Frequent Pattern Mining of Uncertain Data Streams

by

Fan Jiang

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

Master of Science

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

March 2012

Thesis advisor

Dr. Carson K. Leung

Author

Fan Jiang

Frequent Pattern Mining of Uncertain Data Streams

Abstract

When dealing with uncertain data, users may not be certain about the presence of an item in the database. For example, due to inherent instrumental imprecision or errors, data collected by sensors are usually uncertain. In various real-life applications, uncertain databases are not necessarily static, new data may come continuously and at a rapid rate. These uncertain data can come in batches, which form a data stream. To discover useful knowledge in the form of frequent patterns from streams of uncertain data, algorithms have been developed to use the sliding window model for processing and mining data streams. However, for some applications, the landmark window model and the time-fading model are more appropriate. In this M.Sc. thesis, I propose tree-based algorithms that use the landmark window model or the time-fading model to mine frequent patterns from streams of uncertain data. Experimental results show the effectiveness of our algorithms.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my deepest gratitude to my research supervisor Dr. Carson K. Leung. This thesis would not have been possible without his encouragement and academic support. Back when I was an undergraduate student, I took all database and data mining courses with Dr. Leung. His enthusiasm and inspiration make me interested in the research area of data mining. After I finished my undergraduate degree, Dr. Leung strongly encouraged me to stay in this research area, which gave me an opportunity to research on this M.Sc. thesis.

In addition, I also acknowledge the Database and Data Mining Laboratory, Department of Computer Science, and University of Manitoba for providing me a comfortable research environment, plenty of research equipments, and rapid technical support. Particularly, I thank my lab members, Syed K. Tanbeer, Yaroslav Hayduk, Juan J. Cameron, Lijing Sun, and Monjur M. Dewan.

Moreover, I would like to thank my thesis examination committee members, Dr. Michael Domaratzki and Dr. Xikui Wang, for their precious time to read and examine my thesis. I would also like to thank Dr. James E. Young to chair my thesis defence.

Last but not the least, I thank my parents for their support and understanding throughout my graduate study.

Fan Jiang
B.Sc.(Honours), The University of Manitoba, Canada, 2010

The University of Manitoba
March 2012

This thesis is dedicated to my parents.

Chapter 1

Introduction

Frequent pattern mining is one of the most important research areas in the database and data mining field. Agrawal et al. [AS94] defined frequent pattern mining as searches for implicit, previously unknown, and potentially useful pieces of information—in the form of frequently occurring sets of items (also known as *patterns*)—that are embedded in the data. It can be applied in various real-life applications. For example, it finds from shopping market basket data those sets of popular merchandise items, which in turn helps reveal shopper behaviour. Since frequent pattern mining was introduced, numerous studies have been conducted [OSS02, LLN03, Leu04, Bod05, LIC08, YLL08, PG09, PLL$^+$10]. Note that most of these studies focus on mining traditional *precise data*.

Precise data is the type of data that users are certain about the presence of items in transactions in these databases. One of the earliest frequent pattern mining algorithms on precise databases is called the *Apriori* algorithm [AS94] proposed by Agrawal and Srikant in 1994. It follows a generate-and-test approach. Apriori and its

extension algorithms (e.g., CAP [NLHP98] and U-Apriori [CKH07]) generate candidates first then check the support values (occurrences) of those candidates within the database. Users need to define a support value (called minimum support threshold) before the algorithm execution. In the end, all candidates with a support value that greater than or equals to the minimum support threshold will be returned as frequent patterns.

One of the important observations from these Apriori-based algorithms is that the candidate generation process is a bottleneck of all Apriori-based algorithms [HPY00]. To solve this problem, Han et al. [HPY00] designed a tree-based frequent pattern mining algorithm called *FP-growth* (where FP stands for Frequent Pattern). This algorithm requires two database scans. After the first scan, FP-growth finds all frequent level-1 patterns (i.e., frequent pattern with single items, also known as singletons). After that, it scans the database the second time to construct a tree data structure called an FP-tree. This tree data structure contains all information in the database. In other words, we do not need to read the database any more after the FP-tree is constructed. We can mine frequent patterns based on this tree only. A recursive process called frequent pattern growth will be applied after that. It processes from the bottom (i.e., the leaf level) of the tree up to the root using a test-only approach (i.e., only test the support for each pattern). As a result, we completely avoid any candidate generation. This tree-based design of data mining algorithm is extended to many other algorithms (e.g., UF-growth [HPY00], SOP-tree [KD11]) by various researchers in the field. As tree-based mining is an efficient method for mining frequent patterns, the algorithms we propose in this M.Sc thesis are also tree-based algorithms,

which do not require candidate generation.

Algorithms we have discussed so far mine from traditional *static databases*. Nowadays, the automation of measurements and data collection is producing tremendously huge volumes of data. For instance, the development and increasing use of a large number of sensors (e.g., electromagnetic, mechanical, and thermal sensors) for various real-life applications (e.g., environment surveillance, manufacture systems) have led to data streams [MSL08, Cuz09, CGWD10]. To discover useful knowledge from these streaming data, several mining algorithms [GZK05, CE06, CNOT07] have been proposed. In general, mining frequent patterns from dynamic data streams [JG06, GBK10] is more challenging than mining from traditional static transaction databases due to the following characteristics of data streams:

1. *Data streams are continuous and unbounded.* As such, we no longer have the luxury to scan the streams multiple times. Once the streams flow through, we lose them. We need some techniques to capture important contents of the streams. For instance, *sliding windows* capture the contents of a fixed number (w) of batches (i.e., the w most recent batches) in the streams. Alternatively, the *landmark windows* capture contents of all batches after the landmark (i.e., sizes of windows keep increasing with the number of batches). Similarly, the *time-fading windows* also capture contents of all the batches but weight recent data heavier than older data (i.e., monotonically decreasing weights from recent to older data). We will discuss more about these windowing techniques in Section 2.4.

2. *Data in the streams are not necessarily uniformly distributed.* As such, a cur-

rently infrequent pattern may become frequent in the future, and vice versa. We have to be careful not to prune infrequent patterns too early; otherwise, we may not be able to get complete information such as supports of some patterns (as it is impossible to recall those pruned patterns).

In this thesis, we propose data stream mining algorithms that successfully overcome the above two challenges.

Regarding the problem of data stream mining, there is one more problem we trying to solve in this thesis, which is uncertain data mining. Many existing mining algorithms discover frequent patterns from *precise data* (in either static databases [EZ09, HPY00] or dynamic data streams [GHP+04, YCLZ04]), in which users definitely know whether an item is present in, or absent from, a transaction in the data. However, there are situations in which users are uncertain about the presence or absence of items. For example, due to dynamic errors (e.g., inherited measurement inaccuracies, sampling frequency), streaming data collected by sensors may be uncertain. As such, users may highly suspect but cannot guarantee that an item x is present in a transaction t_i. The uncertainty of such suspicion can be expressed in terms of existential probability $P(x, t_i) \in (0, 1]$, which indicates the likelihood of x being present in t_i in probabilistic data. With this notion, every item in t_i in (static databases or dynamic streams of) precise data can be viewed as an item with a 100% likelihood of being present in t_i. A challenge of handling these uncertain data is the huge number of "possible worlds" (e.g., there are two "possible worlds" for an item x in t_i: (i) $x \in t_i$ and (ii) $x \notin t_i$). Given q independent items in all transactions, there are $O(2^q)$ "possible worlds" [Leu11].

Over the past few years, several data mining algorithms have been proposed to discover frequent patterns from uncertain data. However, most of them (e.g., UF-growth [LMB08], UH-Mine [ALWW09], U-Eclat [CGG10], UV-Eclat [LS11]) mine frequent patterns from static databases but not dynamic streams of uncertain data. For the algorithms that mine from data streams (e.g., UF-streaming [LH09]), they mostly use sliding windows. While the use of sliding windows is useful for situations where users are interested in discovering frequent patterns from a fixed-size time window (e.g., frequent patterns observed in the last 24 hours), there are also other situations where users are interested in a variable-size time window capturing all historical data (with or without stronger preference on recent data than older one). In these situations, other window models (e.g., the time-fading window model or the landmark window model) are needed.

1.1 Thesis Statement

Motivated by the problems and solutions from the previous section, my **M.Sc. thesis statement** is as follows. I propose tree-based data mining algorithms to perform the following tasks:

- Retrieve frequent patterns from uncertain data,
- retrieve frequent patterns from data streams,
- store information from potentially infinite numbers of batches in data streams by applying the time-fading window models and the landmark window models,
- reduce the memory usage and speed up the execution time, as well as

Table 1.1: Our Proposed Algorithms vs. the Most Relevant Algorithms

	Tree-based mining	Uncertain data mining	Data stream mining	Handle infinite streams
Apriori [AS94]				
FP-growth [HPY00]	√			
U-Apriori [CKH07]		√		
UF-growth [LMB08]	√	√		
FP-streaming [GHP+04]	√		√	
UF-streaming [LH09]	√	√	√	
Our proposed LUF-streaming	√	√	√	√
Our proposed TUF-streaming	√	√	√	√

- design a data structure so that it can correctly handle the delayed data streams.

In this thesis, we first propose an uncertain data stream mining algorithm, called *LUF-streaming*, which uses the landmark window model. Then, we propose a family of uncertain data stream mining algorithms, named *TUF-streaming*, which use the time-fading window model. Specifically, we provide three different versions of TUF-streaming (Naïve, space-saving, and time-saving). Each one of them applies the time-fading window model in a different way. We also develop an additional version for "delayed" uncertain data streams. Table 1.1 shows the comparison between our algorithms and some existing algorithms. Experimental results in Chapter 4 show the effectiveness of all our proposed algorithms.

1.2 Problem Description and Thesis Contributions

Most existing uncertain data stream mining algorithms only apply sliding window models. However, the sliding window model only records recent data in a fixed-size

window and discards old data "outside" the window. In real-life cases, a lot of users do not want to completely discard the old data. Based on that, some logical questions are:

1. Are there any other models than the sliding window model that we can use?
2. Can we apply other windowing models to discover frequent patterns from dynamic streams of uncertain data?
3. How to construct a data structure that helps discover frequent patterns from dynamic streams of uncertain data?
4. How does the new data structure efficiently store more information than applying the sliding window model?
5. How can we handle potentially infinite streams of incoming data?
6. Will the mining result stay the same when we apply other window models?
7. How will other window models affect the mining algorithm in terms of runtime performance?
8. How will other window models affect the mining algorithm in terms of memory usage?

In response to all above questions, we explore on different stream data handling methods (i.e., the sliding window model, the time-fading window model and the landmark window model), which show that alternatives to sliding windows can be used for mining streams of uncertain data. We are certain that both the time-fading

window models and the landmark window models can be used to discover frequent patterns from dynamic streams of uncertain data. Then, we focus the tree-based data structures. Since we need to process data streams (which have no upper bound in terms of numbers of data streams), the memory usage is our first priority. Among all data structures, tree-based data structures give us good performance in terms of memory usage.

Hence, in this M.Sc. thesis, we propose algorithms for discovering useful knowledge from streams of uncertain data using the time-fading window models and the landmark window models. Our **key contributions** are:

1. the proposal and maintenance of a tree-based structure in capturing the frequent patterns discovered from batches of transactions in dynamic streams when using the time-fading window model and the landmark window models;

2. the design of tree-based stream mining algorithms that (a) use such a tree structure for discovering and storing frequent patterns and (b) do not require the traversal; and update of all tree nodes; and

3. analytical and experimental evaluation of these algorithms.

1.3 Thesis Organization

This thesis is organized as follows. The next chapter gives background information and related work. We first describe some existing precise data mining algorithms and uncertain data mining algorithms and their differences. We then introduce some existing data stream mining algorithms with focus on uncertain data stream mining.

We also compare and contrast three different window models (namely the sliding window model, the landmark window model and the time-fading window model).

Afterwards, we start describing our contribution of this thesis in Chapter 3. In the chapter, we start from proposing the first algorithm, called *LUF-streaming*, which we designed to mine frequent patterns from uncertain data streams. LUF-streaming applies the landmark window model for storing and processing dynamic databases. Part of this work has been published in a refereed paper in the *Proceedings of the 15th International Database Engineering and Applications Symposium (IDEAS 2011)* [LJH11]. We also design a family of algorithms, called *TUF-streaming*, which use the time-fading window model for uncertain data stream mining. Specifically, we start from a naïve version of TUF-streaming. Then, we come up two improvements that save memory space and runtime. We further extend the algorithm to handle "delay" in data streams. The work on TUF-streaming has been partially published as a refereed paper in the *Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2011)* [LJ11].

Both analytical evaluation and experimental evaluation of our proposed algorithms will be provided in Chapter 4. Finally, in Chapter 5, we present the conclusion of this thesis and future research.

Chapter 2

Background and Related Work

In this chapter, we provide background materials and related work that are relevant to this thesis. First, we present the Apriori algorithm [AS94] and the tree-based FP-growth algorithm [HPY00] for mining frequent patterns from traditional static databases of *precise* data. We explain the differences between the Apriori-based and tree-based algorithms, and explain why we choose the tree-based approach other than Apriori-based approach for this thesis. Then, we present algorithms (e.g., U-Apriori [CKH07] and UF-growth [HPY00]) for mining frequent patterns from static databases of *uncertain* data. Moreover, to provide related works on mining frequent patterns from *dynamic* streams, we also present FP-streaming [GHP$^+$04] (a stream data mining algorithm for precise data) and UF-streaming [LH09] (a stream data mining algorithm for uncertain data). Furthermore, since UF-streaming uses the sliding window model to store data but we want to explore other windowing techniques, we discuss and compare three windowing techniques (the sliding window, the landmark window and the time-fading window).

2.1 Precise Data Mining Algorithms

Precise data are traditional types of data that users are certain about the presence of items in transactions in these databases. To mine frequent patterns from precise databases, two famous algorithms are Apriori [AS94] and FP-growth [HPY00].

2.1.1 The Apriori Algorithm

Agrawal and Srikant [AS94] proposed the Apriori algorithm in 1994 for mining *precise static* databases. Every pattern in databases has a support value, which describes the number of occurrences of the pattern. A pattern (or set of items) is considered as frequent pattern if and only if the support value of this pattern is greater than or equal to user-specified *min*imum *sup*port threshold (*minsup*). The Apriori algorithm follows a bottom-up generate-and-test framework to mine frequent patterns from precise databases. It first generates candidate patterns with single item (i.e., singleton patterns). This group of candidates is called C_1 (a set of *c*andidate 1-patterns). A k-pattern with the set is a pattern containing k domain items. For example, $\{a\}$, $\{b\}$ ... $\{e\}$ in Figure 2.1(a) are examples of candidate 1-patterns in C_1. Then, the algorithm finds frequent patterns from C_1 by comparing the support value of each singleton with the minimum support threshold. The algorithm uses C_1 to find L_1 (a set of *L*arge/frequent 1-patterns). From L_1, the algorithm generates C_2, which contains all possible level-2 patterns (patterns with two items). Similarly, the algorithm finds frequent patterns from C_2 and group L_1. This process is repeated until there are no more candidates. See Example 2.1.

