px.utilitylistiutils) ≥ minutil then
3          output Px;
4      end
5      If SUM(Px.utilitylist.iutils) + SUM(Px.utilitylist.rutils)≥ minutil then
6          ExtensionsOfPx = null
7          Foreach Py ∈ ExtensionsOfP such that y ≻ x do
8          If ∃(x, y, c) ∈ EUCS such that c ≥ minutil then
9              Pxy ← Px ∪ Py
10             Pxy.utilitylist = Construct (P, Px, Py)
11             ExtensionsOfPx = ExtensionsOfPx ∪ Pxy.
12         end
```

First part of the algorithm is similar with normal HUI- miner, if the sum of iutil of X greater or equal minutil then X is high utility, if sum of total iutil and rutil greater or equal then

X need to open. For each y $\succ x$ we check in EUCS if combination of X with each Y has TWU greater or equal minutil to make decision to explore or not. Utility of Px with respective Py is create in similar with HUI miner, then search procedure begin with single item and recursively explore combination with another item. With EUCS support, the search will ignore all the combination of two item does not generate high utility reducing search space and increase algorithm performance.

2.3.3. EIHI Algorithm [3]

The problem with HUI miner is it assume the database is static, however most database is dynamic which mean they are continuously update cause data mining with HUI became a costly operation. In 2015, Philippe and colleagues came up HUI miner adapt to increment update database which is EIHI

High utility itemset mining in increment database is developed from normal High utility itemset mining therefore, so it inherits all the definitions from the problem as well as include new definitions and properties

Given database transaction D, an updated database is DN if $D' = D \cup N$ where N is a non-empty transaction update transaction.

Given database D, updated database D', *minutil* defined by the user and H is a high utility set in D. The problem is to find high utility set H' in new database base on given information H in D

Table 2.8 TWU of D

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

Table 2.9 TWU of D'

Item	f	g	a	b	d	e	c
TWU	46	38	77	77	86	100	108

For instance, consider database in table 2.1 with *minutil =30*. D' is an updated database of D by adding T6= (b, 2) (d, 5) (f, 2) (table 2.4)... In the new database, we have H' is the HUI of D' H' = **{b, d: 50}**, {a, c, e: 31}, {b, c, d: 34}, {b, c, e: 31}, {b, d, e: 36}, {b, c, d, e: 40}, {a, b, c, d, e, f: 30} and **{d: 30}**. When T6 is added utility of {bd} increase from 30 to 50 and new utility list {d:30} appeared

New definition:

Definition 10: *Increment Utility List.* An extend version of Utility List with specific section for update transaction

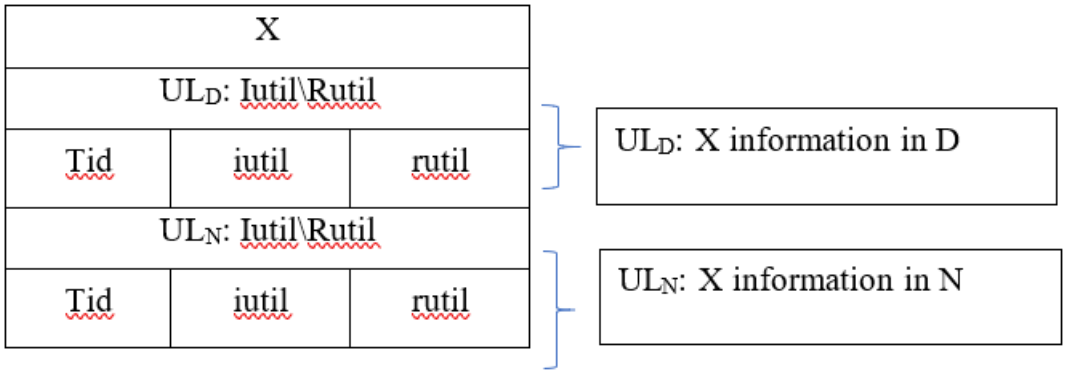

Figure 2.1 Increment Utility Structure

There will be distinct Utility List for update database (UL_n)

$$iUL(X).ul_D = U_{X \in TD\ tid}\ (tid; u(X, TD_{tid});\ \sum_{i \in X \wedge x > i} U(i, TD_{tid}))$$

$$iUL(X).ul_N = U_{X \in TN\ tid'}\ (tid'; u(X, TN_{tid'});\ \sum_{i \in X \wedge x > i} U(i, TN_{tid'}))$$

New Properties:

Property 2.1: Given an itemset $X \in H$ appear in D but not in N and $D' = D \cup N$. Utility of X remain the same

Property 2.2: *Pruning using TWU.* Given an itemset X if *TWU (X)* < *minutil* then X and all its subset is also a low utility itemset.

Property 2.3: *Pruning using the Sum of Iutil and rutil.* Given an item X and remain Item Y such that $y \succ i,\ \forall i \in X$. If Sum of iutil and rutil < *minutil* then all the subset is also low util

Property 2.4: *Sum of Iutil Value.* Given a set of items X, the utility of the set X on the database D' is the sum of the utility of X in D and the utility in N. U(X) is calculated by the formula

$u(X) = \sum iUL(X)ul_D.iutil + \sum iUL\ (X)ul_N.iutil$

Property 2.5: *If an itemset X is empty in N then X does not need to explore.*

Example: Table 2.1 with update database in Table 2.3 we build Uility List as below:

g

Tid	itil	rutil
2	5	22
5	2	9

f

Tid	itil	rutil
3	5	25
6	2	14

a

Tid	itil	rutil
1	5	3
2	10	12
3	5	20
7	5	7

b

Tid	itil	rutil
3	10	10
4	8	12
5	4	5
6	4	10

d

Tid	itil	rutil
1	2	1
3	6	4
4	6	6
6	10	0
7	2	5

e

Tid	itil	rutil
2	6	6
3	3	1
4	3	3
5	3	2
7	3	2

c

Tid	itil	rutil
1	1	0
2	6	0
3	1	0
4	3	0
5	2	0
7	2	0

Figure 2.2 UL after update

Property 2.6: *Pruning Using the Increment Utility-list.* Sort items in ascending order and expand X by appending an item y to X such that $y > i,\ \forall i \in X$. If the sum of the values of iutil and the values of rutil in $iUL(X).ul_D$ and the sum of iutil and rutil values in $iUL(X).ul_N$ is less than minutil, then X and all its bridged extensions are of low utility. To effectively prune the itemset, we first join UL in N then join UL in D since from **property 2.5** if UL of itemset in N is null then we do not need to join.

Property 2.7: Iutil is unchanged however rutil should be recalculate after update

EIHI algorithm is proposed by Fournier-Viger and colleagues based on the effectiveness of HUI miner to insert transactions, based on HUI-Trie structure. It performs twice the database scan in the determined order. The first part of EIHI is same with HUI miner

Example: Run HUI miner in table 2.1 and 2.2 with minutil =30

Step 1: scan database

Step 2: compute TWU

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

Step 3: Exclude any itemset has TWU < minutil and sort the remain increasing order, we have

$$I^* = \{f, g, d, b, a, e, c\}$$

Step 4: build EUCS data structure

	f	g	d	b	a	e	c
f							
g	0						
d	30	0					
b	30	11	50				
a	30	27	38	30			
e	30	38	50	61	57		
	30	38	58	61	65	88	

Step 5: Construct Utility List

{f}		
T3	5	25

{g}		
T2	5	22
T5	2	9

{d}		
T1	2	6
T3	12	13
T4	6	14

{b}		
T3	4	9
T4	8	6
T5	4	5

{a}		
T1	5	1
T2	10	12
T3	5	4

{e}		
T2	6	6
T3	3	1
T4	3	3
T5	3	2

{c}		
T1	1	0
T2	6	0
T3	1	0
T4	3	0
T5	2	0

Step 6: Evaluation is the same as with HUI miner which will not be mention again. The result in the first scan is High utility itemset Q = ({f, d, b, a, e, c}, {b, c, d, e}, {b, d, e}, {b, c, e}, {b,

c, d}, {a, c, e}, {b, d}) with corresponding value : {30, 40, 36, 31, 34, 31, 30}. We will need a structure to record the result so as to it can be update when new transaction is inserted.

After that, we recursively compute and explore its extension of item {f}. Here is how we evaluate an item

- Let consider Utility List of the first item {fb} has iutil 9 and remaining utility rutil 9. Since total of iutil and rutil < minutil therefore it is not a high utility itemset.

- Another situation, this time let's look at itemset {fdb}. Its iutil is 21, however sum of iutil and rutil is equal minutil, therefore extension of this itemset is explorable.

- At the end of {fdb} itemset extension we have {fdbaec} itemset that has iutil = 30 which is a high utility itemset.