Example 2.1 Let us consider the transaction database shown in Table 2.1. Let

Table 2.1: Transaction Database Example

Transactions	Contents
t_1	{a, c, d}
t_2	{b, c, e}
t_3	{a, b, c, e}
t_4	{b, e}

Pattern	Sup Value
{a}	2
{b}	3
{c}	3
{d}	1
{e}	3

(a) C_1

Pattern	Sup Value
{a}	2
{b}	3
{c}	3
{e}	3

(b) L_1

Pattern	Sup Value
{a, b}	1
{a, c}	2
{a, e}	1
{b, c}	2
{b, e}	3
{c, e}	2

(c) C_2

Pattern	Sup Value
{a, c}	2
{b, c}	2
{b, e}	3
{c, e}	2

(d) L_2

Pattern	Sup Value
{b, c, e}	2

(e) C_3

Pattern	Sup Value
{b, c, e}	2

(f) L_3

Figure 2.1: The Apriori Mining (Example 2.1).

us assume the user-specified minimum support threshold is 2. The Apriori algorithm first generates C_1 as shown in Figure 2.1. Then it counts the occurrences of each items in C_1 to get support values. In this example, the support value of $\{d\}$ is 1 which is smaller than minimum support threshold. Hence, in L_1, there are only 4 items (candidate 1-patterns) remaining. The Apriori algorithm generates candidate

2-patterns based on the frequent 1-patterns in L_1. Observe from Figure 2.1, after the counting of occurrences of all candidate 2-patterns in C_2, only 4 of them are frequent (which form L_2). This process is repeated until there is no more candidates. In this example, the algorithm stops at L_3. ■

Note that each step of the Apriori algorithm uses a candidate generation process. However, not every candidate generated by the algorithm is necessarily frequent. For example in Example 2.1, C_2 was generated with six candidate 2-patterns, but only four of them are frequent. Experimental results of Han et al. [HPY00] showed that this candidate generation process is a bottleneck of the Apriori-based algorithms.

2.1.2 The FP-growth Algorithm

To solve the bottleneck problem of Apriori-based algorithms, Han et al. [HPY00] proposed the FP-growth algorithm in 2000, which avoids the candidate generation process. There are two main processes in the FP-growth algorithm: (1) constructing an FP-tree, and (2) *growing* frequent patterns. The FP-tree is an extended prefix-tree structure that captures the contents of databases. The algorithm needs to scan the database twice to construct the FP-tree. In the first scan, it only looks for singletons. All infrequent singletons are removed and all frequent singletons are sorted in a decreasing order of their support values in attempt to reduce the size of the potential FP-tree. Then, the algorithm scans the database the second time to construct an FP-tree. Once the FP-tree is constructed, no more database scans are needed. The FP-tree contains all information in the whole database. All frequent patterns can then be found by a recursive growing process from each branch of the FP-tree. This

Table 2.2: Transaction Database Example

Transactions	**Contents**
t_1	{a, h}
t_2	{a, b}
t_3	{a, b, c, e}
t_4	{a, b, c, d, f}
t_5	{a, b, d, g}

growing process can be described as follows. For each frequent singleton x in the FP-tree, the algorithm forms a project database for x (all transactions having the singleton x as prefix). Then, a sub-tree is built based on this projected database called the $\{x\}$-projected tree. After that, the same process is applied to this $\{x\}$-projected tree for building a smaller sub-tree—namely, the $\{x, y\}$-projected tree. This operation is applied recursively until no more projected trees can be built. It is easy to observe that this is a divide-and-conquer approach. For a better understanding of the FP-growth mining process, see Example 2.2 for the construction of the FP-tree to the recursive mining of frequent patterns.

Example 2.2 Let us consider the following transaction database showed in Table 2.2. Assume the user specifies the minimum support threshold of 2. In the first step of the FP-growth algorithm, we compute the support value for each singleton in the transaction database. The results are (in the form of "items:support value"): $\{a : 5\}, \{b : 4\}, \{c : 2\}, \{d : 2\}, \{e : 1\}, \{f : 1\}, \{g : 1\}, \{h : 1\}$. Since $\{e : 1\}, \{f : 1\}, \{g : 1\}, \{h : 1\}$ do not have a support value ≥ 2 (*minsup*), we can remove them. After sorting the remaining "items:support value" pairs, we get $\{a : 5\}, \{b : 4\}, \{c : 2\}, \{d : 2\}$. Based on this set of frequent singletons, we perform the second scan against the database. Then, we get the global FP-tree as shown in

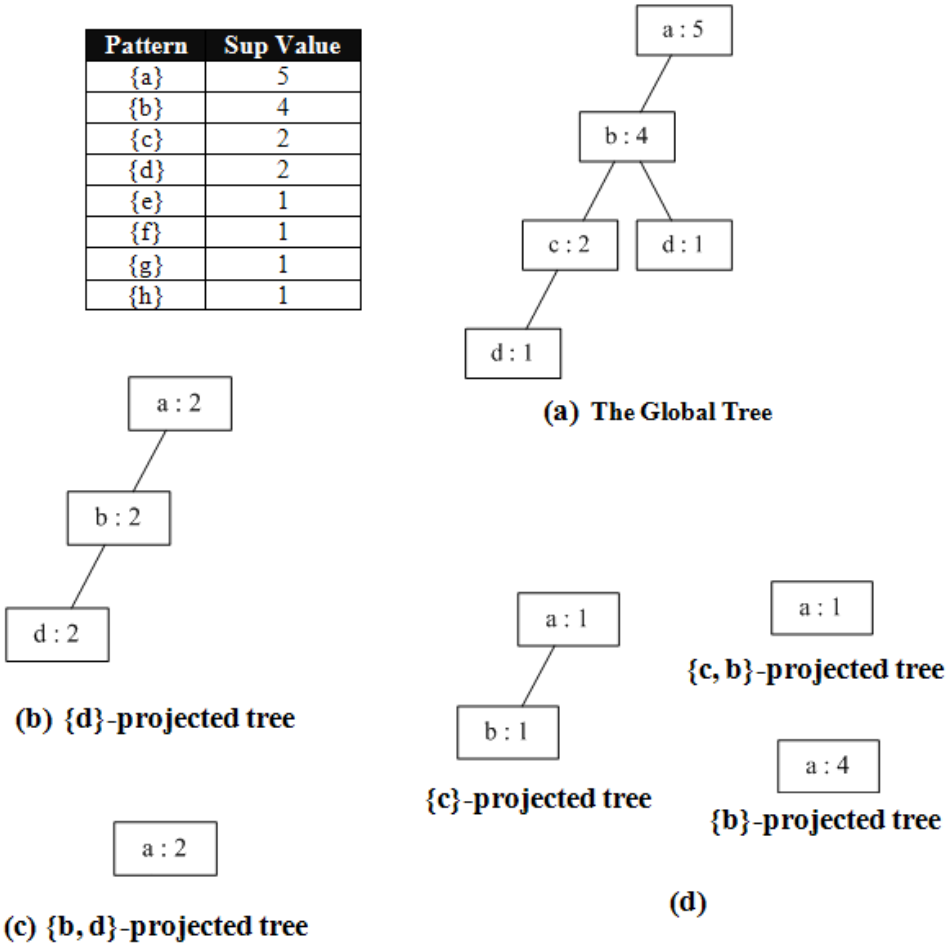


Figure 2.2: The FP-growth Mining (Example 2.2).

Figure 2.2(a). The recursive growing process starts after the FP-tree is constructed. First, we build the $\{d\}$-projected tree as shown in Figure 2.2(b). Note that the support value of $\{c\}$ is less than minimum support threshold. Hence, we remove it from $\{d\}$-projected tree. We next build the $\{b, d\}$-projected tree as shown in Figure 2.2(c). With the same approach, we form the projected databases and projected trees for

$\{c\}$, $\{c, b\}$ and $\{b\}$ (shown in Figure 2.2(d)). Finally, we find all the frequent patterns as follows: $\{a\}$*:5*, $\{b\}$*:4*, $\{c\}$*:2*, $\{d\}$*:2*, $\{a,\ b\}$*:4*, $\{a,\ c\}$*:2*, $\{a,\ d\}$*:2*, $\{b,\ c\}$*:2*, $\{b,\ d\}$*:2*, $\{a,\ b,\ c\}$*:2*, $\{a,\ b,\ d\}$*:2*. ■

By avoiding candidate generation, the FP-growth algorithm successfully improves the performance [HPY00]. This is one of the important reasons that we decided to use the tree-based algorithm for uncertain data mining in this M.Sc. thesis.

2.2 Uncertain Data Mining Algorithms

Uncertain data is the data type where users are not certain about the presence of an item in the database. For example, due to inherent instrumental imprecision or errors, data collected by sensors are usually uncertain. Items in each transaction in this type of data are usually associated with existential probability values. Each existential probability value indicates the "likelihood" of that item to appear in the uncertain database. The existential probability can be defined as $P(x, t_i)$, where x represent an item, t_i is the index of the transaction. If we use the "possible world" interpretation of uncertain data [LCH07, CKH07, LMB08, YMD+09], $P(x, t_i)$ represents two possible worlds: (1) the possible where $x \in t_i$, and (2) the possible world where $x \notin t_i$. The probability of the first possible world (W_1) is $P(x, t_i)$, and the second possible world (W_2) is $1 - P(x, t_i)$.

Furthermore, if we consider two transactions t_1 and t_2 together and they all contain item x, then there are four possible worlds: (1) x is in both t_1 and t_2, (2) x is in t_1 but not in t_2, (3) x is in t_2 but not in t_1, and (4) x is in neither t_1 nor t_2. To compute the probability of each possible world, we have:

- $prob(W_1) = P(x, t_1) \times P(x, t_2)$,
- $prob(W_2) = P(x, t_1) \times [1 - P(x, t_2)]$,
- $prob(W_3) = [1 - P(x, t_1)] \times P(x, t_2)$, and
- $prob(W_4) = [1 - P(x, t_1)] \times [1 - P(x, t_2)]$.

Similarly, if there are two independent items x and y in the same transaction t_i, we have the following four possible worlds: (1) $x, y \in t_i$, (2) $x \in t_i$, $y \notin t_i$, (3) $x \notin t_i$, $y \in t_i$, and (4) $x, y \notin t_i$. The probability of each possible world is as follows:

- $prob(W_1) = P(x, t_1) \times P(y, t_1)$,
- $prob(W_2) = P(x, t_1) \times [1 - P(y, t_1)]$,
- $prob(W_3) = [1 - P(x, t_1)] \times P(y, t_1)$, and
- $prob(W_4) = [1 - P(x, t_1)] \times [1 - P(y, t_1)]$.

To generalize, the probability of W_j to be the true world can be computed by the following equation:

$$prob(W_j) = \prod_{i=1}^{n} \left(\prod_{x \in t_i in W_j} P(x, t_i) \times \prod_{y \notin t_i in W_j} [1 - P(y, t_i)] \right); \tag{2.1}$$

and, the expected support of X, which represents the support value of an *item set* (pattern) X in possible world W_j over all possible worlds, can be computed by the following:

$$expSup(X) = \sum_{j} [sup(X) \in W_j \times prob(W_j)]\,. \tag{2.2}$$

Equation (2.2) can be simplified to the following equation [DYMV05]:

$$expSup(X) = \sum_{i=1}^{n} \left(\prod_{x \in X} P(x, t_i) \right). \tag{2.3}$$

Hence, if the value of $expSup(X) \geq$ the user-specified minimum support threshold *minsup*, we know that item set (pattern) X is frequent.

In the following sections, we describe two algorithms that apply the above equations. These algorithms are U-Apriori [CKH07] (an Apriori-based uncertain data mining algorithm) and UF-growth [LMB08] (a tree-based uncertain data mining algorithm).

2.2.1 The U-Apriori Algorithm

U-Apriori was proposed by Chui et al. [CKH07, CK08]. Similar to the Apriori algorithm that we introduced in Section 2.1.1, it follows a bottom-up generate-and-test framework to mine frequent patterns. However, instead of incrementing the support counts of candidate patterns by their *actual* support, the U-Apriori algorithm increments the support counts of candidate patterns by their *expected* support under the uncertainty model. In the final step of the algorithm, if the support value of a pattern is greater than or equal to the user-specified minimum support threshold (*minsup*), the pattern is considered frequent.

Since U-Apriori inherits the Apriori-based approach from Apriori Algorithm, it also has the same bottleneck—the candidate generation process. This issue becomes even more significant in the case of mining uncertain data because the same pattern with different support values are considered as different candidates. For example,

the pattern $\{a\}$ with support value 0.55 is considered to be different from pattern $\{a\}$ with support value 0.95. This means the candidate generation process creates a large C_i. However, in the end, only a few may be frequent and in group L_i. To avoid this situation, Chui et al. designed a method to combine local trimming, global pruning and single-pass patch-up together (called LGS-trimming). To elaborate, the algorithm first trims those items with expected support values less than *minsup* (i.e., trims an item x if $expSup(x) < minsup$) from the original database. Then, the algorithm mines frequent patterns from the resulting trimmed database. During the mining process, the algorithm performs a pruning process simultaneously. Pattern x is pruned if and only if its support value ($expSup(x)$) plus its upper bound of an estimation error ϵx is lower than the *minsup*. To avoid missing frequent patterns, in the end, the algorithm performs the patch up by scanning the original database again to verify those trimmed and pruned patters are truly infrequent. The LGS-trimming strategy helps in reducing the cost of candidate generation process, however, the bottleneck (candidate generation process) still exists.

2.2.2 The UF-growth Algorithm

Leung et al. [LMB08] proposed UF-growth. UF-growth inherits the tree-based design from FP-growth, and was extended for uncertain data mining. To discover frequent patterns, UF-growth constructs a UF-tree to capture contents of uncertain data. Each tree node keeps an item $\{x\}$, its existential probability $P(x, t_i)$, and its occurrence count. The UF-tree is constructed in a similar fashion to that of the FP-tree except that nodes in the UF-tree are merged and shared only if they

Table 2.3: Uncertain Database Example

Transactions	**Contents**
t_1	{a:0.8, d:0.62, e:0.61}
t_2	{a:0.8, c:0.71, d:0.61, e:0.611}
t_3	{b:0.78, c:0.76}
t_4	{a:0.8, d:0.62, e:0.61}
t_5	{b:0.78, c:0.76, d:0.05}
t_6	{b:0.78}

represent the same x and $P(x, t_i)$. Once the UF-tree is constructed, UF-growth extracts appropriate tree paths to mine frequent patterns using the "possible world" interpretation. Again, a pattern X is frequent if its expected support ($expSup(X)$) $\geq$ the user-specified minimum support threshold. See the following example.

Example 2.3 Let us consider the following transaction database shown in Table 2.3. Assume user specifies the minimum support threshold as 1.0. Similar to FP-growth, the first step of the UF-growth algorithm is calculating the support value for each singleton in the transaction database. The results are (in the form of "items:support value"): *a:2.4*, *b:2.34*, *c:2.23*, *d:1.9*, *e:1.831*. Since every singleton has a support value $\geq 1.0 = minsup$, nothing is pruned. Based on this set of frequent singletons, we perform the second scan against the database. Then, we get a UF-tree as shown in Figure 2.3(a).