We put itemset {fdbaec} to our result

Repeat above step for the remaining item we have the result:

Q = ({f, d, b, a, e, c}, {b, c, d, e}, {b, d, e}, {b, c, e}, {b, c, d}, {a, c, e}, {b, d}) with respective value: {30, 40, 36, 31, 34, 31, 30}

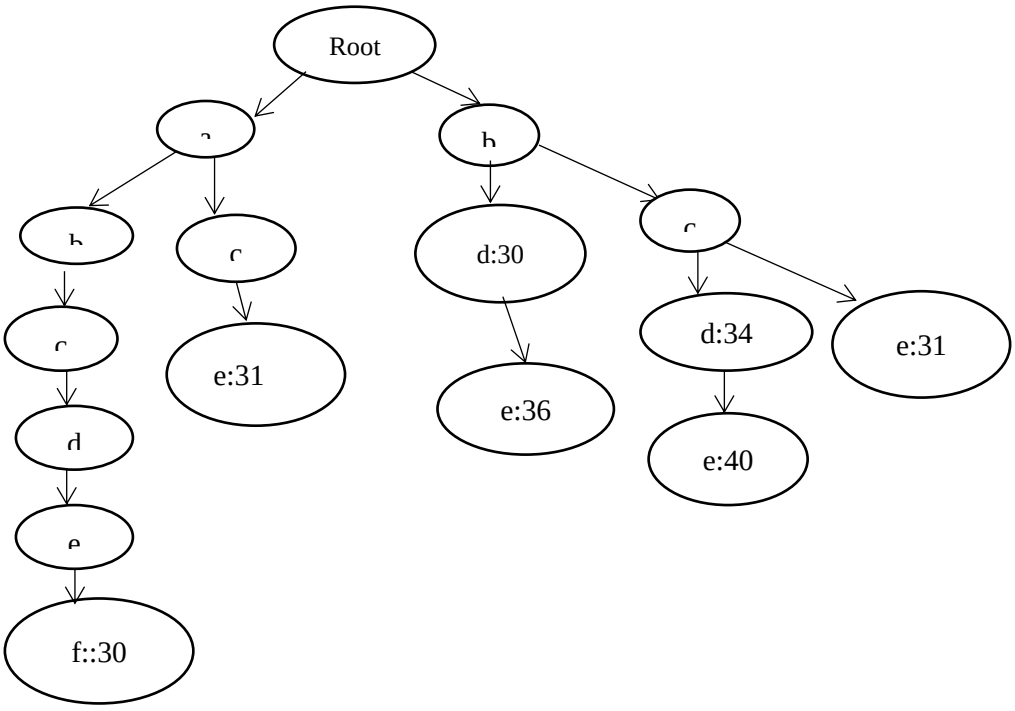

Figure 2.3 HUI-trie

Hui-trie is another type of data structure used to store high utility itemset that has been mine in the first run. The tree begins at the root, each node represents an item and each itemset represent by a path. Since after an update HUI-trie can only expand or stay the same as the result of the insertion of a new transaction, this allows the update of a new value or a new item set effectively as it is required to transverse or create |P| + 1 nodes starting from the root. This can be archived by performing a binary search at each node when looking for the child node corresponding to a given item.

Improvement and demonstration of EIHI compare with FHM

1. When update, it is only necessary to scan the new transactions in N to update TWU

Item	f	g	a	b	d	e	c
TWU	46	38	77	77	86	100	108

2. Re-evaluate items in I* when compare to minutil since several item have new TWU value that pass minimum utility.

$I^* = \{f, a, b, d, e, c\}$ we do not include g since it does not appear in update N

T6	(f,2), (d,5), (b,2)
T7	(d,1), (a,1), (e,1), (c,2)

3. Reorder item in I* in ascending order New order: $\{f, a, b, d, e, c\}$

4. Re-calculate Utility List for each item appear in N

{f}		
T3	5	25
T6	2	14

{d}		
T1	2	6
T3	12	13
T4	6	14
T6	10	4
T7	2	10

{b}		
T3	*4*	*9*
T4	*8*	*6*
T5	*4*	*5*
T6	*4*	*0*

{a}		
T1	*5*	*1*
T2	*10*	*12*
T3	*5*	*4*
T7	*5*	*5*

{e}		
T2	6	6
T3	3	1
T4	3	3
T5	3	2
T7	3	2

{c}		
T1	1	0
T2	6	0
T3	1	0
T4	3	0
T5	2	0
T7	2	0

An important idea here that we only consider and working on the update part, which mean we ignore order of previous mining as well as rutil. We only consider order of new item in step 3 to calculate rutil of update part.

5. Update new TWU in EUCS structure

	f	g	d	b	a	e	c
f							
g	0						
d	46	0					
b	46	11	66				
a	30	27	50	30			
e	30	38	62	61	69		
c	30	38	70	61	77	100	

6. Determine HUI by combine iutil in D and iutil in N

For example:

UL {f} = {5+2, 25+14} = {7,29}, iutil =7 < 30 however, 7+29 > 30 => expand item f

Evaluate {f, d}, {f, b}, {f, a}, {f, e}, {f, c} respectively

Consider itemset {fd}, TWU {fd} =46 > 30 therefore we can expand:

+UL {fd} = {17+12, 13+4} = {29,17}, iutil =29 < 30 however, 29+17 > 30 => expand {fd}

+UL {fdb} = {21+13, 9} = {34,9}, iutil =34 > 30 => HUI

Since {fdb} is not in HUI trie, we add from root

Consider itemset {fb}, TWU {fb} = 46 > 30 therefore we can expand:

+UL {fd} = {9+6, 9+0} = {15,9}, iutil =15 < 30 and 15+9 > 30 => no need to expand{fb}

Consider itemset {fa}, TWU {fa} = 30 however since fa is null in update N (f has T6 and a has T7 only) no need to expand {fa}. Similar case with {fe}, {fc}

Repeat with another item we built a new HUI-trie

7. If sum of iutil and rutil of D and N greater or equal *minutil* then we should expand that itemset

8. if an itemset does not have new utility in N (UL_N = null) then that itemset and all of its extension do not need to be explored

9. Using HUI-trie to store new HUI itemset. Perform binary search to check whether an itemset is in HUI-trie and update with new utility value, if not then we add new itemset from root

10. Combine D and N to a new D and prepare of the next update

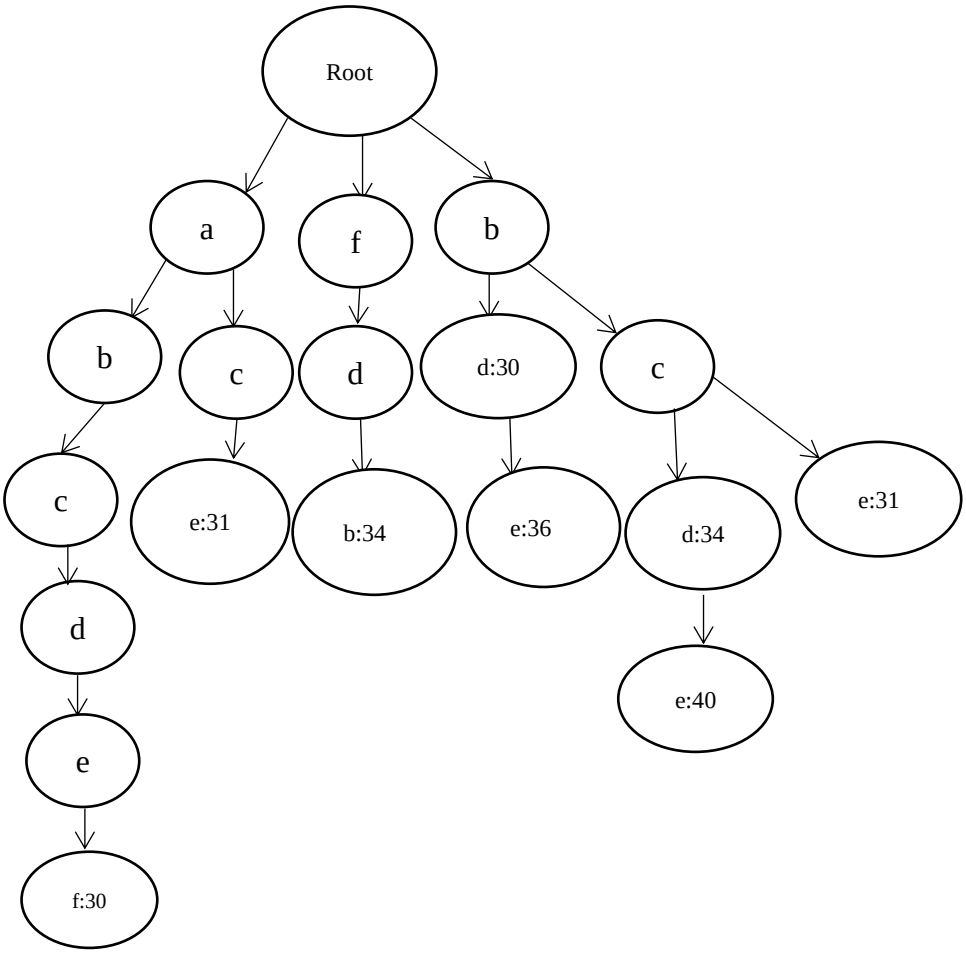

Figure 2.4 HUI trie after update update

CHAPTER 3

Trie Structure and improvement

3.1. Overview

A prefix tree, sometimes known as a trie, is a type of k-array search tree, which is a tree data structure for finding specific keys inside a collection. These keys are frequently strings, with character-by-character links between nodes rather than the whole key. The empty string is associated with the root, and all a node's children have a common prefix with the string associated with that parent node.

Example from the trie in Figure 2.3, we have the following HUI itemset: {abcdef}(30), {ace}(31), {fdb}(34), {bde}(36), {bcde}(40), {bce}(31), {bcd}(34)

Each itemset having the same prefix will share the same node path, by this way HUI trie can store and update itemset effectively.

3.2. Perquisition analysis

To create the trie data structure to store the HUI itemset and its utility, all nodes must have an array-like data structure to store sub-node or children nodes and a method to search a child node within the node in order to transverse and search across the trie.

One way to implement this is to have an ArrayList to store a list of child nodes within the node and have a binary search to look up the node we want recursively, this also required an array to be sorted by TWU order beforehand. However, this plan still consumes many time and resources. The second way is to create a HashMap pointer to quickly allocated to the child node we need; however, more resources will be expected since the hash pointer consumes more memory than ArrayList.

In this research, I will implement the second plan and compare the efficiency between two types of trie structures in both running time and memory usage. Moreover, it is

necessary to have an additional attribute to store and update the utility for an item set. The utility of an itemset will be placed at the last item while the remained item will have fixed utility (-1).

3.3. Compare between ListTrie and Hashtrie

3.3.1 ListTrie

ListTrie is a trie data structure that makes use of the Java ArrayList as its fundamental implementation. Every node in ListTrie contains a list of sub nodes with a default size of 10. Because ArrayList is an index-based data structure that array supports. The operation insertion in ListTrie includes sorting each item in TWU order using quick sort and then allocating the next node using binary search. For both quick sort and binary search, the computational complexity is O (n log n) * O (log n). Assuming an itemset has length M, the overall procedure is M * O (n log n) * (log n)

ArrayList merely saves the items as values and retains the indexing for each element internally, hence it is predicted to require less memory. The insertion technique using ListTrie, on the other hand, involves many complex additional operations such as searching and sorting therefore it is expected to be slower than Hashtrie.

3.3.2 Hashtrie

Hashtrie on the other hand uses HashMap as the core implementation to store data in terms of key-value pairs to achieve better running time.

HashMap operates on the hashing concept, which is a technique for mapping object data to a representative integer value. In order to store and retrieve any key-value pair, the hashing function is used to the key object to compute the index of the bucket. As a result, Hashtrie stores and retrieves items in constant time O (1)

The issue comes when the number of items is raised as a result of updating the new HUI itemset, but the bucket size remains constant. It will contain more items in each bucket that will cause

temporal complexity to be disrupted. HashMap in Hashtrie needs to increase the number of buckets as the number of objects grows for objects to be redistributed among all buckets.

For inserting and searching operation, thanks to the pair key-value, Hashtrie searching complexity is O (1) and the total complexity for the whole operation is O(M) * O (1)

In term of memory usage, Hashtrie is expected to consume more memory than ListTrie because ArrayList use more resources than HashMap, however List still need more operations than yield more resource to have new itemsets updated in HUI trie. The resources consumed by both data structures remained competitive

Another thing to consider when using hashing operation is a collision as there is a chance when the number of items is high enough, some keys will have the same hash value. This solution initially uses LinkedList to overcome this problem but Java 8 and subsequent versions have now stored colliding elements in a balanced tree (also known as a red-black tree) rather than a LinkedList. This reduces HashMap's worst-case performance from O(n) to O (log n).

However, the insertion to trie does not affect the mining procedure and only attempts to speed up the searching and adding process in trie to make the algorithm run faster and more consistent.

3.4. Constructor and insert method

Node Class:

```
Class Node {
        int item;
        int utility = -1;
        HashMap <item, Node> edges
    }
```

Insert Procedure

Input: P: an item set

Utility: utility of an itemset

Output: itemset is inserted in tree (void)

4. Node currNode =null
5. Node pointer = root
6. **For (i**=0, i<= P. size() - 1, i++) do**:** *// transverse across itemset except the last item*
7. currNode = pointer.edges.get(*P*[i]) *//find the current item*
8. **If** $currNode = null$ **then:**
9. currNode = Node (*P*[i])
10. Put currNode in pointer HashMap
11. pointer = currNode
12. **Else:**
13. pointer = currNode
14. lastitem = Last item of the itemset P
15. currNode = pointer.edges.get(lastitem) *//find the last item*
16. **If** $currNode = null$ **then**
17. currNode= Node (lastitem, utility)
18. Put currNode in pointer HashMap
19. **Else:**
20. currNode.utility = utility
21. **end**

CHAPTER 4

IMPLEMENT AND RESULTS

4.1. Experimental database

The test was performed on a computer with a Core i5 processor running on Windows 10 operating system and 8 GB of RAM. The algorithm is implemented in Java language, the algorithm compares the performance of the improved improved data structure Hashtrie with ListTrie - the existed one in the EIHI algorithm. The experiment was run on three real datasets, Mushroom, Chainstore, Chess, which were obtained from the website https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php. The characteristics of these databases are described in Table 4.1.

Table 4.1 Experiment data

Dataset	Transaction	Item count	Average length	Density
Retail	88,162	16470	10.30	0.06 %
Mushroom	8416	119	23	19.33%
Chainstore	1,112,949	46086	7.23	0.02 %

4.2. Experiment on Mushroom Dataset

The test was performed on mushroom dataset which has 8,416 transaction to compare the performance of Hashtrie and ListTrie.

Experiment 1: Assume the initial database has transactions from 1 to 6500 and the remaining 1500 transactions will be divided in three times adding 500 transactions each time to evaluate the execution time and memory of the two algorithms.

Min_utility= 100,000

Table 4.2 number of new HUI itemset Mushroom exp 1

number transaction	1 - 6500	6501-7000	7001-7500	7501-8000
Number of HUI	898015	19999	773	373

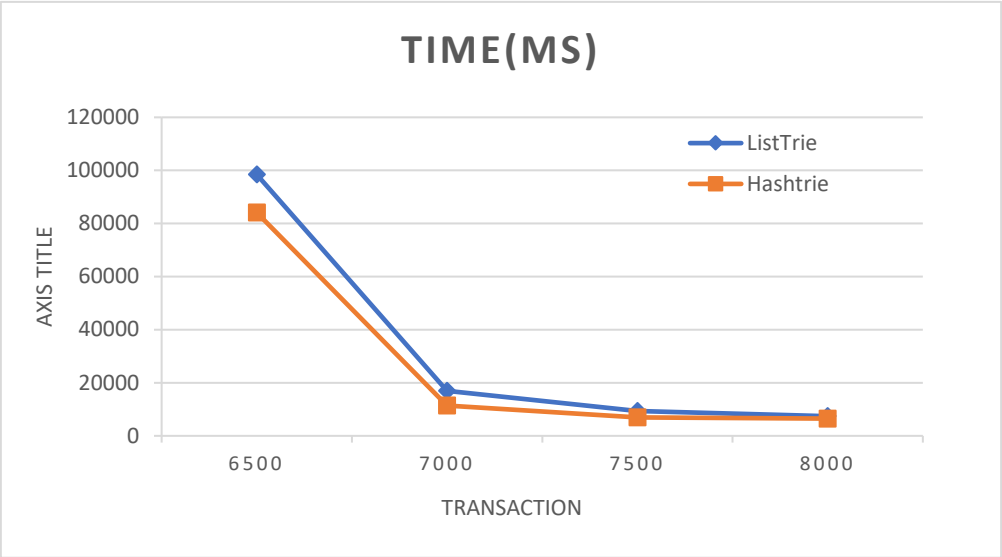

Figure 4.1: Running time between two types of Trie in the Mushroom dataset

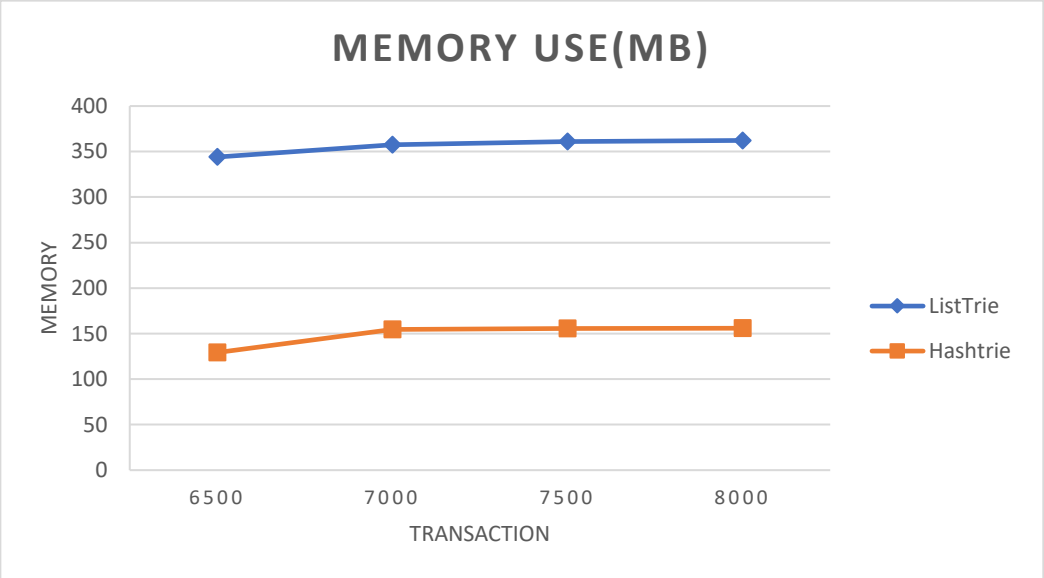

Figure 4.2: Memory usage between two types of Trie in the Mushroom dataset

Discussion: Based on the results of our tests on the Mushroom database, we discovered that Hashtrie is quicker in the first run and about the same as ListTrie in the second run. Overall, Hashtrie uses less memory than ListTrie in terms of memory use.