The recursive growing process starts right after the UF-tree is constructed. First, we build the $\{e\}$-projected tree as shown in Figure 2.3(b). In the UF-tree, the support value of $\{c, e\}$ is less than minimum support threshold. ($\{c, e\} = 1 \times 0.611 \times 0.7 = 0.4277$ less than $minsup$) Hence, we do not include it in the $\{e\}$-projected tree. From this $\{e\}$-projected tree we can get: (1) $\{a, e\} = (1 \times 0.611 \times 0.8) + (2 \times 0.61 \times 0.8) =$

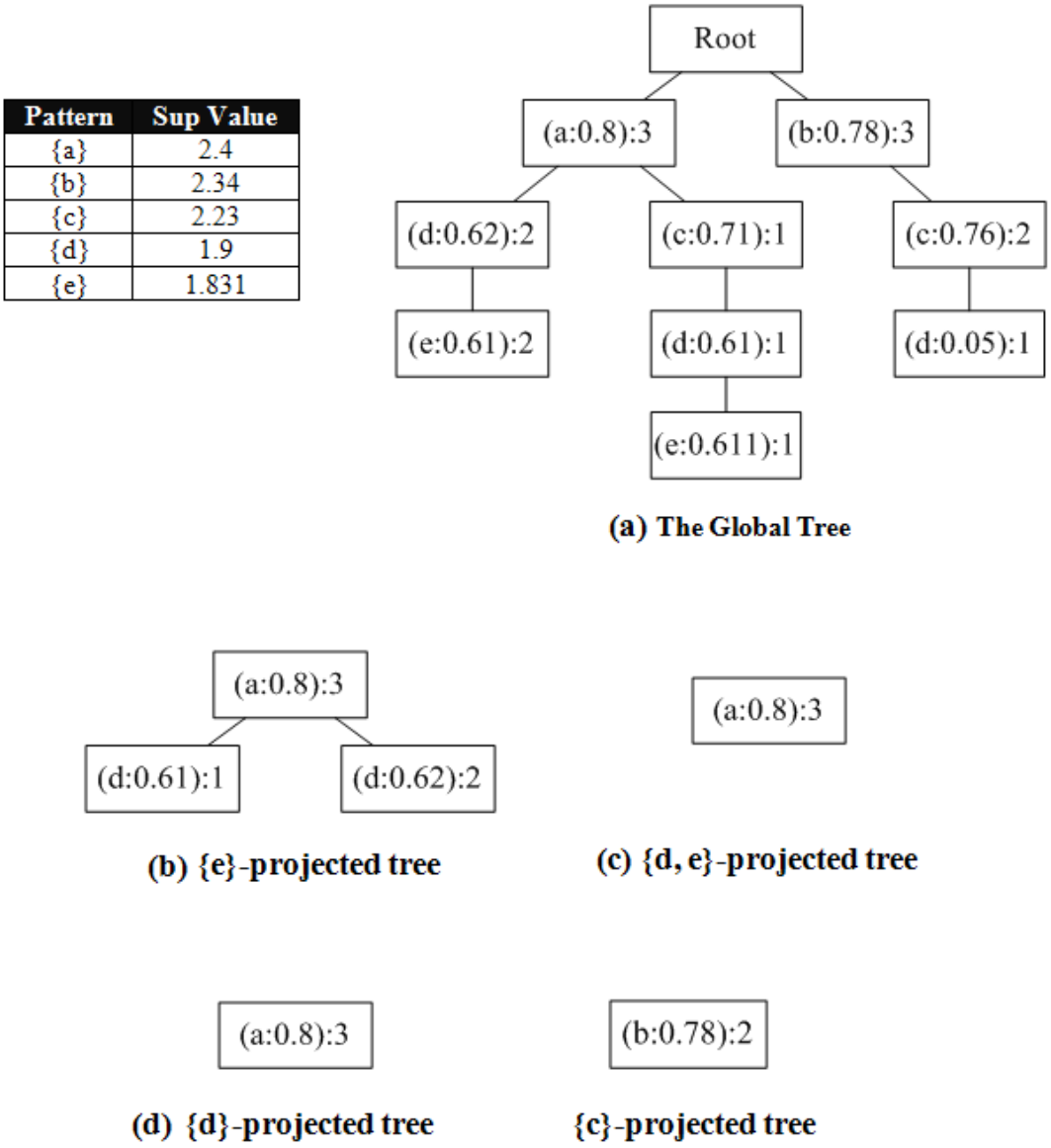


Figure 2.3: The UF-growth Mining (Example 2.3).

1.4648, (2) $\{d, e\} = (1{\times}0.611{\times}0.61){+}(2{\times}0.61{\times}0.62) = 1.12911$. Continuing with the recursive process, we build the $\{d, e\}$-projected tree as shown in Figure 2.3(c). From the $\{d, e\}$-projected tree, we can get: $\{a, d, e\} = (3{\times}0.61{\times}0.62{\times}0.8) = 0.90768$ which

is less than *minsup* 1.0. With the same approach, we form the projected databases and projected trees for $\{d\}$ and $\{c\}$ (as shown in Figure 2.3(d)). Finally, we find all frequent patterns as follows: $\{a\}$*:2.4*, $\{b\}$*:2.34*, $\{c\}$*:2.23*, $\{d\}$*:1.9*, $\{e\}$*:1.831* $\{a,$ $e\}$*:1.4648*, $\{d,$ $e\}$*:1.12911*, $\{a,$ $d\}$*:1.48*, $\{b,$ $c\}$*:1.1856*. ■

2.3 Stream Data Mining Algorithm

In real-life applications new data are flowing in dynamically, and the size of the data keeps increasing. Mihaila et al. [MSL08] defined this type of flowing-in data as *data streams*. As discussed in Section 1, mining frequent patterns from dynamic data streams [GBK10, JG06] is more challenging than mining from traditional static transaction databases due to two characteristics of data streams: (1) Data streams are continuous and unbounded, and (2) data in the streams are not necessarily uniformly distributed. Giannella et al. [GHP+04] proposed *FP-streaming* which is a data stream mining algorithm for *precise data*. Leung and Hao [LH09] designed *UF-streaming*, which is a tree-based data stream mining algorithm for *uncertain data* (by applying the sliding window model). Although these two algorithms handle different types of data, they shared some similarities. They both have two phases for the entire mining process: (1) applying a tree-based mining algorithm for mining current stream of data (current data batch), and (2) storing the mining result (of the current data batch) into a different data structure for future process while a new data batch is flowing in. In the first phases, they both use the idea of *preMinsup*, which is less than the minimum support threshold *minsup* (i.e., *minsup* > *preMinsup*). The *preMinsup* is used in an attempt to avoid pruning a pattern too early because data in the continuous streams

Table 2.4: Data Stream Example for *preMinsup*

Batches	**Transactions**	**Contents**
First	t_1	{a}
	t_2	{b}
	t_3	{a}
Second	t_4	{b}

are not necessarily uniformly distributed. To explain this, let us consider an example (for precise data) as in Table 2.4. If we set *minsup* as 2, the mining process of the first batch returns only $\{a\}$ with a support value 2. Pattern $\{b\}$ (with support value 1) is not considered frequent. The information of $\{b\}$ is discarded when the next batch flows in. Then, FP-streaming processes the next batch. There is only one $\{b\}$ that exists. The support value of $\{b\}$ is 1, less than the *minsup* (which is 2). We have to also discard pattern $\{b\}$ since it is not frequent in this batch. If the algorithm stops here, the final mining result contains only one frequent pattern—namely, $\{a\}$. However, both $\{a\}$ and $\{b\}$ are truly frequent. Hence, we need to use a relatively smaller value as support threshold for each data stream (batch). In each data stream (batch), patterns with support value $\geq$ *preMinsup* are recorded. However, in the final result, only patterns with support value $\geq$ *minsup* are returned as frequent patterns (i.e., frequent pattern support $\geq$ *minsup* $>$ *preMinsup*).

Let us discuss two data stream mining algorithms, FP-streaming [GHP$^+$04] and UF-streaming [LH09], in more details in the following sections.

2.3.1 FP-streaming Algorithm

FP-streaming was proposed by Giannella et al. [GHP+04] in 2004. It is a tree-based *precise data* stream mining algorithm. As described before, the first step of FP-streaming is to call the FP-growth algorithm with a $preMinsup$ to mine current data batch (stream). Once the FP-growth algorithm finds all "frequent" patterns, the next step of FP-streaming algorithm is to store and maintain these patterns in another tree structure called *FP-stream*. In the FP-stream, each path represents a "frequent" pattern (different from the FP-tree where each path represents a transaction). Each node in the FP-stream contains a sliding window table which contains multiple support values, one for each batch of transactions. The definition and properties of the "sliding window" will be provide later in Section 2.4.1.

Note that, FP-streaming is a data stream mining algorithm for *precise data*. The problem gets more challenge when we want to process *uncertain data* streams.

2.3.2 UF-streaming Algorithm

UF-streaming [LH09], proposed by Leung and Hao, mines frequent patterns from *uncertain data* streams by using a fixed-size sliding window of w recent batches. UF-streaming first calls UF-growth (see Section 2.2.2) to find "frequent" patterns from the current batch of transactions in the streams (using *preMinsup* as the threshold). A pattern is *"frequent"* (i.e., *subfrequent*) if its expected support $\geq$ *preMinsup*.

UF-streaming then stores the mined "frequent" patterns and their expected support values in a tree structure, in which each tree node X keeps a list of w support values. When a new batch flows in, the window slides and support values shift so that

Table 2.5: Uncertain Data Stream Example for UF-streaming

Batches	**Transactions**	**Contents**
First	t_1	{a:0.3, b:0.6, d:0.4, e:0.5}
	t_2	{a:0.8, d:0.8, e:0.9, f:0.2}
	t_3	{a:0.8, b:0.6}
Second	t_4	{a:0.9}
	t_5	{a:0.7, b:0.2}
	t_6	{b:0.2, c:0.4}
Third	t_7	{a:0.8, d:0.62, e:0.61}
	t_8	{a:0.8, c:0.71, d:0.61, e:0.611}
	t_9	{b:0.78, c:0.76}
	t_{10}	{a:0.8, d:0.62, e:0.61}
	t_{11}	{b:0.78, c:0.76, d:0.05}
	t_{12}	{b:0.78}

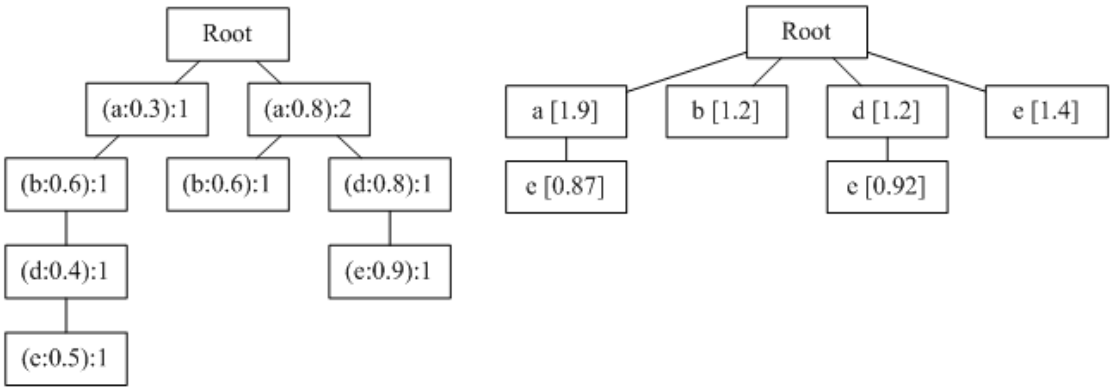
the "frequent" patterns (and their expected support values) mined from the newest batch are inserted into the window and those representing the oldest batch in the window are deleted. This process is repeated for each batch in the stream. The expected support of any frequent pattern X can be computed by summing all w expected supports of X (one for each batch in the sliding window). Let $expSup(X, B_i)$ denote the expected support of X in Batch B_i. Then, at time T, the expected support of X in the current sliding window containing w batches of uncertain data in Batches B_{T-w+1}, ..., B_T inclusive can be computed as follows:

$$expSup(X, \bigcup_{i=T-w+1}^{T} B_i) = \sum_{i=T-w+1}^{T} expSup(X, B_i). \tag{2.4}$$

To get a better understanding of the UF-streaming algorithm, let us take a look at the following example.

Example 2.4 Let us consider the transaction database in Table 2.5. Let us set the window size w to 2, the minimum support threshold *minsup* to 1.0, and the

preMinsup to 0.5. First, the algorithm calls UF-growth (Section 2.2.2), constructs the UF-tree as in Figure 2.4(a). After that, all frequent patterns that are returned by UF-growth are stored into the UF-stream. The UF-stream for the first data batch (stream) is shown in Figure 2.4(b). Then, the second data batch flows in, the algorithm frees the memory of the UF-tree, applies UF-growth again to construct a new UF-tree for the second batch (while the UF-stream stay the same). After UF-growth processed the second batch, the mining result is stored into the UF-stream (Figure 2.5(d)). Note that, now, the sliding window in the UF-stream contains two numbers. When the third data batch flows in, the same process repeats. However, when the UF-stream stores the mining result of the third data batch, the sliding window in each node of the UF-stream is full. Hence, as shown in Figure 2.6(f), the sliding window discards the oldest data for storing the new data. ■



(a) UF-tree for the 1st batch **(b)** UF-stream for the 1st batch

Figure 2.4: The UF-streaming Mining (The 1st Data Batch of Example 2.4).

By using the sliding window model, the UF-streaming algorithm is able to handle many streams of data. However, the sliding window model discards information when

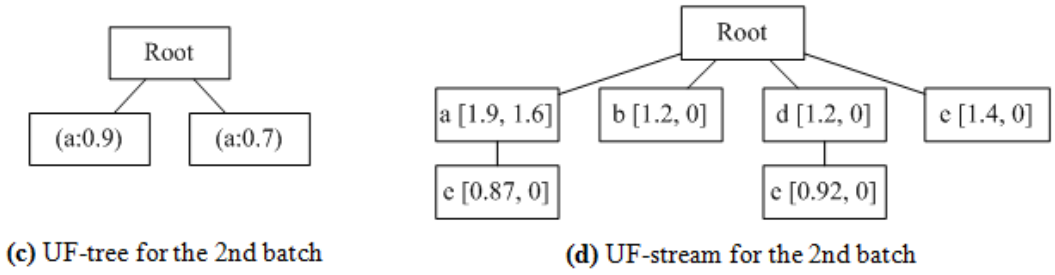


Figure 2.5: The UF-streaming Mining (The 2nd Data Batch of Example 2.4).

the window is full, which means the mining result we can get from the sliding window model only covers the recent w data batches. Is there a better solution to the problem of data stream mining if we do not want to discard data? In the next section, we will discuss different windowing techniques.

2.4 Different Windowing Techniques

In this section, we discuss three windowing models: (1) the sliding window model, (2) the landmark window model, and (3) the time-fading window model (also known as the damped window model). In the remainder of this chapter, we use the term *data segment* to represent a batch of incoming data. Each data segment is associated with a value. For instance, one may consider such a value as the support of a pattern in that batch.

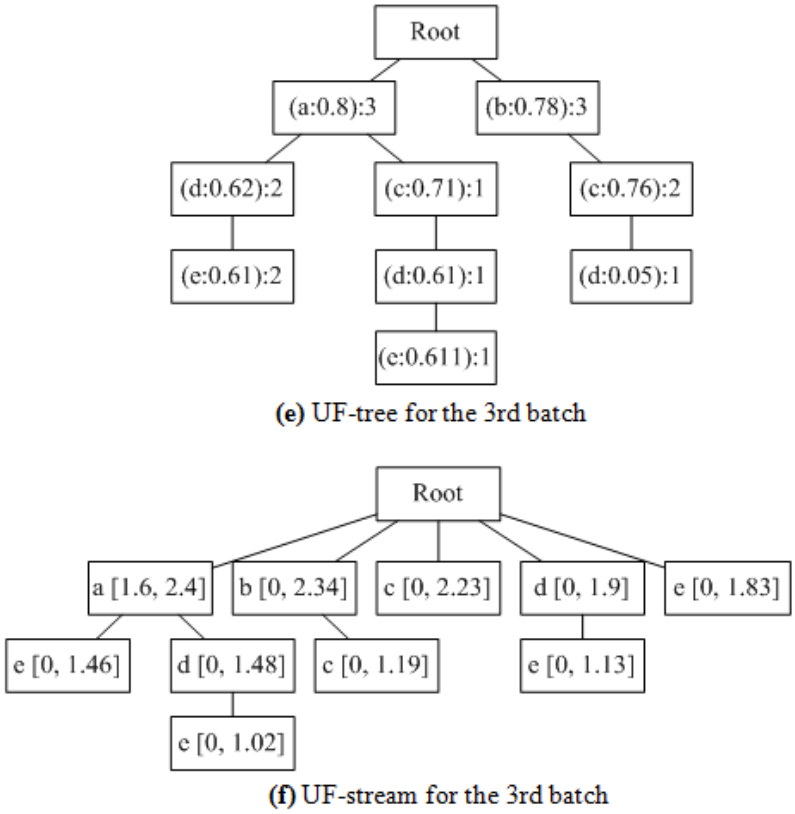


Figure 2.6: The UF-streaming Mining (The 3rd Data Batch of Example 2.4).