4.3. Experiment on Retail Dataset

The test was performed on Retail dataset which has 88,162 transaction. However, only 86,000 transaction will be used to compare the performance of Hashtrie and ListTrie.

Experiment 1: Assume the initial database has transactions from 1 to 80,000 and the remaining transactions will be divided in three times, adding 2000 transactions each time to evaluate the execution time and memory of the two algorithms.

Min_utility= 1350

Table 4.3 number of new HUI itemset Retail exp 1

number transaction	1 - 80000	80001-82000	82001-84000	84001-86000
number of HUI	3994423	208470	411	462

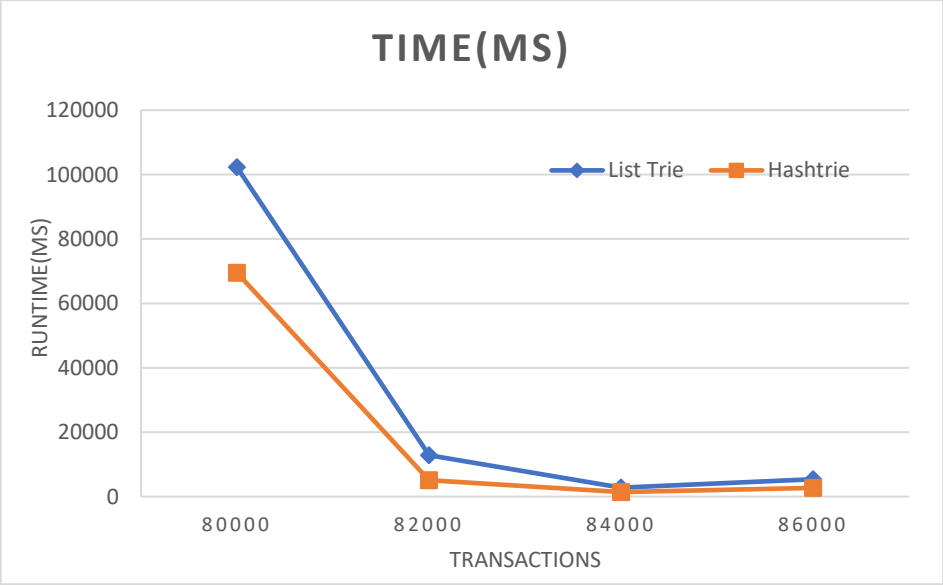

Figure 4 3: Memory usage in the Retail dataset experiment 1

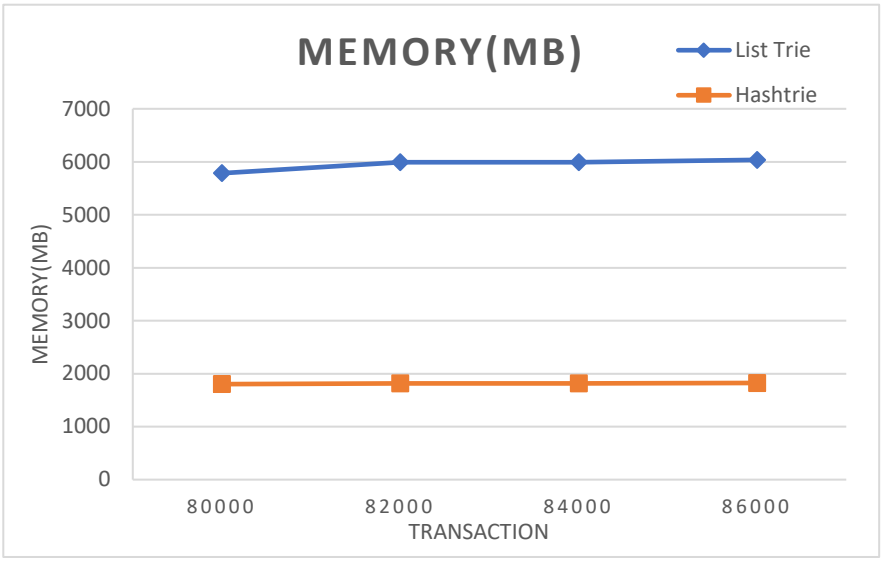

Figure 4.4 : Running time in the Retail dataset experiment 1

Experiment 2: Repeat the same as the first experiment, minimum utility =5000

Min_utility= 500

Table 4.4 number of new HUI itemset Retail exp 2

number transaction	1 - 80000	80001-82000	82001-84000	84001-86000
number of HUI	2177	481	46	71

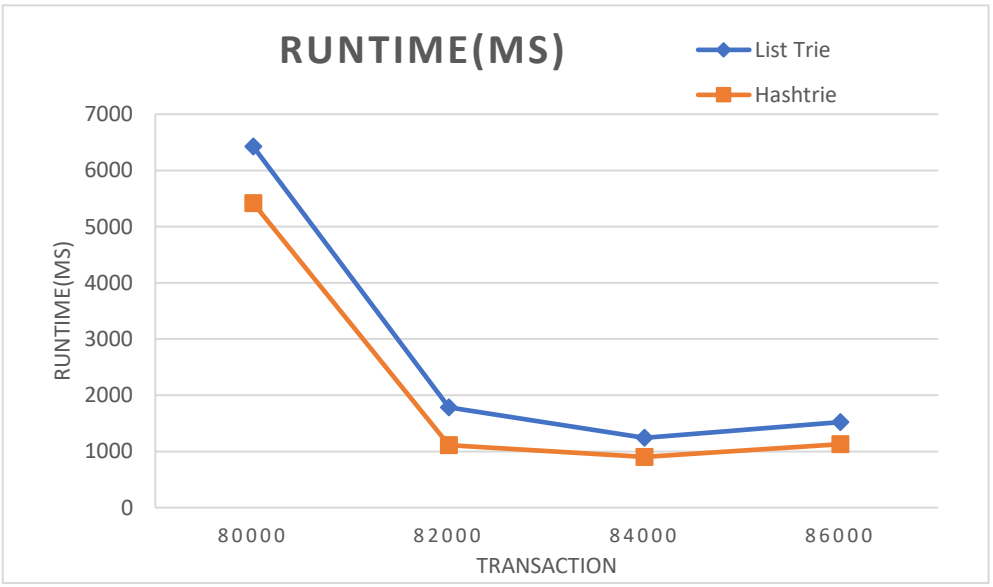

Figure 4.5: Running time in the Retail dataset experiment 2

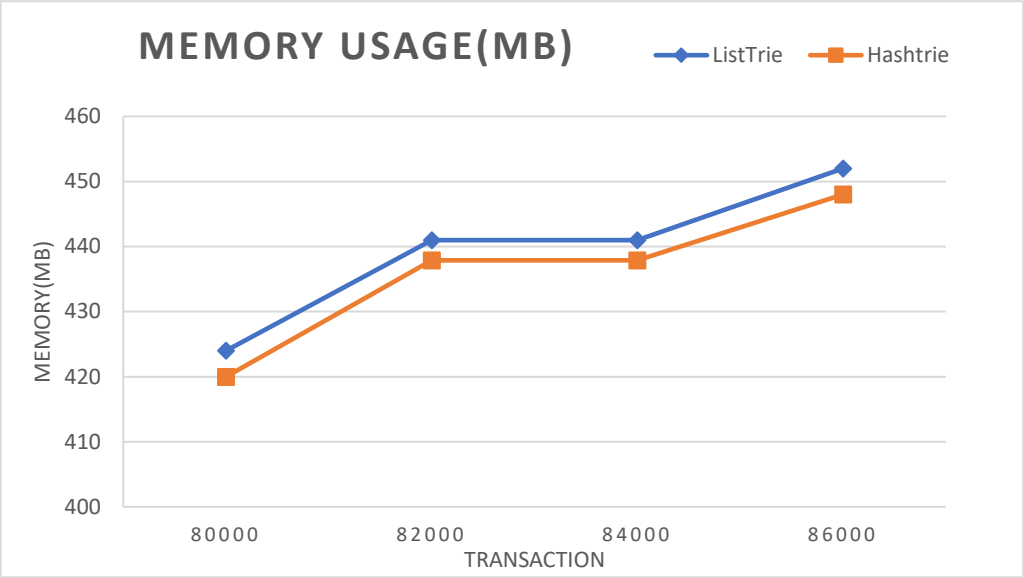

Figure 4.6: Memory usage result in the Retail dataset experiment 2

Discussion the first experiment, at the initial run, there are substantial changes in runtime. In the following run, the runtime between two data structures drops almost the same. The same result when we increase minutil to 5000, results in a smaller number of HUI itemset. However, Hashtrie is faster than ListTrie approximately 5%.