2.4.1 The Sliding Window Model

The sliding window model requires user define a size of the window, and the window *slides* over time periods. As show in Figure 2.7, the arrow indicates the "window". Initially, window W (with fixed-size of 3) is covering the first data segments. When the next time period comes, the sliding window grows to size 2. Then, in the third time period, the sliding window grows to size 3, which is the maximum capability of the sliding window in the example. In the next time period, the window *slides* one segment to the right side. Now, it is covering data segment 2 to data segment 4. It

discards the data segment in the right end (oldest data segment) in order to maintain the fixed window size. The process repeats itself, and the window "slides" as shown in Figure 2.7.

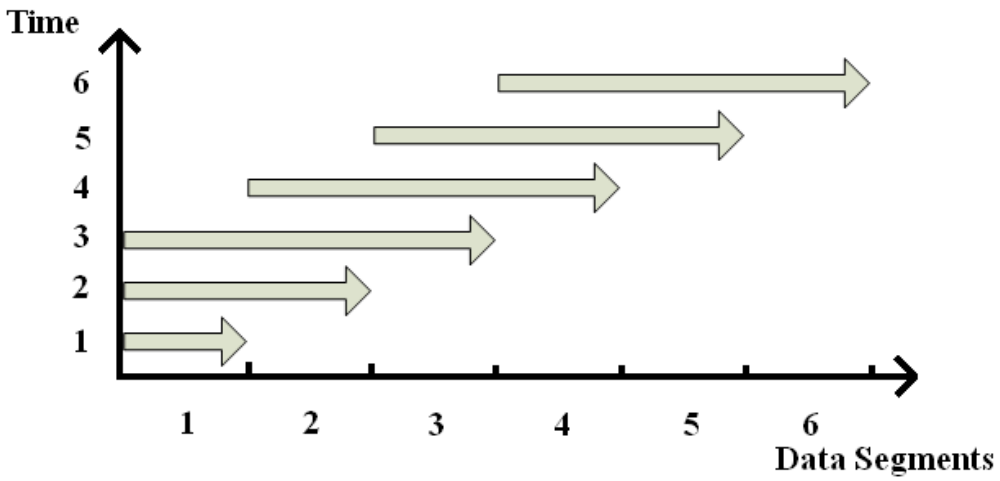


Figure 2.7: The Sliding Window Example.

2.4.2 The Landmark Window Model

The landmark window model is not a fixed-size window technique. It starts from a given time point and ends at the current time, and keeps increasing the size of the window. Note that the starting point is fixed, but the end point keeps moving as time passes. As show in Figure 2.8, the landmark window model starts from the left-most data segment. The window consists of data segment 1. In the next time period, the end-point (left-most point) of the landmark window moves to the next data segment, but the start-point (right-most point) stays the same. As time passes, the size of the landmark window keeps increasing, and stores every data segment.

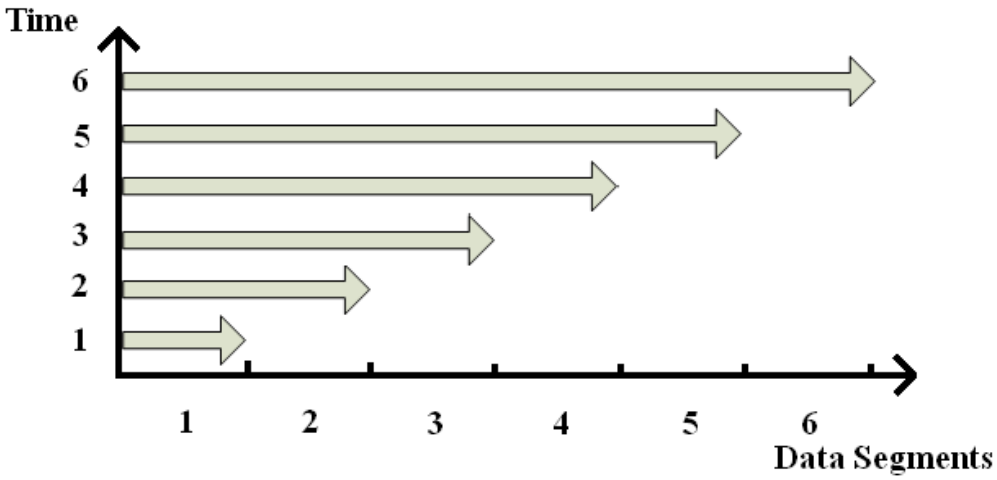


Figure 2.8: The Landmark Window Example.

2.4.3 The Time-Fading Window Model

The time-fading window model considers the current data more valuable than old data. It assigns a weight value (denoted as α) for each data segment, and the weight value decreases over time. In other words, as time passes, old data make less contribution to the whole data set. Figure 2.9 shows how the time-fading window model process data. The time-fading window starts from the right side, when it moves to the next data segment, the old data become less important. However, the start point of the time-fading window always stays at the left-most side.

We can observe that, unlike the sliding window model, the time-fading window model does not discard data segment as time passes. Also, unlike the landmark window model, it does not consider everything as the same importance. It "fades out" old information so that users could focus more on the current data.

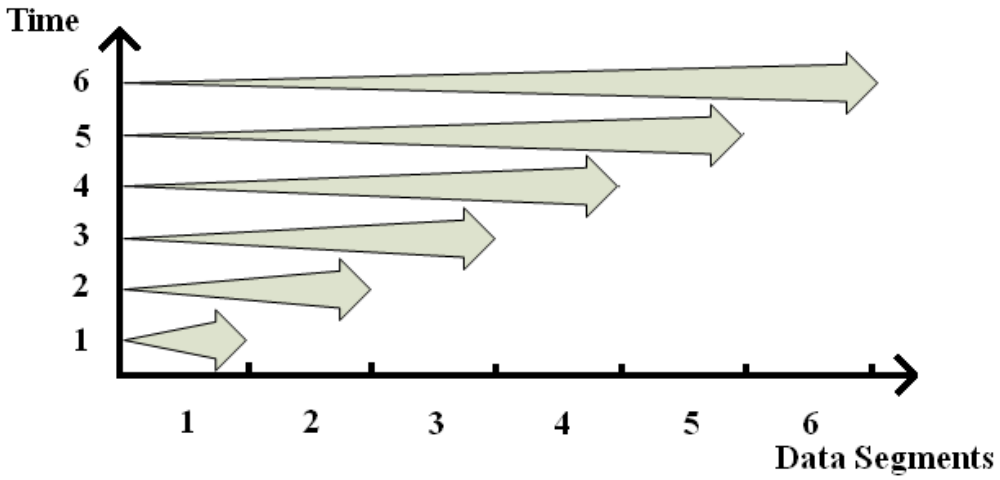


Figure 2.9: The Time-Fading Window Example.

Table 2.6: Data Segments Example for Windowing Techniques

Data segment	Value
1st	10
2nd	8
3rd	12
4th	6
5th	20
6th	2

2.4.4 Comparison of Three Windowing Techniques

To illustrate the difference between the above three windowing techniques, let us consider the following example in Table 2.6. The first column of Table 2.6 represents the sequence of each data segment. The second column represents the value (e.g., support value) of each data segment. For example, the first row "1st | 10" represents the data segment with value 10 is the first data segment that we received. If we consider these sequence of data segments as data streams, each of the three window

model processes these data streams differently as follows.

Example 2.5 Suppose the window size is 3. For the sliding window model, after the last data segment arrives, the value that the sliding window model returned is $6 + 20 + 2 = 28$ (i.e., the sliding window model captures the value for the last 3 data segments—from the 4th to 6th data segments). ■

Example 2.6 Suppose data segment 1 is the "landmark". For the landmark window model, after the last data segment arrives, the value that the sliding window model returned is $10 + 8 + 12 + 6 + 20 + 2 = 58$ (i.e., the landmark window model captures the value for all 6 data segments since the "landmark" segment). ■

Example 2.7 Suppose α value (weight) is 0.5. For the time-fading window model, after the last data segment arrives, the value that the time-fading window model return is $10 \times 0.5^5 + 8 \times 0.5^4 + 12 \times 0.5^3 + 6 \times 0.5^2 + 20 \times 0.5^1 + 2 \times 0.5^0 = 15.8125$ (i.e., the time-fading window model captures the weighted value for all 6 data segments). ■

2.5 Summary

In this chapter, we reviewed the following related works: (1) precise data mining algorithms (e.g., Apriori algorithm [AS94] and FP-growth [HPY00]), (2) uncertain data mining algorithms (e.g., U-Apriori [CKH07] and UF-growth [LMB08]), and (3) stream data mining algorithms (e.g., FP-streaming [GHP+04] and UF-streaming [LH09]).

To mine frequent pattern from traditional databases of precise data, the Apriori algorithm [AS94], which uses a bottom-up generate-and-test framework. The bottleneck of this approach is the candidate generation process. To improve performance, Han et al. [HPY00] designed FP-growth, which avoids the candidate generation pro-

cess by applying a divide-and-conquer approach on a data structure called an FP-tree. Both the Apriori algorithm and FP-growth only handles precise data.

To mine frequent pattern from uncertain data, U-Apriori [CKH07] was proposed. As it inherits the Apriori-based approach from the Apriori Algorithm, U-Apriori also inherits the bottleneck of Apriori. To avoid the candidate generation process, UF-growth [LMB08] was designed. It inherits the divide-and-conquer approach from FP-growth. The data structure (the UF-tree) is redesigned for storing and processing uncertain data.

Unlike traditional static databases, most databases have new data flowing in dynamically in real-life situations. To mine frequent patterns from dynamic databases, stream mining is needed. Leung and Hao proposed UF-streaming [LH09] for mining frequent patterns from dynamic streams of uncertain data. For each data stream, UF-streaming applies UF-growth to process the data and stores the mining results in a data structure called a UF-stream. The UF-stream uses the sliding window model approach, which discards old data if the window is full.

Besides the sliding window model, there are other windowing techniques such as the landmark window model and the time-fading window model. Note that the sliding window model considers only some recent data. It discards old data when the "window" is full. The landmark window model considers all data are of the same importance. It keeps increasing the size of the window and stores everything from the "landmark". The time-fading window model, on the other hand, puts more emphasis on recent data than historical data by putting heavier weights to recent data than historical ones.

Chapter 3

Mining Frequent Patterns from Uncertain Data Stream

In this chapter, we start describing our research works and contributions. Recall from Section 2.4.1 that some existing works (i.e., UF-streaming [LH09]) uses the sliding window model to handle dynamic data streams, where the sliding window model discards old data if the window is full. However, in most real-life applications, users usually do not want to (or should not) completely ignore old data (or data streams). In this chapter, we describe how to apply different windowing techniques into our researches, what are the challenges, and how do we solve them?

Here, we propose algorithms for mining frequent patterns from streams of uncertain data. We first propose an algorithm called *LUF-streaming* (LUF stands for *L*andmark and *U*ncertain *F*requent Patterns), which uses the landmark window model. We also propose a family of algorithms, called *TUF-streaming* (TUF stands for *T*ime-fading and *U*ncertain *F*requent patterns), which uses the time-fading win-

dow model.

3.1 Frequent Patten Mining with the Landmark Window Model

Recall from Section 2.4.2, the landmark window model is not a fixed-size window technique. It records all data segments starting from a given time point and ending at the current time. Since it considers every data of the same importance, no data are discarded. As such, the size of the window keeps increasing. Our proposed algorithm works as follows.

Given (1) a stream of uncertain data and (2) a user-specified minimum support threshold (minsup), our proposed LUF-streaming algorithm uses the landmark model in an uncertain data environment to mine frequent patterns (i.e., patterns having expected support $\geq$ minimum support threshold *minsup*) from streaming data. This algorithm consists of the following three key modules: (1) the batch mining module, (2) the stream mining module, and (3) the pruning module.

3.1.1 The Batch Mining Module

This module mines "frequent" patterns from each batch of the stream by calling UF-growth. Recall from Section 2.3 that data in the stream are not necessarily uniformly distributed. A currently infrequent pattern may become frequent later. Hence, instead of applying UF-growth with the minimum support threshold (*minsup*), this module attempts to avoid discarding a (currently infrequent but potentially frequent)

pattern too early by applying UF-growth with a lower $preMinsup$ threshold (where $preMinsup < minsup$). As such, the module finds every subfrequent pattern X (i.e., X with expected support $\geq preMinsup$) from each batch of uncertain data in the stream. The expected support value of X in Batch B_j can be computed by summing (over all transactions within B_j) the product (of existential probabilities of independent items within pattern X) using the following equation [Leu11]:

$$expSup(X, B_j) = \sum_{i=1, t_i \in B_j}^{w} \left(\prod_{x \in X} P(x, t_i) \right). \tag{3.1}$$

where w is the number of batches in the stream.

3.1.2 The Stream Mining Module

This module maintains the subfrequent patterns mined from each batch of the stream in a tree structure called LUF-stream such that each tree node represents a pattern and contains information about its expected support in the stream (from the landmark to the present time). For example, the leftmost branch of the LUF-stream shown in Figure 3.1(b) represents $\{a\}$ and $\{a, e\}$ with expected supports of 2.2 and 0.87, respectively.

Recall that the number of batches increases and thus the window size for the landmark model keeps growing as time progresses, and older data are of the same importance as new data. Hence, it is impractical for each tree node representing X to keep a list of $expSup(X, B_j)$ for every batch.

Instead, the stream mining module keeps the sum of $expSup(X, B_j)$ in each node representing X. One way to maintain this sum is to visit/update every tree node in

the LUF-stream whenever the batch mining module finds some subfrequent patterns. This would lead to an O($b \mid \bigcup_j fp_j \mid$) runtime, where b is the number of batches, $\mid fp_j \mid$ is the number of subfrequent patterns mined from Batch B_j , and thus $\mid \bigcup_j fp_j \mid$ is the total number of patterns kept in the tree structure.

Example 3.1 Consider the transaction database in Table 3.1, Let us set the $preMinsup = 0.5$. After processing batch B_1, $|fp_1| = 5$ subfrequent patterns are mined. Then, the LUF-stream stores $|fp_1| = 5$ nodes. Then, if $|fp_2| = 3$ subfrequent patterns are mined from batch B_2 and one of them already exists in the LUF-stream, then stream mining module needs to update $|\bigcup_{j=1}^{2} fp_j| = 5 + (3 - 1) = 7$ nodes. The total number of updates for these b=2 batches is about $b|\bigcup_j fp_j| = 5$ (for B_1) + 7 (for B_2) = $12 \leq 2 \times |\bigcup_{j=1}^{2} fp_j| = 2 \times 7 = 14$. We can observe that this approach is not too efficient. The cost of updating the entire LUF-stream can be extremely expensive if the size of the LUF-stream gets huge. The number of incoming data batches is potentially unbounded. ■

To improve the performance, instead of visiting/updating every LUF-stream node, the stream mining module only visits/updates the nodes representing those patterns that are subfrequent in B_j . This reduces the runtime complexity to O($\sum_j \mid fp_j \mid$). The summation here ($\sum$) is different from the union ($\bigcup$) that we used previously. The summation computes the sum of the number of frequent patterns in each data batches, while the union computes the total number of frequent patterns in the LUF-stream.