In terms of memory, Hashtrie uses less memory than ListTrie by 3 times, but the difference becomes less significant in the second experiment.

Experiment 3: Assume we only use 80,000 transaction and divided into 4 parts; each part has 20,000 transactions. The initial data will have same volume with the incremental data. Same minimum utility with the first experiment is applied

Table 4.5 number of new HUI itemset Retail exp 2

number transaction	1 - 20000	20001-40000	40001-60000	60001-80000
number of HUI	3531388	6314	153348	303373

Min_utility= 1350

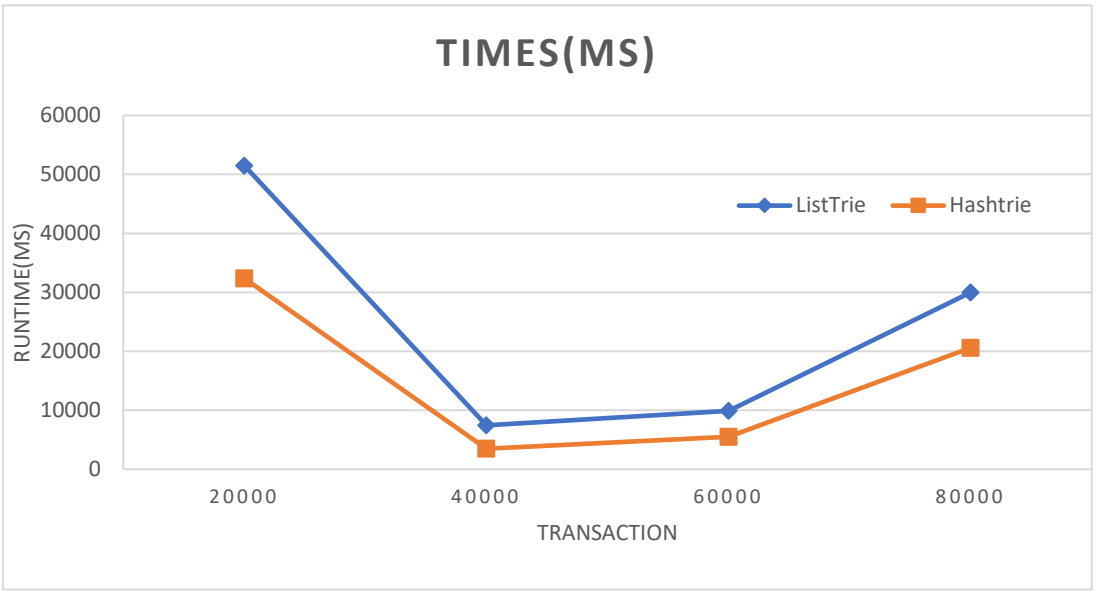

Figure 4 7: Running time result in the Retail dataset experiment 3

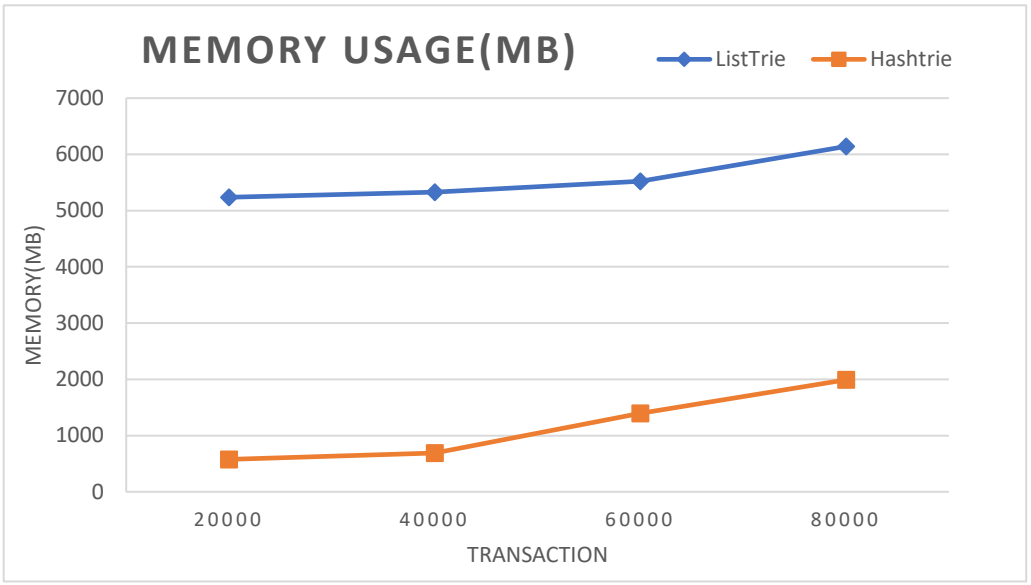

Figure 4 8: Memory usage in the Retail dataset experiment 3

Experiment 4: Repeat the same as experiment 3, minimum utility is set as 5000

Min_utility= 5000

Table 4 6 number of new HUI itemset Retail exp 4

number transaction	1 - 20000	20001-40000	40001-60000	60001-80000
number of HUI	270	641	389	877

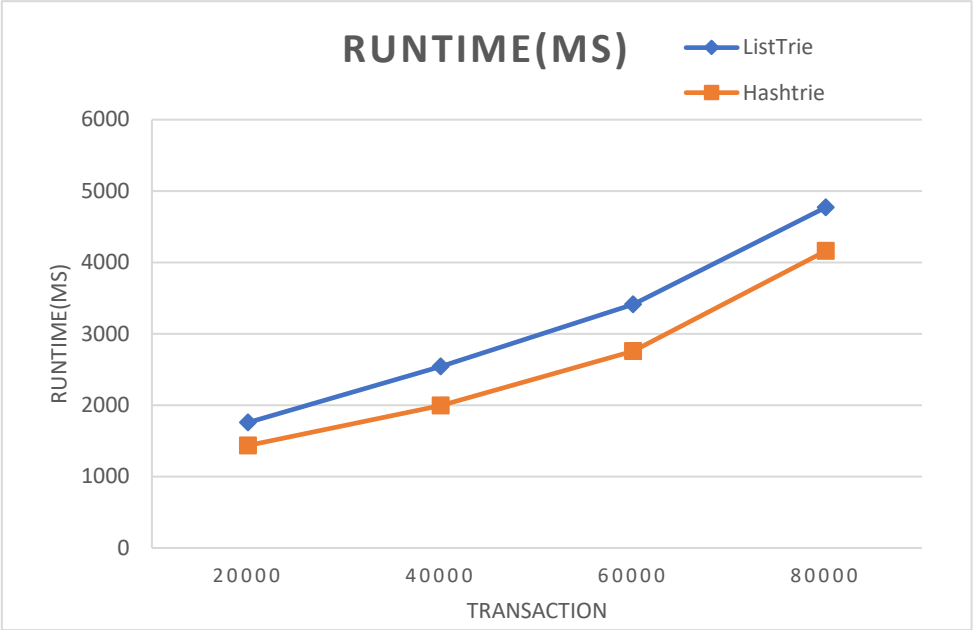

Figure 4.9: Running time result in the Retail dataset experiment 4

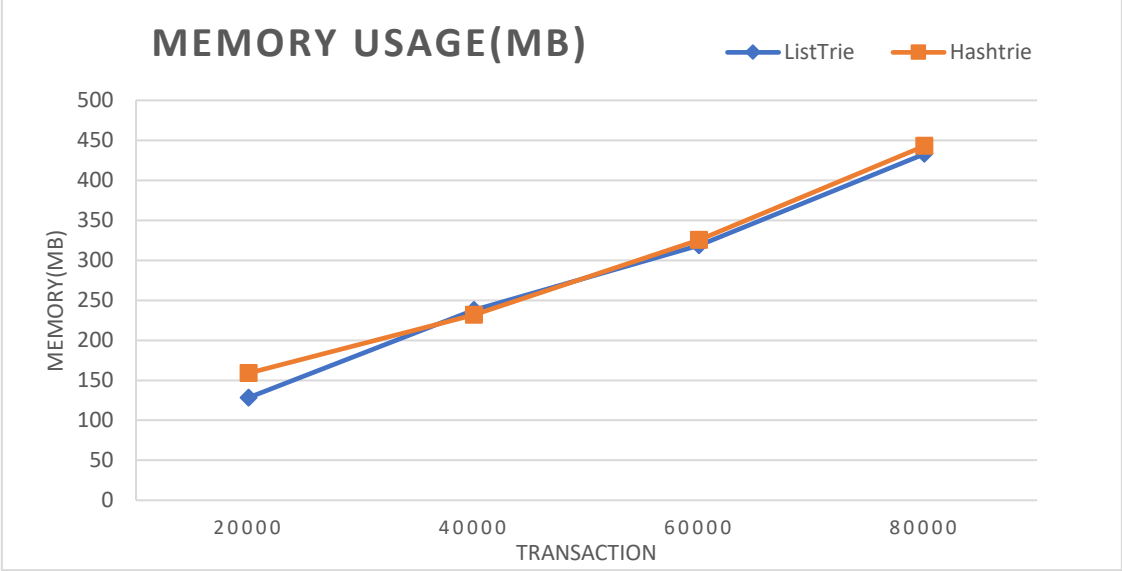

Figure 4.10: Memory usage result in the Retail dataset experiment 4

Overall, Hashtrie has a 5% -10% faster runtime than ListTrie and highly corresponds with the amount of HUI itemsets, the more HUI itemsets the more difference between the two types of data structure can be seen. This appears to be correct with the theory before as changing the Trie data structure does not affect the mining procedure but boosts the organizing process of the HUI itemset in Trie. Therefore, the greater number of itemsets the more significant performance can be seen.