Example 3.2 Let us revisit the above example. After processing batch B_1, $|fp_1| =$ 5 subfrequent patterns are mined. Then, the LUF-stream stores $|fp_1| = 5$ nodes.

Then, if $|fp_2| = 3$ subfrequent patterns are mined from batch B_2 and one of them already exist in the LUF-stream, then stream mining module needs to update only these three newly mined subfrequent patterns (from B_2). The total number of updates for these b=2 batches is $\sum_{j=1}^{2} |fp_j| = 5$ (for B_1) + 3 (for B_2) = 8. ■

3.1.3 The Pruning Module

Finally, the pruning module prunes away those subfrequent patterns that are not truly frequent at the time when the user requests for the final mining results. Note that, in an attempt to avoid pruning patterns too early, the batch mining module finds subfrequent patterns with a *preMinsup* (i.e., patterns with expected support value $\geq$ *preMinsup* but may be $<$ *minsup*). These subsequent patterns are kept in the LUF-stream, and their expected support values are updated by the stream mining module. At the time when the user requests for truly frequent patterns, the pruning module discards those patterns with expected support value greater than or equals to *preMinsup* but less than *minsup*. See the following example:

Example 3.3 Let us consider the transaction database in Table 3.1. Let us set the minimum support threshold to 1.0, and the $preMinsup$ to 0.5. The LUF-streaming algorithm first calls UF-growth to find frequent pattern from the first data stream. The UF-growth constructs a UF-tree as shown in Figure 3.1(a). Then, the mining results from the UF-tree are stored into the tree data structure the LUF-stream (Figure 3.1(b)). After that, the second data stream flows in. The algorithm releases the memory of the previous UF-tree, calls UF-growth again to construct a new UF-tree for the second data stream (Figure 3.1(c)). After the mining process, the

Table 3.1: Uncertain Data Stream Transaction Database

Batches	**Transactions**	**Contents**
First	t_1	{a:0.3, b:0.6, d:0.4, e:0.5}
	t_2	{a:0.8, d:0.8, e:0.9, f:0.2}
	t_3	{a:0.8, b:0.6}
	t_4	{a:0.3, c:0.9}
Second	t_5	{a:0.9}
	t_6	{a:0.9, b:0.1, c:0.1}
	t_7	{b:0.1, c:0.2}
Third	t_8	{a:0.8, d:0.62, e:0.61}
	t_9	{a:0.8, c:0.71, d:0.61, e:0.611}
	t_{10}	{b:0.78, c:0.76}
	t_{11}	{a:0.8, d:0.62, e:0.61}
	t_{12}	{b:0.78, c:0.76, d:0.05}
	t_{13}	{b:0.78}

LUF-stream receives a group of new frequent patterns. Instead of keeping a growing "landmark window", the algorithm processes the support value for each frequent pattern immediately. In Figure 3.1(d), since only the pattern $\{a\}$ is frequent, the algorithm only needs to visit the tree node which represents $\{a\}$, and update the support value in the tree node (from 2.2 to 4.0 in the example). Then, the algorithm repeats this process until all data streams are processed. The final LUF-stream data structure is shown in Figure 3.1(f). ■

3.2 Frequent Patten Mining with the Time-Fading Window Model

The landmark window model considers all data segments the same importance as each other. However, in some real-life applications, users want to focus more on

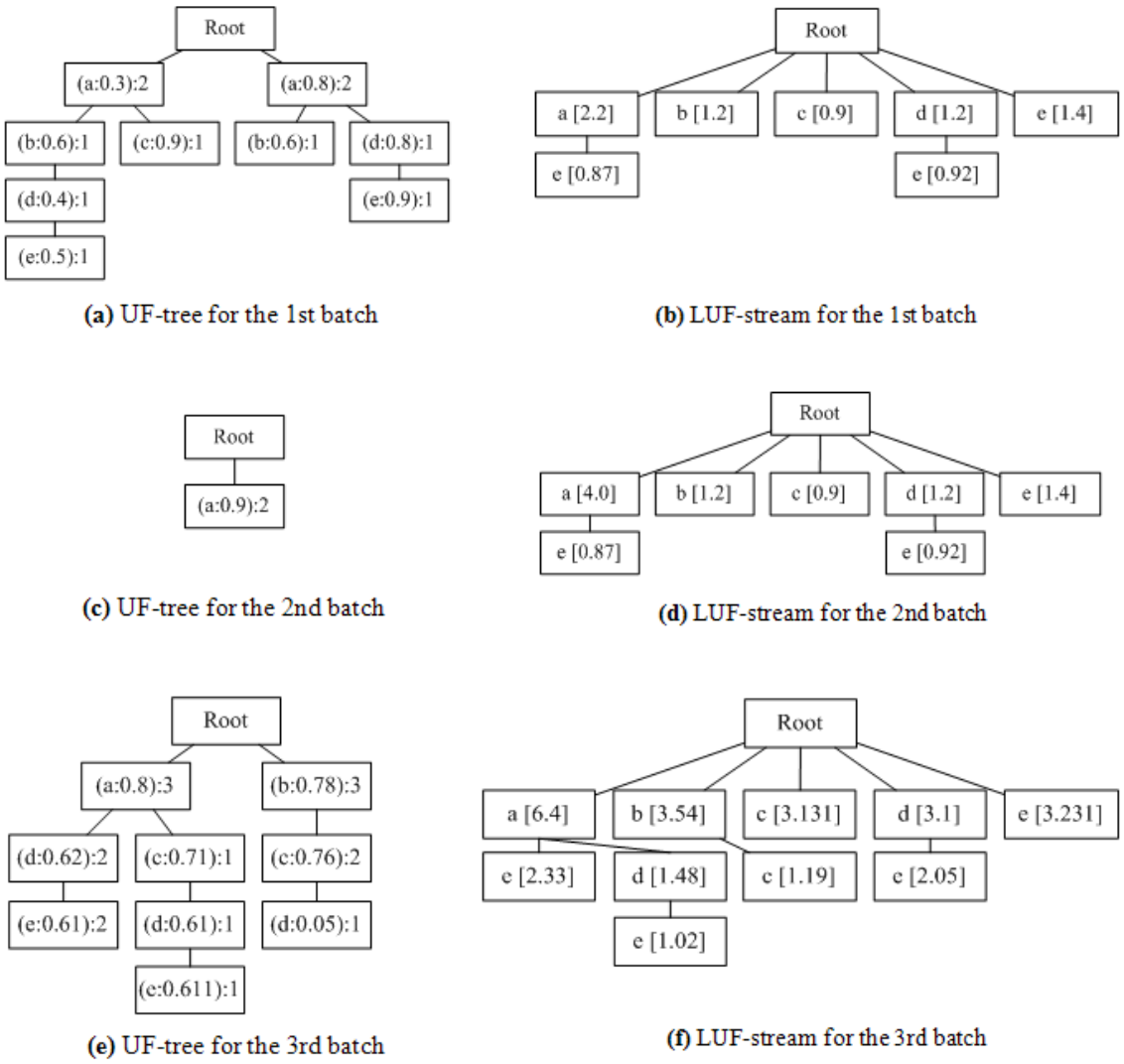


Figure 3.1: The LUF-streaming Mining Example.

the recent data (but also do not want to completely ignore the effect from old data). For example, in supermarket sales database, managers want to focus more on what products are popular now. At the same time, they do not want to totally discard the old sales data. To fulfill that perspective, the time-fading window model is a correct

approach. Here, we propose our *TUF-streaming* algorithm family (consists of three algorithms), which uses the time-fading window model.

Recall from Section 2.4.3, the time-fading window model considers the current data more valuable than old data by assigning a weight value for each data segment. The weights decrease over time to represent less importance of a data segment. In this section, we propose our TUF-streaming algorithm family, which applies the time-fading model in an uncertain data environment to mine frequent patterns from streaming data. We developed the following three algorithms in the TUF-streaming family,

1. A Naïve Algorithm called TUF-Streaming(Naïve).
2. A Space-Saving Algorithm called TUF-Streaming(Space), which reduces the memory usage.
3. A Time-Saving Algorithm called TUF-Streaming(Time), which reduces the running time.

3.2.1 A Naïve Algorithm: TUF-Streaming(Naïve)

The first algorithm of the TUF-streaming family is *TUF-streaming(Naïve)*. The key steps of the algorithm can be described as follows. First, for each batch B_i of uncertain data in the stream, our algorithm applies UF-growth (Section 2.2.2) with $preMinsup$ to find "frequent" patterns (i.e., patterns with expected support value $\geq preMinsup$ from a batch). Then, it stores the mined "frequent" patterns and their expected support values in a tree structure called TUF-stream, in which each tree

node corresponding to a pattern X keeps a list of support values. Note that the time-fading window does not slide. Instead, it grows.

This mining process and the TUF-stream insertion process are repeated for each batch in the stream of uncertain data. Let $expSup(X, B_i)$ denote the expected support of X in B_i. Then, at time T, the expected support of X mined from the time-fading model can be computed by summing over all batches the expected supports of X (weighted by the time-fading factor α, where $0 < \alpha \leq 1$):

$$expSup(X, \cup_{i=1}^{T} B_i) = \sum_{i=1}^{T} \left(expSup(X, B_i) \times \alpha^{T-i} \right), \tag{3.2}$$

Example 3.4 Let us take the same example (Example 3.3) as we used for LUF-streaming. Consider the stream of uncertain data in Table 3.1. Here, each transaction contains items and their corresponding existential probabilities, e.g., $P(a, t_1) = 0.7$. Let the user-specified *minsup* threshold be 1.0. When using the time-fading model, TUF-streaming(Naïve) applies UF-growth to B_1 in the uncertain data stream using $preMinsup < minsup$ (say, $preMinsup = 0.5$) and finds "frequent" patterns $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{a, e\}$, and $\{d, e\}$ with their corresponding expected support of $2.2, 1.2, 0.9, 1.2, 1.4, 0.87$ and 0.92 (shown in Figure 3.2(a)). These patterns and their expected support values are then stored in the TUF-stream structure as shown in Figure 3.2(b). Each node in the TUF-stream keeps an item and a list (window) of expected support values. So far (after processing B_1), each list is of length 1 (e.g., $e : [0.92]$ on the branch $\langle b[1.2],\ e[0.92] \rangle$ represents "frequent" pattern $\{b, e\}$ with an expected support of 0.92).

Next, when the second batch B_2 arrives, TUF-streaming(Naïve) applies a similar

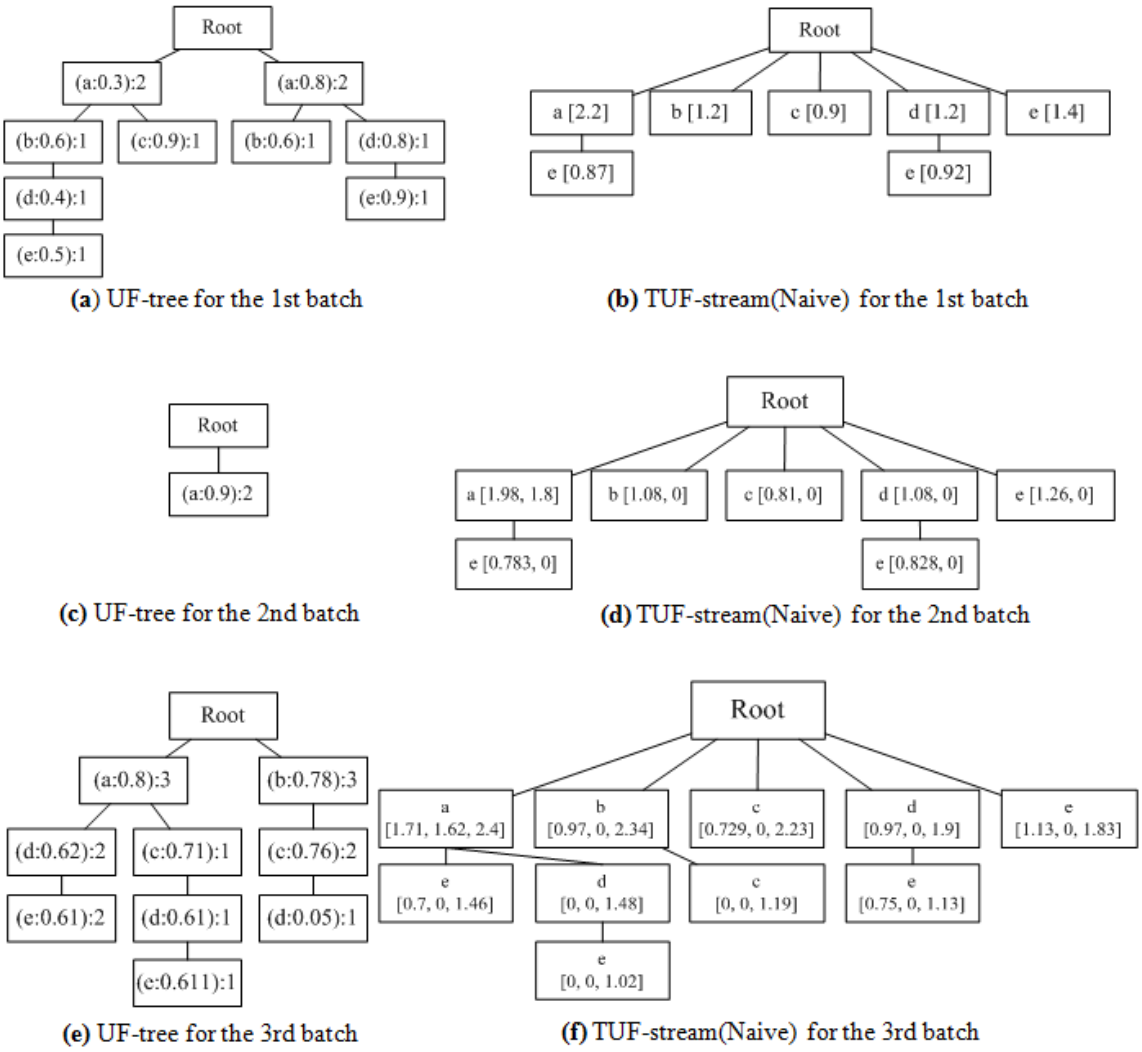


Figure 3.2: The TUF-streaming(Naïve) Mining Example.

procedure: Call UF-growth to find "frequent" patterns. In this data batch, only $\{a\}$ with support value 1.8 is frequent (shown in Figure 3.2(c)). Then, the algorithm appends each expected support value to the list of the appropriate tree node in the TUF-stream. The resulting TUF-stream, as shown in Figure 3.2(d), consists of eight

nodes (including the root node). Note that the list in each node now consists of two expected support values. The expected support of any pattern X can be computed using Equation (3.2) based on the expected support values stored in the list of X. For instance, let the time-fading factor α be 0.9, then $expSup(\{a\}, B_1 \cup B_2) = 2.2\alpha + 1.8 = 1.98 + 1.8$.

Similarly, when subsequent batches arrive, TUF-streaming(Naïve) applies a similar procedure. Figure 3.2(f) shows the resulting TUF-stream structure after processing B_3. At that time, take pattern $\{a\}$ as example again, the expected support value $expSup(\{a\}, B_1 \cup B_2 \cup B_3) = 2.2\alpha^2 + 1.8\alpha + 2.4 = 1.71 + 1.62 + 2.4$. ■

Recall that in Section 3.1, we described the updating process that LUF-streaming uses (instead of visiting/updating every tree node, the stream mining module only visits/updates the nodes representing those patterns that are subfrequent in B_j). This updating process cannot be applied directly here in the TUF-streaming algorithm family. The reason for this is that the landmark window model does not require a change in the α value for each data stream. In other word, in the landmark window model, the value of α is always 1. Based on that, in the landmark window model, the Equation (3.2) can be simplified to become the following:

$$expSup\left(X, \cup_{i=1}^{T} B_i\right) = \sum_{i=1}^{T} (expSup(X, B_i) \times 1^{T-i}) = \sum_{i=1}^{T} expSup(X, B_i). \qquad (3.3)$$

Therefore, for each tree node that is not frequent in the current data stream. We do not need to update the value of α^{T-i}. (It is always 1.) Based on this unique characteristic of the landmark window model, we were able increase the performance by only visiting/updating the tree node when it is frequent in the current data stream.

In the time-fading window model, we cannot apply this directly because α can be < 1. Instead, we come up with two different algorithms to increase performance in both memory usage and execution efficiency.

3.2.2 A Space-Saving Algorithm: TUF-Streaming(Space)

Although TUF-streaming(Naïve) finds all "frequent" patterns, it may require a large amount of space. As data streams are continuous and unbounded, storing the expected support value of X for each batch in the streams can be impractical because it could lead to a potentially infinite list ("window") for each node. A careful analysis on Equation 3.2 reveals that the expected support of X is the sum of weighted expected support of X over all batches. Unlike the sliding window model (which requires the deletion of the oldest batch), the fading-time model does not require the deletion of any old batches. Instead, it assigns lighter weights to old batches than recent batches. As a special case, for the landmark model, the algorithm assigns the same weights to all batches (regardless whether they are old or recent). Hence, we propose a space-saving algorithm called *TUF-streaming(Space)*, which does not need to keep track of the details for each batch. We rewrite Equation (3.2) in a recursive form as follows:

$$expSup\left(X, \bigcup_{i=1}^{T} B_i\right) = \left[expSup\left(X, \bigcup_{i=1}^{T-1} B_i\right) \times \alpha\right] + expSup(X, B_T). \qquad (3.4)$$

By doing so, the algorithm keeps only a single value (i.e., $expSup(X, \cup_{i=1}^{T} B_i)$), instead of a potentially infinite list of $expSup(X, B_i)$.

Example 3.5 Let us consider the same example in Table 3.1, the process of ap-

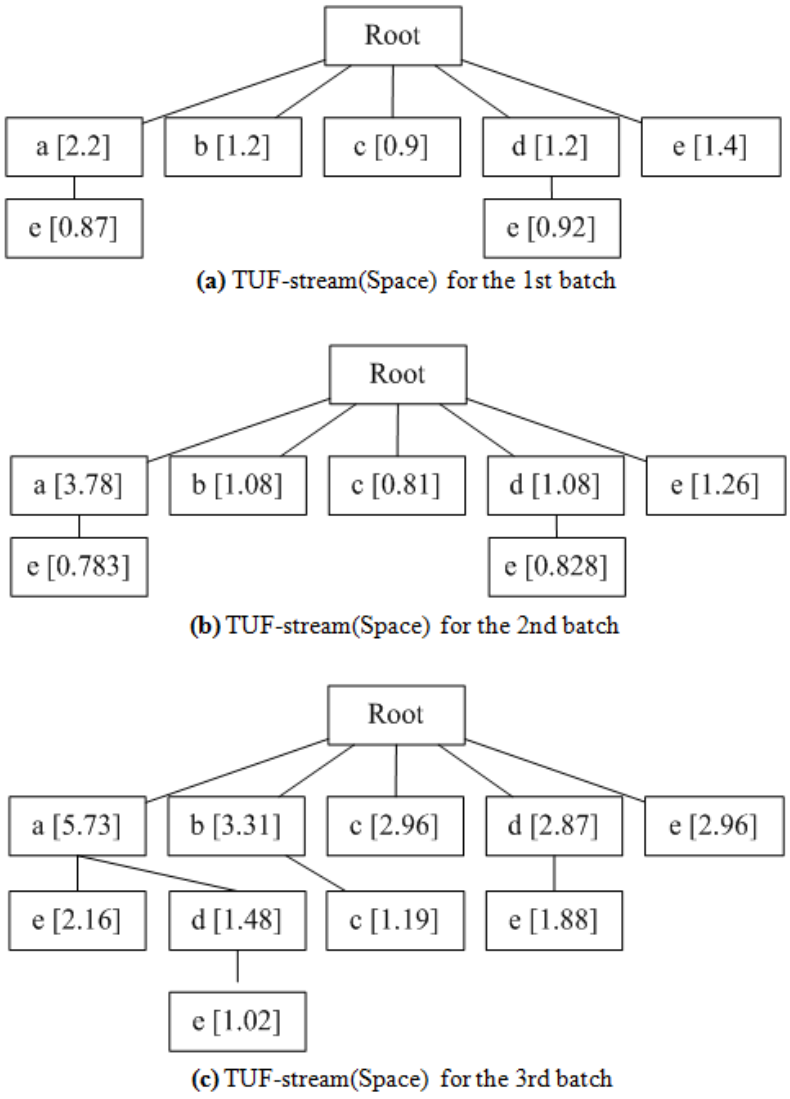


Figure 3.3: The TUF-streaming(Space) Mining Example.

plying TUF-streaming(space) is as follows: Since the UF-growth process is the same, in the Figure 3.3, we can remove the UF-tree structures. TUF-streaming(Space) algorithm uses Equation (3.4) to compute expected support values. For instance, Figure 3.3(a) shows the expected support values stored in the TUF-stream after min-

ing "frequent" patterns from B_1. They are identical to those shown in Figure 3.2(b).

Afterwards (say, after mining B_i for $i \geq 2$), instead of appending the expected support values of "frequent" patterns mined from B_i, the algorithm modified the stored value. For instance, after mining B_2, instead of storing [1.98, 1.8] for {a} as in Figure 3.2(d) for TUF-streaming(Naïve), TUF-streaming(Space) stores their sum $2.2\alpha + 1.8 \approx 3.78$ as $expSup(a, B1 \cup B2)$ in Figure 3.3(b). Similarly, after mining B_3, instead of storing [1.71, 1.62, 2.4] as in Figure 3.2(f) for TUF-streaming(Naïve), TUF-streaming(Space) stores their sum $2.2\alpha^2 + 1.8\alpha + 2.4 = 5.73$ as $expSup(a, B_1 \cup B_2 \cup B_3)$ in Figure 3.3(c). ■

3.2.3 A Time-Saving Algorithm: TUF-Streaming(Time)

TUF-streaming(Space) greatly reduces the amount of space required from a potentially infinite list of expected support values to a much more realistic and practical requirement of storing only a single expected support value in each node in the TUF-stream. However, as observed from the recursive formula shown in Equation (3.4), the expected support of any pattern X up to time T directly depends on the expected support of X up to time $T - 1$. As such, TUF-streaming(Space) needs to visit every node in the TUF-stream after mining each batch (even if the corresponding pattern is not "frequent" in that batch) in order to compute expected support for "frequent" patterns. On the one hand, such a requirement may incur a long runtime. On the other hand, if one were to skip nodes in some batches, then the resulting expected support may not be correct.

Example 3.6 Let us revisit previous transaction database example in Table 3.1.

Figure 3.3(a) shows that $\{c\}$ is "frequent" with $expSup(c, B_1) = 0.9$. We know that $\{c\}$ does not appear in B_2, but it is "frequent" is B_3 with $expSup(c, B_3) = 2.23$. If after mining B_2, we decide to skip the node corresponding to $\{c\}$, then the expected support value stored in the TUF-stream would remain unchanged (i.e., at 0.9). Then, after mining B_3, we decide to visit and update the node for $\{c\}$, then the expected support value stored in the TUF-stream would become $0.9\alpha + 2.23 = 3.04$ while the correct expected support should be $0.9\alpha^2 + 0\alpha + 2.23 = 2.96$. The problem was caused by skipping this node after mining B_2 (even though $\{c\}$ is not "frequent"). The skip led to the missing multiplication of α. ■

To solve the above problem, we propose a time-saving algorithm called *TUF-streaming(Time)*, which does not need to visit every node in the TUF-stream after mining each batch. With this algorithm, the number of nodes visited at each batch is *proportional* to the number of "frequent" patterns mined from that batch. In other words, it visits only those nodes representing patterns that are "frequent" in that batch. It is possible due to our analytical results, which reveal that Equation (3.4) can be written as Equation (3.5), where LV is an additional field stored in each node of the TUF-stream to indicate the batch number of last visit of the node.

$$expSup\left(X, \bigcup_{i=1}^{T} B_i\right) = \left[expSup(X, \bigcup_{i=1}^{LV} B_i) \times \alpha^{T-LV}\right] + expSup(X, B_T), \qquad (3.5)$$

See the following example.

Example 3.7 Let us revisit the example in Table 3.1. When using the TUF-streaming(Time), each node in the TUF-stream contains an additional "last visit" field (i.e., requires slightly more space), we no longer need to visit every node in the

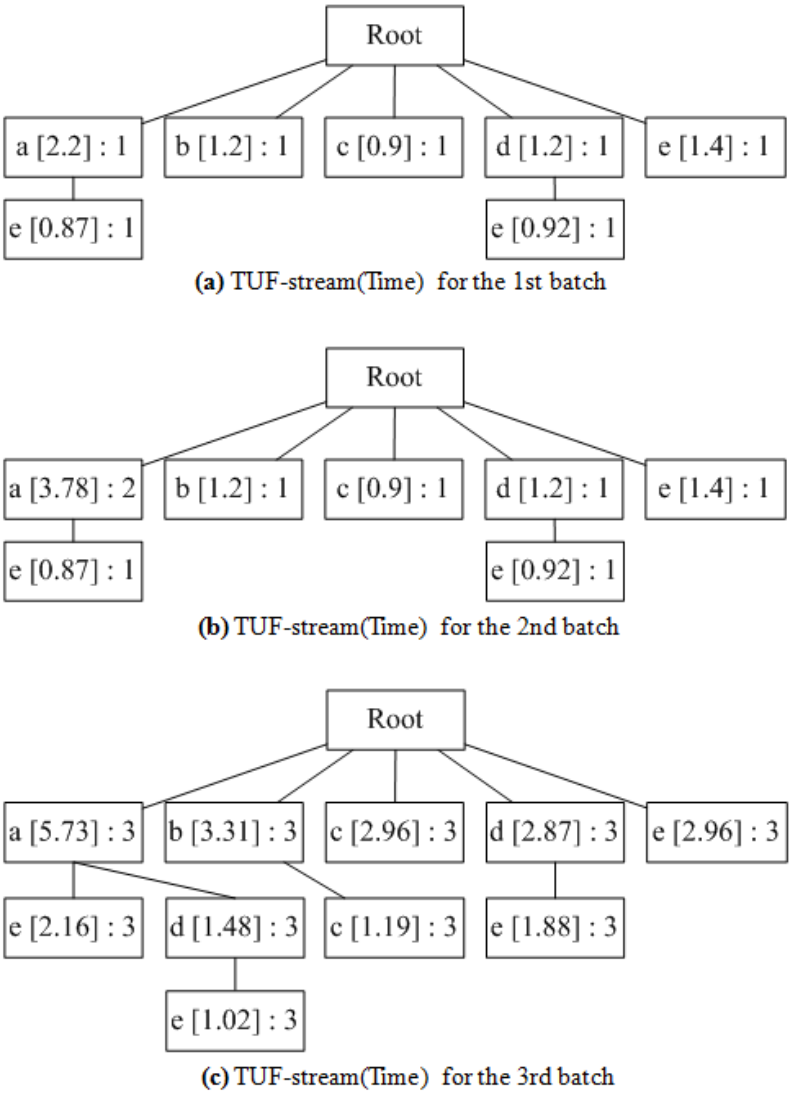


Figure 3.4: The TUF-streaming(Time) Mining Example.

TUF-stream after mining a batch (i.e., takes less time). This is a space-time tradeoff.

Our TUF-streaming(Time) uses the "last visit" field in Equation (3.5) (the variable LV) for computing expected support values to be stored in the TUF-stream for the time-fading model. Unlike the TUF-streaming(Space) that visits every node, the

TUF-streaming(Time) visits only "frequent" nodes. After mining B_1, the algorithm visits and stores seven expected support values (i.e., $2.2, 1.2, 0.9, 1.2, 1.4, 0.87$ and 0.92 for "frequent" patterns $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{a, e\}$, and $\{d, e\}$, respectively) in the TUF-stream. The "last visit" fields for all these eight nodes are "1", indicating that they were last visited in Batch B_1. For example, the node $c[0.9] : 1$ shown in Figure 3.4(a) indicates that $\{c\}$ with expected support of 0.9 was last visited in Batch B_1.

Then, for B_2, only one pattern is "frequent"($\{a\}$ with support value 1.8). TUF-streaming(Time) only visits and updates the node represents $\{a\}$. It visits the node for $\{a\}$ and updates its expected support (to 3.78) by multiplying the old $expSup(a, B_1) = 2.2$ by α and then adding $expSup(a, B_2) = 1.8$ to the product: $2.2\alpha + 1.8 = 3.78$. It results in $a[3.78] : 2$ as shown in Figure 3.4(b). For those patterns that are not "frequent", the algorithm delays visiting to that node. It explains why the node $\{c\}$ is represented as $a[0.9] : 1$, which means $\{c\}$ with expected support of 0.9 was last visited(updated) at Batch B_1.

After mining B_3, TUF-streaming(Time) visits and updates four nodes, including $\{c\}$. As shown in Figure 3.4(c), the node is represented as $c[2.96] : 3$, which is the calculation result of: $expSup(c, B_1 \cup B_2 \cup B_3) = 0.9 \times \alpha^{3-1} + expSup(c, B_3) \approx 2.96$.

And finally, when the user requires the final mining result, we access every tree node in the entire TUF-stream tree. If the support value in the tree node is up-to-date, the algorithm compares it with *minsup* and return the pattern as frequent if the support value of this pattern is greater than or equal to *minsup*. If the support value is not the latest ("last visit" value shows there is a gap between now and the last

update), the algorithm then calculates the support value using Equation 3.5. Then, the algorithm compares with *minsup*, and determines if this is a frequent pattern. ■

By doing so, TUF-streaming(Time) visits nodes only for "frequent" patterns mined in a batch. It requires less runtime than TUF-streaming(Space), which visits every node in the TUF-stream.

The above two algorithms (TUF-streaming(Space) and TUF-streaming(Time)) are two algorithms that reduce the memory usage and speed up runtime, respectively. However, in real-life applications, there are a lot more challenges/problems that we need to overcome/solve. In the next section, we will discuss one more contribution of our research, which solved another real-life problem.