Another aspect that needs to be considered is memory usage. Theoretically, memory consumed by Hashtrie needs to be higher or at least have a close similar result with ListTrie, but this is only shown in the fourth experiment in which the number of HUI is low.

The experiment in the retail dataset yielded the same result as the Mushroom dataset experiment: Hashtrie's execution time is faster than ListTrie in the first run and virtually the same as ListTrie from the second run. In the third and fourth experiments when the dataset is divided equally, we see that Hashtrie is significantly faster than ListTrie in four runs. This was owing to the quantity of new HUI itemsets discovered in each run being different. Hashtrie has a faster insertion procedure than ListTrie when a new itemset is created, owing to rapid indexing and node object allocation.

4.4. Experiment of Chainstore dataset

The test was performed on Chainstore dataset which has 1,112,949 transaction. We will use 1,100,000 transaction to test the performance between two data structure on large data.

Experiment 1: The initial data has 900,000 records, the second data has 100,000, the last is divided into two parts each has 50,000 record, minimum utility 100,000 is applied.

min_utility=100,000

Table 4 7 number of new HUI itemset chainstore exp 4

number transaction	1-900000	900001-1000000	1000001-1050000	1050001-1100000
Number of HUI	15009	2690	407	318

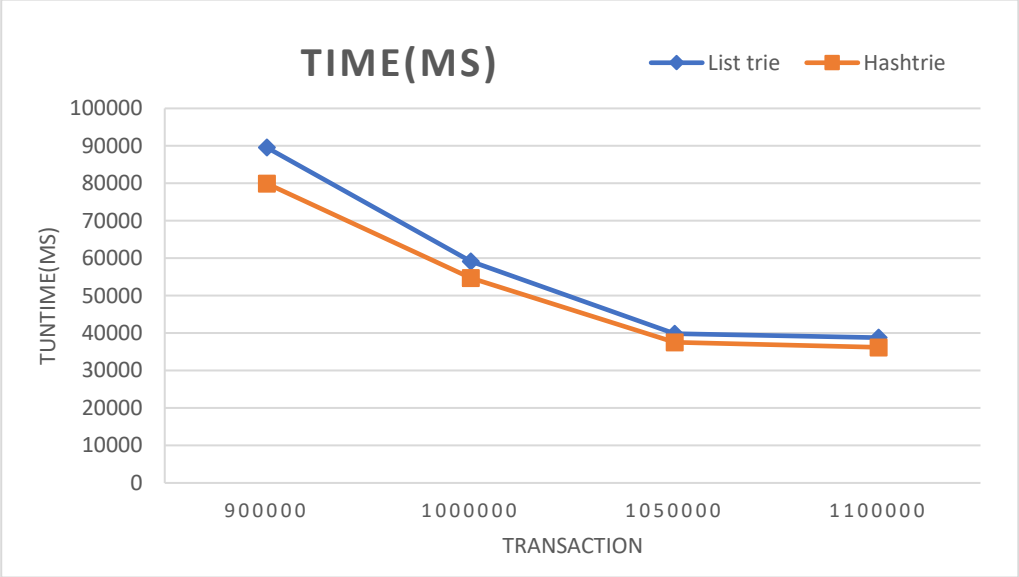

Figure 4.11: Running time in the chainstore dataset experiment 1

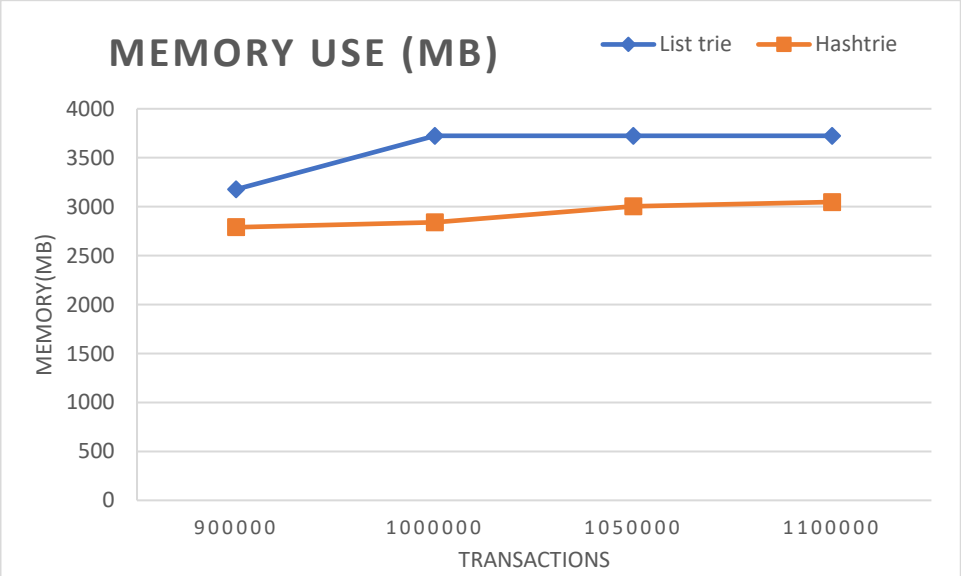

Figure 4 12 : Memory usage in the chainstore dataset experiment 1

Experiment 2: Apply the same dataset from the experiment 1 but this time minimum utility is 200,000

min_utility=200,000

Table 4 8 number of new HUI itemset chainstore exp 2

number transaction	1-900000	900001-1000000	1000001-1050000	1050001-1100000
number of HUI	4457	822	169	113

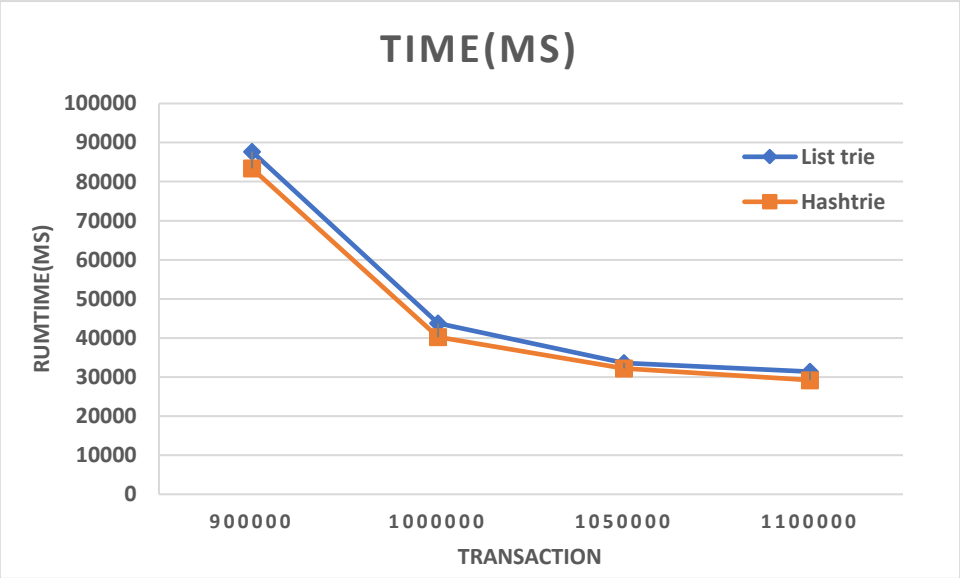

Figure 4.13: Running time in the chainstore dataset experiment 2

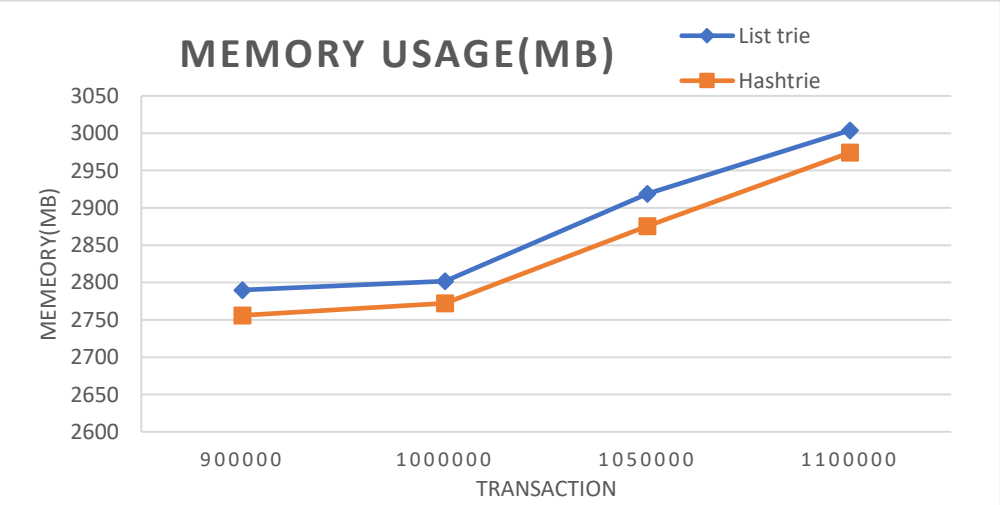

Figure 4.14: Memory usage in the chainstore dataset experiment 2

Discussion: The first experiment with the Chainstore dataset shows similar results found in the Mushroom dataset and the first experiment in the Retail dataset. Hashtrie runs faster in the first run and fairly the same as ListTrie in the following run corresponding to the number of new HUI itemset found.

However, in the second experiment, when the minimum utility is raised to 200,000, fewer HUI itemset is generated, thus reducing the complexity of the Trie. The result hardly shows a significant change between the two types of data structure in both time and memory consumption.

Experiment 3: We used 1,100,000 records divided into equal batches. The first three batches each hold 300,000 records, and the final batch has 200,000 record

min_utility=100,000

Table 4 9 number of new HUI itemset chainstore exp 3

number transaction	1-300000	300001-600000	600001-900000	900001-1100000
number of HUI	3219	4819	6971	3415

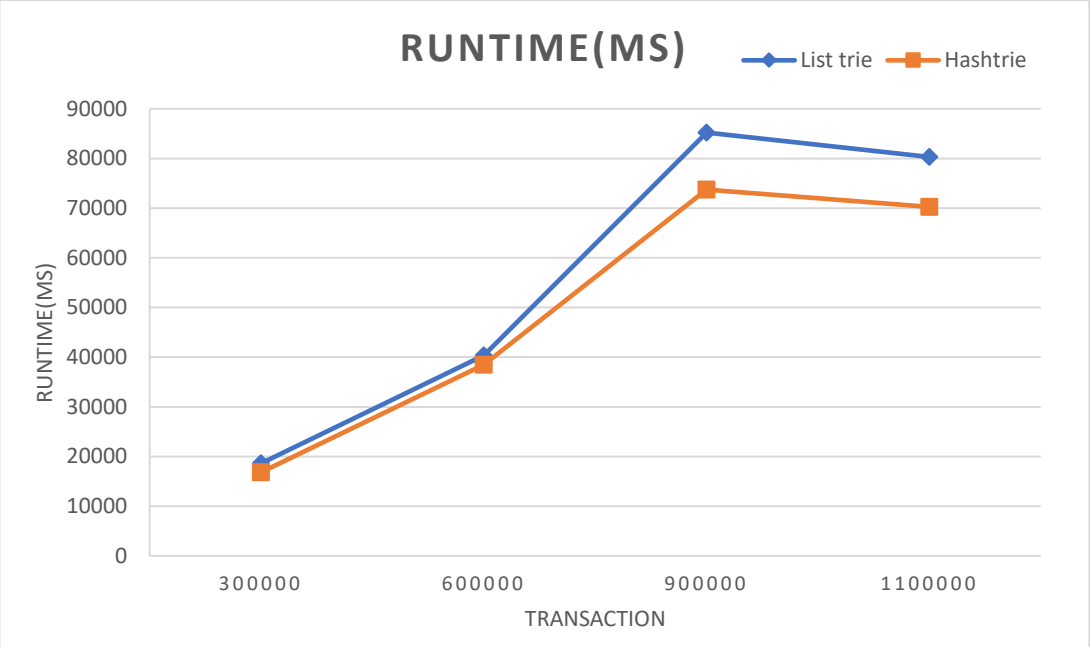

Figure 4.15: Running time in the chainstore dataset experiment 3

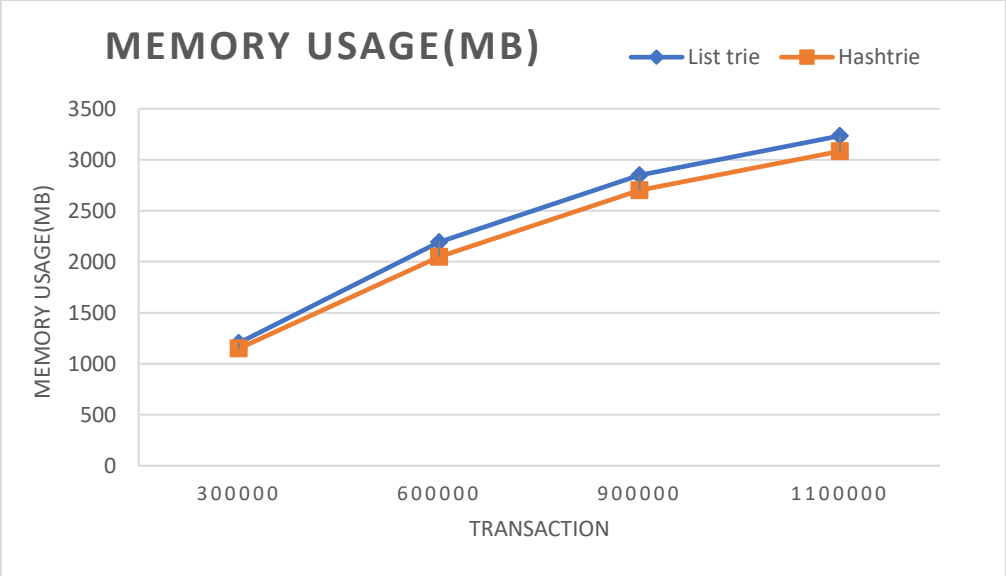

Figure 4.16: Memory usage in the chainstore dataset experiment 3

Experiment 4: Apply the same dataset from the experiment 3 minimum utility is 200,000

min_utility=200,000

Table 4 10 number of new HUI itemset chainstore exp 4

number transaction	1-300000	300001-600000	600001-900000	900001-1100000
number of HUI	736	2780	941	1104

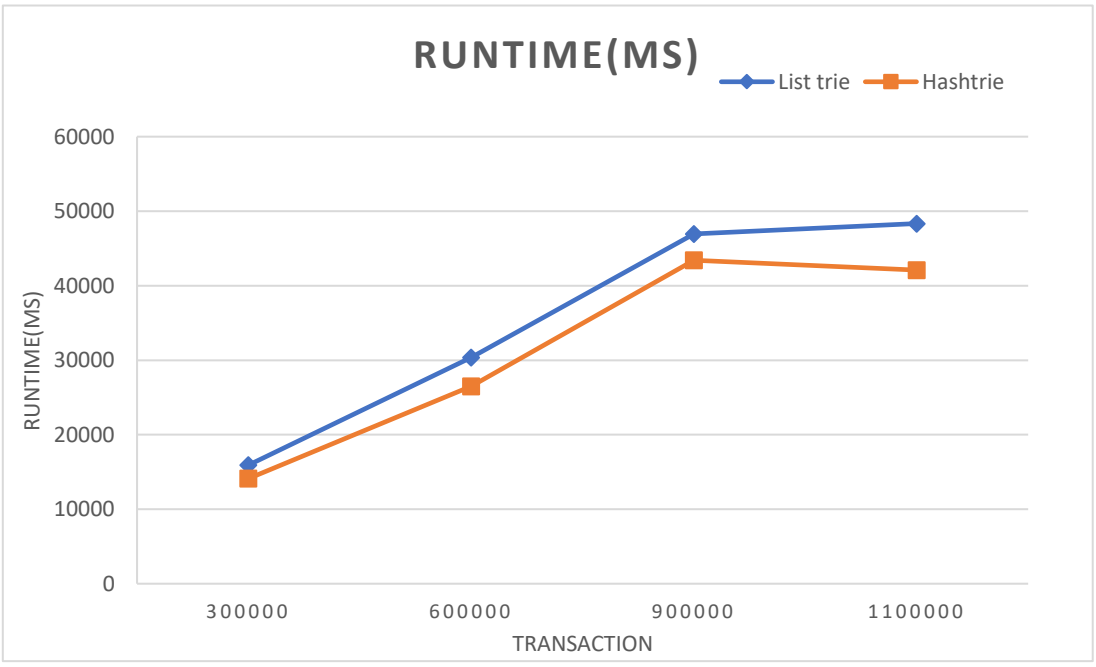

Figure 4.17: Running time in the chainstore dataset experiment 4

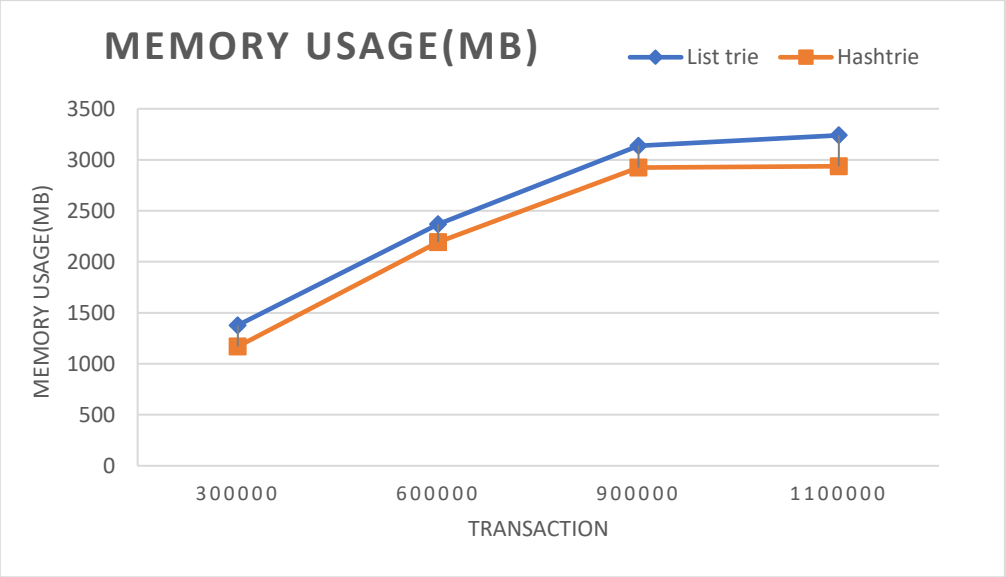

Figure 4.18: Memory usage in the chainstore dataset experiment 4

In the third and the fourth experiment with the Chainstore dataset when each batch is divided equally. The execution time of Hashtrie is faster than ListTrie from 5% to 10%. The number of new HUI itemsets in each run fall around 3000 to 5000 per run, relatively low compared with the result in the Mushroom or Chainstore dataset which has around 900,000 itemsets per run. This led to the overall insignificant performance in both execution time and memory consumption between the two types of data structure as Hashtrie performs better when handling many HUI itemsets. Overall, the Hashtrie structure has a small improvement or is relatively the same as ListTrie when dealing with datasets that yield a little number of HUI.

In general, the data structure suggested in the thesis is quicker than the previous structure, owing to the combination of hash pointer to rapidly allocate and index node to enable fast insert and update HUI itemset in Trie. Experiments show that the Hashtrie performs better when more itemsets are generated from the algorithm in both execution time and memory usage. The performance is little or relatively the same with ListTrie when the number of HUI is small.

CHAPTER 5

DISCUSSION AND EVALUATION

5.1. Conclusion

The content of the thesis corresponded to the outline. The following are the specific outcomes of the thesis:

- Find out about database architecture, extremely helpful item set mining, and high utility item set mining techniques like HUI-Miner, FHM, EIHI
- Research the limitations of the EIHI algorithm, thereby proposing a data structure to increase efficiency when mining high utility itemset on the growing database.
- Hashtrie can greatly reduce memory consumption and the same time boost the performance of the algorithm, making it run faster and more scalable to deploy. Open the possibility of distributed computing for big data.

However, the algorithm performs better when the number of HUI itemsets discovered is big since it can manage a considerably larger number of itemsets placed in the Trie while using fewer resources.

Some experiments in several datasets need to be further explained as there remain some inconsistencies about the running time and memory usage specifically in experiment 4 in the retail dataset as well as experiments 3 and 4 in the chainstore dataset. These experiments have a similar pattern and the memory usage between Hashtrie and ListTrie is very insignificant compared to the previous one. I also came up with a theory that suggested that this behavior is due to the different expansion of the array between HashMap and ArrayList. As HashMap consumes more resources than ArrayList and more computational power.