3.3 An Extension of TUF-streaming Algorithm: TUF-streaming(Delay)

Let us consider a situation shown in Figure 3.5. The central sever (the middle circle) collects uncertain data streams from several different nodes (for example, thermal sensors or environment surveillance). Each node sends its new data to central server every certain amount of time in sequence. All these nodes connect to the central server by a network. Assume that all data batches have a date/time information (called "time tag"), and all data streams have to be processed (by the central server) in the order of date/time. What will happen if two data batches were sent out in different time by different nodes but arrived the central server at the same time? What will happen if one of the networks suddenly runs into issues and delays the data

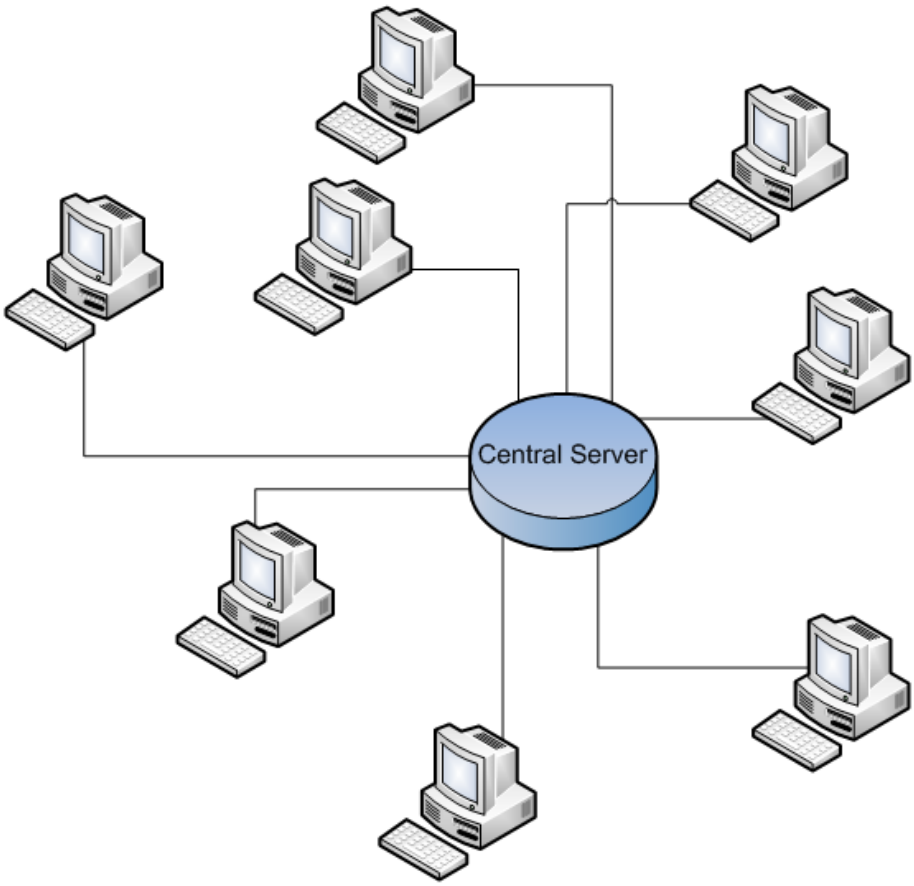


Figure 3.5: Data Stream Delay Situation.

batch delivery? In this thesis, we also design an extended version of TUF-streaming to handle this situation. Theoretically, the processes that we want to perform are as follows:

- Normally, all data streams are processed on a first-send-first-serve order.
- if two or more data streams arrive at the same time, depending on the date/time information (the "time tag"), the algorithm will process them in date/time order,

- if a data stream was delayed, when it arrives at central server, each corresponding frequent pattern (tree node) in the TUF-stream needs to extract the last n support values (depends on how "late" the new data stream is) from the calculated support value in the tree node. Then, the algorithm should re-calculate the correct support value with the delayed support value in the correct order.

Recall that both Equation (3.4) and Equation (3.5) calculate the support value of patterns during the tree-updating process. If the a support value has been processed by the algorithm, there is no way we can undo the calculation and extract previous support values back. Take Equation (3.4) as example, if the process of $[expSup(X, \cup_{i=1}^{T-1} B_i) \times \alpha] + expSup(X, B_T)$ has been performed, we are not be able to retrieve the $expSup(X, B_T)$ back any more. In other words, if new data arrive late, the value is $expSup(X, \cup_{i=1}^{T} B_i)$. There is no way to undo the calculation and get the value of $expSup(X, B_{T-1})$, in order to perform $[expSup(X, \cup_{i=1}^{T-2} B_i) \times \alpha] + expSup(X, B_{T-1}) \times \alpha + expSup(X, B_T)$.

Example 3.8 Let us consider the following example. A TUF-streaming(Space) algorithm is running with $\alpha = 0.9$. In the TUF-stream tree, the tree node which represents pattern $\{a\}$ contains a value 0.5 as the support value of $\{a\}$ in the first data batch. Then, the second data batch flows in, in which patter $\{a\}$ is also frequent with a support value 0.5. The TUF-streaming(Space) algorithm uses Equation (3.4) to compute the current support value in the tree node for pattern $\{a\}$ as: $expSup\left(\{a\}, \bigcup_{i=1}^{2} B_i\right) = [expSup\left(\{a\}, \cup_{i=1}^{2-1} B_i\right) \times 0.9] + expSup(\{a\}, B_2) = 0.5 \times 0.9 + 0.5 = 0.95$. Now, the third data batch arrives, in which $\{a\}$ is frequent with support value 0.7. Before the processing, the algorithm realizes that, this "third" data batch

was "delayed". It supports to be the "second" data batch. The algorithm needs to consider the new data batch as the second data batch, and re-compute the value in the tree node as: $0.5 \times 0.9 + 0.7 = 1.15$. Then, process the old "second" data batch as the third data batch. However, the only information that the algorithm can retrieve from the existing tree node is the single value 0.95. There is no more information for the algorithm to undo the previous computation. ■

To solve this problem, we introduce a buffer array into our data structure. This buffer array temporarily stores a certain number of support values (in the order time information). The size of the buffer array (B) need to be specified before the algorithm starts. For all delayed data batches, they have T_B amount of time to "catch up". In other words, users can determine how long they want to give to a delayed data streams by assigning a value to B. When the array is full, the algorithm applies TUF-streaming to the "oldest" value in the array, and stores it into the tree node. Now the process of storing data becomes:

When a new patten X with a support value $sup(X)$ needs to be stored into the TUF-stream, the algorithm performs the following five steps:

1. (Special case) If $sup(X) = 0$ and also the latest element in the buffer array also equals to 0, we only need to insert one 0 in the end of the array.

2. Check the "time tag" of the latest element in the buffer array, compare it with the "time tag" of X.

3. If the "time tag" of X is later than the latest element in the buffer array, insert $sup(X)$ in the end of buffer array.

Table 3.2: Data Stream Example for Extension of TUF-streaming

Batches	**Transactions**	**Contents**	**Time Tag**
1st	t_1	{a:0.9}	5:00
	t_2	{a:0.8}	
2nd	t_3	{b:0.75}	5:15
	t_4	{b:0.75}	
3rd	t_5	{a:0.8}	4:00
	t_6	{a:0.7}	
4th	t_7	{c:0.5}	6:00
	t_8	{c:0.7}	
5th	t_9	{a:0.5}	5:30
	t_{10}	{a:0.9}	
6th	t_{11}	{b:0.6}	7:00
	t_{12}	{b:0.7}	
7th	t_{13}	{a:0.7}	3:00
	t_{14}	{a:0.5}	
8th	t_{15}	{a:0.35}	6:30
	t_{16}	{a:0.95}	
9th	t_{17}	{c:0.65}	6:30
	t_{18}	{c:0.7}	

4. If the "time tag" of X is earlier than the latest element in the buffer array, move on to the 2nd latest element, compare the "time tag" with X. Repeat this process until reach the correct location in the buffer array for inserting $sup(X)$.

5. After any of the above insertions, if the buffer array is full, shift the earliest element in the buffer array out. Call TUF-streaming process the element.

6. If the "time tag" of X is earlier than all elements in the buffer array, worst case, we call TUF-streaming process pattern X immediately (this process could be already too late, which means, users need a larger size B for buffer array).

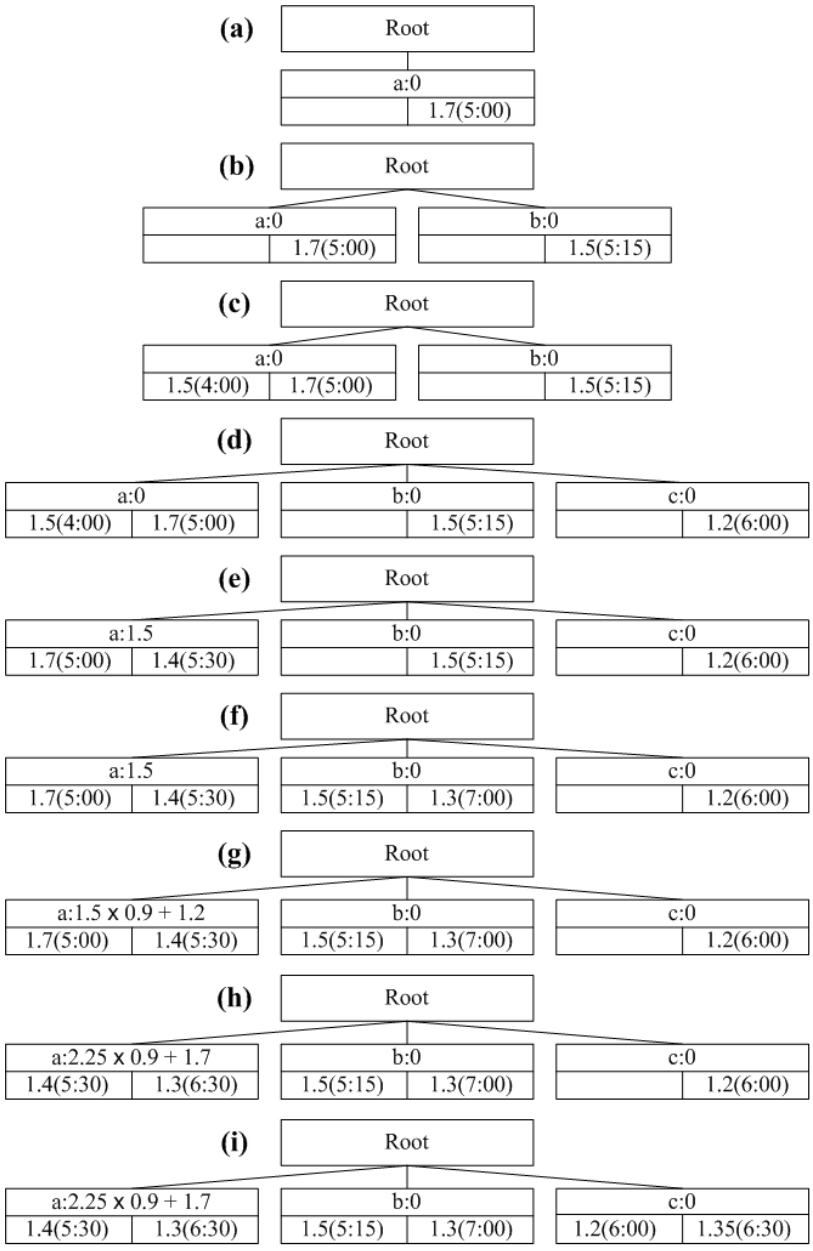


Figure 3.6: TUF-streaming(Delay) Example.

Example 3.9 Consider a sequence of data batches that flows in to the cental server, shown in Table 3.2. For simplicity, let us assume that, in Table 3.2, every data stream only contains two transactions. Every transaction only contain one item.

Let us set the size of the buffer array to 2. The process of how the algorithm works is shown in Figure 3.6. The following two time points are special cases: (1) Batch number 3 (in Figure 3.2(c)), the time tag of the data {a:1.5} (4:00) is early than the old data in the buffer array (5:00). Hence, the algorithm inserts the {a:1.5} (4:00) before {a:1.7} (5:00). (2) Batch number 7 (in Figure 3.2(g)), the time tag of data {a:1.2} (3:00) is early than all old data in the buffer array. Hence, the algorithm processes {a:1.2} (3:00) immediately. ■

One important limitation of this extended version we have to discuss is that, for this version of TUF-streaming, we can only apply TUF-streaming(Space), but not TUF-streaming(Time). The reason is as follows. The TUF-streaming(Time) takes the advantage that in Equation (3.5), when a support value is 0, it simply adds the part of $expSup(X, B_T)$ as 0. The only thing that matters is the α^{T-LV}. Hence, we can use an extra integer to record the LV (last visit/update), and use it to complete the calculation. However, in this "delayed" version, if we have a case where multiple 0's were already added into the TUF-stream tree node support value, we lost all information about when was each of the 0's passed in. If a "delayed" pattern suddenly flows in and the "time tag" of the "delayed" pattern happens to be in between all those 0's, then we are not be able to determine the value of α^{T-LV}.

3.4 Summary

In this chapter, we proposed five algorithms using two window models. First, we proposed the LUF-streaming algorithm, which uses the landmark window model for uncertain data stream mining. Specifically, the LUF-streaming algorithm calls

UF-growth with *preMinsup* to mine frequent patterns from each batch of uncertain data streams. Once the potential frequent patterns are mined from each batch, they are immediately stored into a tree-based data structure called LUF-stream, in which each tree node represents a frequent pattern. As LUF-streaming uses the landmark window model, the LUF-stream treats each batch of the same importance. Hence, when updating the LUF-stream for a new batch, it just visits/updates the nodes corresponding to those newly mined patterns.

Then, we proposed the family of four TUF-streaming algorithms, all of which use the the time-fading window model for uncertain data stream mining. Specifically, TUF-Streaming(Naïve) mines the data stream by calling UF-growth with $preMinsup$. The corresponding data structure, called TUF-stream, stores subfrequent patterns mined from each batch. As TUF-streaming uses the time-fading window model, historical data are considered to be less important than recent data and hence carrying lighter weights (as represented by the α value, which is ≤ 1).

To enhance TUF-Streaming(Naïve), we came up with two additional algorithms in this TUF-streaming family. By modifying Equation (3.2), we reduce the size of the window from potentially unbounded number of incoming batches to one. By doing so, TUF-streaming(Space) reduces greatly the size of TUF-stream.

However, the tree updating process of TUF-streaming(Space) requires a visit to every node in the entire tree when processing a new data stream. So, we proposed TUF-streaming(Time), which further modified the Equation (3.4) by introducing a new variable called LV ($lastvisit/update$) so that TUF-streaming(Time) no longer needs to update the entire tree whenever it processes new batch. Instead, it only

visits/updates those tree nodes corresponding to frequent patterns mined from the recent batch, and updates the LV variables in those tree nodes. By adding the variable of LV into the TUF-stream, TUF-streaming(Time) successfully reduced the runtime.

Finally, we proposed TUF-streaming(Delay) to handle "delayed" data streams. To handle the situation in which there is delay of incoming batches, TUF-streaming(Delay) adds a buffer array to temporarily store incoming data. The size of this buffer array is determined by users. The larger its size, the "longer" the algorithm can "wait".

Chapter 4

Evaluation

In this chapter, we evaluated our proposed algorithms analytically and experimentally. For analytical evaluation, we analyzed the memory usage as well as runtime complexity (in terms of the number of subfrequent patterns mined from each batch of the uncertain streams).

For experimental evaluation, we compared the proposed algorithms with an existing algorithm (UF-streaming [LH09]). Note that three different windowing techniques are used in these algorithms: (1) UF-streaming uses the sliding window model, (2) LUF-streaming uses the landmark window model, and (3) the four TUF-streaming algorithms use the time-fading window model.

4.1 Analytical Evaluation

In this M.Sc. thesis, we proposed five algorithms for discovering frequent patterns from streams of uncertain data using the landmark window model and the time-fading

window model. In this section, let $|fp_i|$ denote the number of frequent patterns mined from Batch B_i. We analytically evaluated our proposed algorithms in two aspects: memory usage and runtime.