Another theory suggests that different types of Trie created from mining procedures can affect the construction of Trie. Hashtrie's performance is better when there are many child nodes

since it allows a constant time to locate an element within big array while an ArrayList indexing process usually took a long time and resources when dealing with a big array. In contrast, if the number of child nodes is few or little, then Hashtrie will take up more memory than ListTrie for storing an unnecessary additional byte which greatly reduces the performance of Hashtrie

5.2. Development

Research in pruning strategy and mining high utility itemset in an incremental database, such as reducing the number of database scans if possible.

Testing more on different type of data characteristic, different density, records, number of items.

Research in HashMap and ArrayList performance to further explain the phenomenon mentioned above.

Changing the data structure is insufficient enough to boost the performance of the algorithm, resulting in an insignificant change in running time that can only be observed in some specific type of dataset.

Different strategy in pruning and detecting high utility itemset in the early stage is crucial to look for if want to improve the speed of the mining process significantly.

Research more on different types of Trie structures, its application in the algorithm, and how they will affect the construction procedure between ListTrie and Hashtrie.

However, with the help of Hashtrie data structure, the algorithm can be more scalable to deploy and it is possible to apply in distributed computing to mine big data. [

REFERENCES

[1] *Qu, J.-F., Liu, M., Fournier-Viger (2019). Efficient algorithms for high utility itemset mining without candidate generation. In: Fournier-Viger et al. (eds). High-Utility Pattern Mining: Theory, Algorithms and Applications, Springer, p. 131-160*

[2] *P. Fournier-Viger, C.-W. Wu, S. Zida and V. S. Tseng. "FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning". In Proc. 21st Intern. Symp. on Methodologies for Intell. Syst., pages 83–9. 2014.*

[3] *Fournier-Viger, P., Lin, J. C.-W., Gueniche, T., Barhate, P. (2015). Efficient Incremental High Utility Itemset Mining. Proc. 5th ASE International Conference on Big Data (BigData 2015)*

[4] *Fournier Viger, Philippe & Lin, Chun-Wei & Gueniche, Ted & Bahrate, Prashant. (2015). Efficient Incremental High Utility Itemset Mining.*

[5] *Clifton, Christopher. "data mining". Encyclopedia Britannica, 14 Feb. 2022, https://www.britannica.com/technology/data-mining. Accessed 15 June 2022.*

[6] *David Loshin, Chapter 17 - Knowledge Discovery and Data Mining for Predictive Analytics, Editor(s): David Loshin, In MK Series on Business Intelligence, Business Intelligence (Second Edition), Morgan Kaufmann, 2013, Pages 271-286, ISBN 9780123858894, https://doi.org/10.1016/B978-0-12-385889-4.00017-X. (https://www.sciencedirect.com/science/article/pii/B978012385889400017X)*

[7] *R. Agrawal and R. Srikant. "Fast algorithms for mining association rules in large databases". In Proc. Intern. Conf. Very Large Databases, pages 487–499. 1994. Qu, M. Liu and J.*

[8] Y. Liu, W. Liao, and A. Choudhary A, “Two-Phase algorithm for fast discovery of high utility itemsets”, *in Proceedings of the 9th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining,*2005, pp. 689-695.

[9] C. Lin, W. Gan, P. Fournier-Viger, L. Yang, Q. Liu, J. Frnda, L. Sevcik, M. Voznak, "High utility-itemset mining and privacy-preserving utility mining," *Perspectives in Science,* vol. 7, pp. 74-80, 2016.

[10] T-L. Dam, H. Ramampiaro, K. Nϕrvag, Q-H. Duong, “Towards efficiently mining closed high utility itemsets from incremental databases”, *Knowl. Data Eng,* no 165, pp 13-19. 2019

[11] R. U. Kiran *et al*., "Distributed Mining of Spatial High Utility Itemsets in Very Large Spatiotemporal Databases using Spark In-Memory Computing Architecture," *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 4724-4733, doi: 10.1109/BigData50022.2020.9377946.