4.1.1 Memory Usage

When using the landmark window model, the LUF-streaming algorithm requires $|\bigcup_i fp_i|$ *expSup* values to be stored in the LUF-stream to store frequent pattern information. When using the time-fading model, the TUF-streaming(Naïve) algorithm requires the largest amount of space among the five proposed algorithms because it requires $w \times |\bigcup_i fp_i|$ expected support values to be stored in the TUF-stream (where w is the number of batches mined so far). In contrast, TUF-streaming(Space) requires the least amount of space because each node only stores a single value (i.e., a total of $|\bigcup_i fp_i|$ values), whereas TUF-streaming(Time) requires slightly more space than TUF-streaming(Space) because each node needs to keep the LV (last visit) field in addition to the usual expected support value for each frequent pattern (i.e., $|\bigcup_i fp_i|$ support value + $|\bigcup_i fp_i| \times LV$ values for a total of $2 \times |\bigcup_i fp_i|$ values). However, it is bounded (cf. an unbounded or potentially infinite list of expected support values in TUF-streaming(Naïve) for huge or protanomaly infinite w). Moreover, such a slight increase in space usually pays off as the TUF-streaming(Time) reduces the runtime. The TUF-streaming(Delay), on the other hand, follows the process of TUF-streaming(Space). The memory usage of TUF-streaming(Delay) will be $|\bigcup_i fp_i|$ (the amount of memory that TUF-streaming(Space) uses) plus the size of the delay buffer array (|Buf|) for each tree node (for each frequent pattern). In details, $|\bigcup_i fp_i|$

Table 4.1: Memory Usage Analytical Evaluation

Algorithm	**Memory Usage for w Data Batches**
LUF-streaming	$\lvert\bigcup_i fp_i\rvert$
TUF-streaming(Naïve)	$w \times \lvert\bigcup_i fp_i\rvert$
TUF-streaming(Space)	$\lvert\bigcup_i fp_i\rvert$
TUF-streaming(Time)	$2 \times \lvert\bigcup_i fp_i\rvert$
TUF-streaming(Delay)	$(1 + \lvert\text{Buf}\rvert) \times \lvert\bigcup_i fp_i\rvert$

support values + $|\bigcup_i fp_i| \times |\text{Buf}|$ values for a total of $(1 + |\text{Buf}|) \times |\bigcup_i fp_i|$ values. Table 4.1 summarizes the comparison between these algorithms. Let us take a look at the following example.

Example 4.1 After processing batch B_1, $|fp_1| = 10$ subfrequent patterns are mined. Then, when processing batch B_2, $|fp_2| = 7$ subfrequent patterns are mined and two of them have been mined from B_1 (i.e., only five new patterns). Then, LUF-streaming stores $|\bigcup_i fp_i| = 10 + (7 - 2) = 15$ expected support values in the LUF-stream tree. TUF-streaming(Naïve) stores $w \times |\bigcup_i fp_i| = 2 \times (10 + (7 - 2)) = 30$ values in the TUF-stream tree. TUF-streaming(Space) stores $|\bigcup_i fp_i| = 10 + (7 - 2) = 15$ values in the TUF-stream tree. TUF-streaming(Time) stores $w \times |\bigcup_i fp_i| = 2 \times (10 + (7 - 2)) = 30$ values in the TUF-stream tree. Finally, let us assume users specified the size of the buffer array as 9. Then, TUF-streaming(Delay) stores $(1 + |\text{Buf}|) \times |\bigcup_i fp_i| = (1 + 9) \times (10 + (7 - 2)) = 150$ values in the TUF-stream tree. ■

4.1.2 Runtime

As for the runtime, for the landmark window model, initially, the LUF-streaming needs to visit/update every node in the LUF-stream tree for each data batches. How-

ever, we can increase the performance in a way that only visit/update those tree nodes, which correspond to frequent patterns in current data stream. For the time-fading window model, both TUF-streaming(Naïve) and TUF-streaming(Space) need to visit/update every node in the TUF-stream regardless whether or not the corresponding pattern is subfrequent, i.e., visit $|\bigcup_i fp_i|$ nodes for each of b batches for a total of $b \times |\bigcup_i fp_i|$ visits. In contrast, TUF-streaming(Time) only visits those nodes corresponding to frequent patterns in current data stream, and it requires $\sum_i |fp_i|$ visits. Similarly, the same comments apply to the landmark model, except that they require less runtime than the time-fading model due to simpler calculation. Similar to simplification of Equation (3.2) to Equation (3.3), computation of expected support values become simpler when $\alpha = 1$ (for the landmark model) because the terms $[expSup(X, \cup_{i=1}^{T-1} B_i) \times \alpha]$ and $[expSup(X, \cup_{i=1}^{LV} B_i) \times \alpha^{T-LV}]$ in Equations (3.4) and Equation (3.5) can be simplified to become $(X, \cup_{i=1}^{T-1} B_i)$ and $(X, \cup_{i=1}^{LV} B_i)$, respectively. Moreover, our LUF-streaming algorithm also only visits those nodes corresponding to frequent patterns but may require less runtime than TUF-streaming(Time) due to the absence of the "LV" (last visit) variable. For the TUF-streaming(Delay), since it follow the process that TUF-streaming(Space) has, the total number of tree node updating for TUF-streaming(Delay) is $|\bigcup_i fp_i|$. Table 4.2 summaries the comparison between these algorithms. Let us take a look at the following example.

Example 4.2 Consider the same situation as in Example 4.1. After processing batch B_1, $|fp_1| = 10$ subfrequent patterns are mined. Then, when processing batch B_2, $|fp_2| = 7$ subfrequent patterns are mined and two of them have been mined from B_1 (i.e., only five new patterns). For processing this data batch, the number

Table 4.2: Runtime Analytical Evaluation

Algorithm	**Number of Tree Node Updates for Data Batch i**
LUF-streaming	$\|fp_i\|$
TUF-streaming(Naïve)	$\|\bigcup_i fp_i\|$
TUF-streaming(Space)	$\|\bigcup_i fp_i\|$
TUF-streaming(Time)	$\|fp_i\|$
TUF-streaming(Delay)	$\|\bigcup_i fp_i\|$

of tree node visits/updates for each algorithm is as follows. The LUF-streaming requires $|fp_i| = 7$ tree node visits/updates in its LUF-stream tree. The TUF-streaming requires $|\bigcup_i fp_i| = 10 + (7 - 2) = 15$ tree node visits/updates in its TUF-stream tree. The TUF-streaming(Space) also requires $|\bigcup_i fp_i| = 10 + (7 - 2) = 15$ tree node visits/updates in its TUF-stream tree. The TUF-streaming(Time) requires only $|fp_i| = 7$ tree node visits/updates in its TUF-stream tree. For TUF-streaming(Delay), requires $|\bigcup_i fp_i| = 10 + (7 - 2) = 15$ tree node visits/updates in its TUF-stream tree. ■

4.2 Experimental Evaluation

In this section, we discuss the experimental evaluation of our proposed algorithms. Since TUF-streaming(Delay) performs the same process as TUF-streaming(Space), in terms of runtime, based on the evaluation result we had, they are almost identical. Hence, we took it off from the chart tables in this section.

We use the following two data sets in this evaluation: (1) IBM synthetic data set [AS94] and (2) Mushroom data set (from the UCI Repository [FA10]). These data sets are well-known benchmark databases in the data mining field.

For IBM synthetic data set, we prepared two sets of data: (50_60) and (10_100). These numbers represent the range of existential probability in the data set. For instance, (50_60) means in the data set, all item sets (patterns) will have support values within the range of 0.5 to 0.6. Similarly, (10_100) means in the data set, all item sets (patterns) will have support values within the range of 0.1 to 1.0. Each of these data sets contain 100000 transactions in total. We divided these transactions into 20 separate data batches (5000 transactions each).

For Mushroom data set, there are total 8124 transactions. We also divided them into 20 data batches (average 406 transactions each).

For each of the above data set we evaluated the following aspects of our proposed algorithms: (1) The effect of the number of batches on runtime, and (2) the effect of the user-specified minimum support threshold *minsup* on runtime. Then, we select algorithms with the overall best performance to compare with an existing uncertain data stream mining algorithm UF-streaming [LH09].

All experiments were run on a 2.4 GHz machine with time-sharing environment. All results in reported figures are based on the average of multiple runs.

4.2.1 Evaluation on Different Number of Data Batches

Experiment 4.1 In this experimental evaluation, we compared the runtime of our proposed algorithms running different numbers of batches on IBM synthetic data set. The minimum support threshold (*minsup*) was set to 1.2 for all algorithms. For fair comparison, we purposely set the α value to 1 for all algorithms in the TUF-streaming family. By doing so, the mining results returned by LUF-streaming were

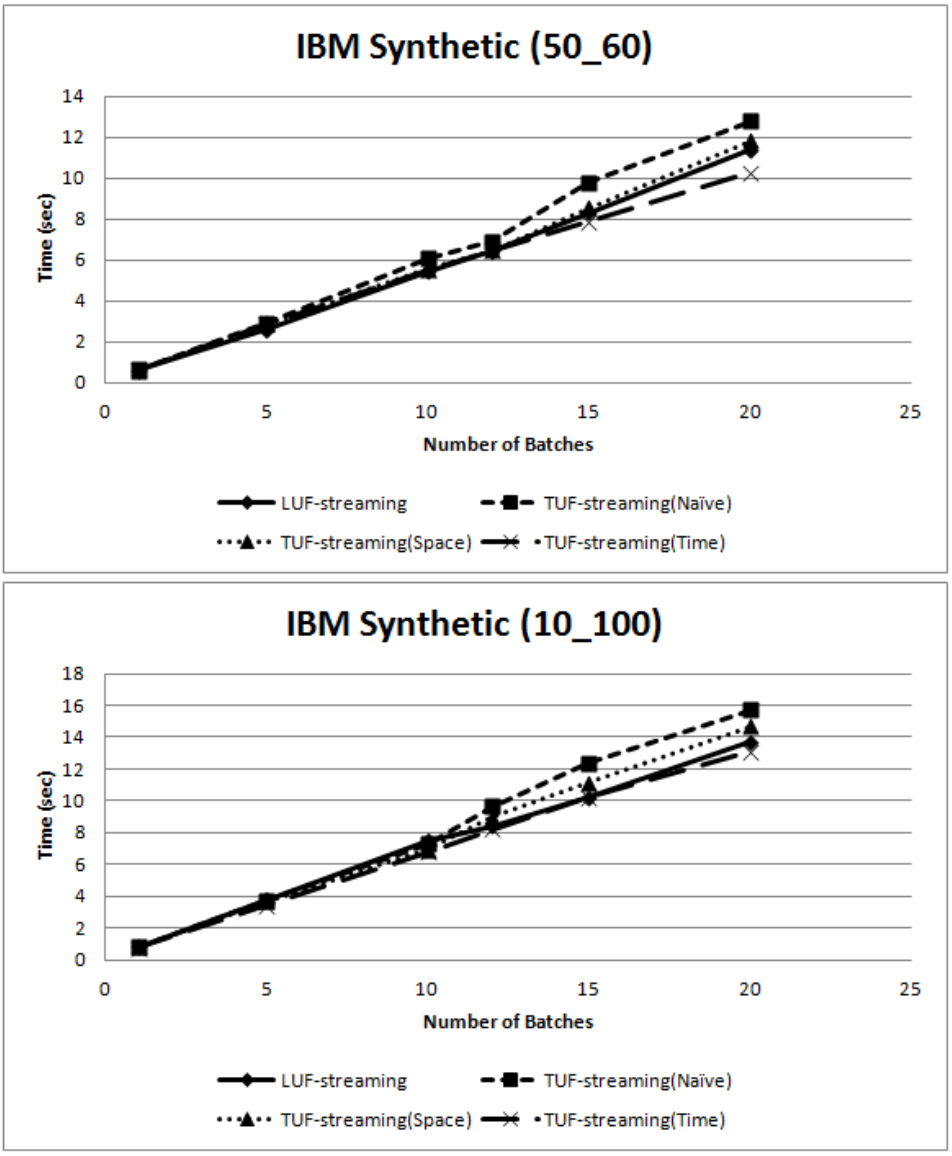


Figure 4.1: Evaluation on Different Numbers of Batches on the IBM Synthetic Data Set (Experiment 4.1)

the same as all algorithms in TUF-streaming family. In Figure 4.1, the x-axis is the number of batches and the y-axis is the execution time. Figure 4.1 shows that, the running time increased with the number of data batches. We observed that the

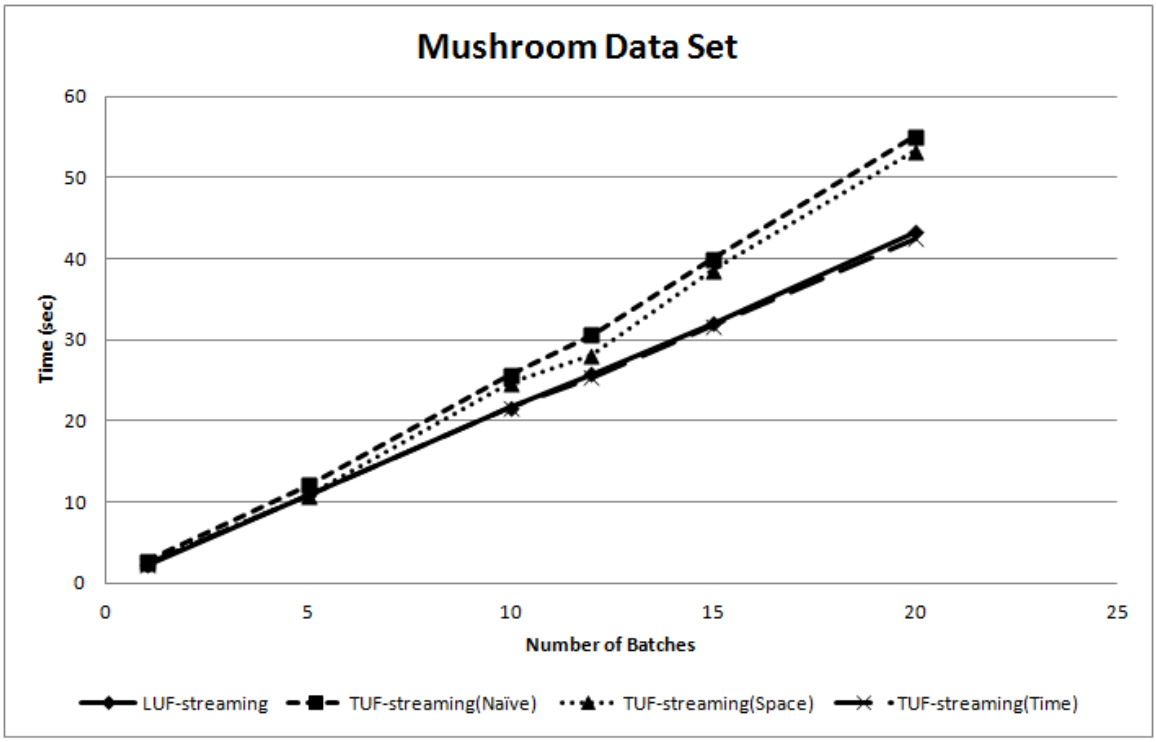


Figure 4.2: Evaluation on Different Numbers of Batches on the Mushroom Data Set (Experiment 4.2)

TUF-streaming(Naïve) took the longest time, while both LUF-streaming and TUF-streaming(Time) took the shortest time.

Experiment 4.2 Figure 4.2 shows the evaluation result for running on mushroom data set. We compared the running time of our proposed algorithms against different numbers of batches. Again, the minimum support threshold was set to 1.2 and the α value for all TUF-streaming family algorithm is 1. The difference in performance between all algorithms were much clearer when using the mushroom data set than the IBM synthetic data set. The reason is that the mining result of mushroom data contains more frequent patterns than the mining result of IBM synthetic data,

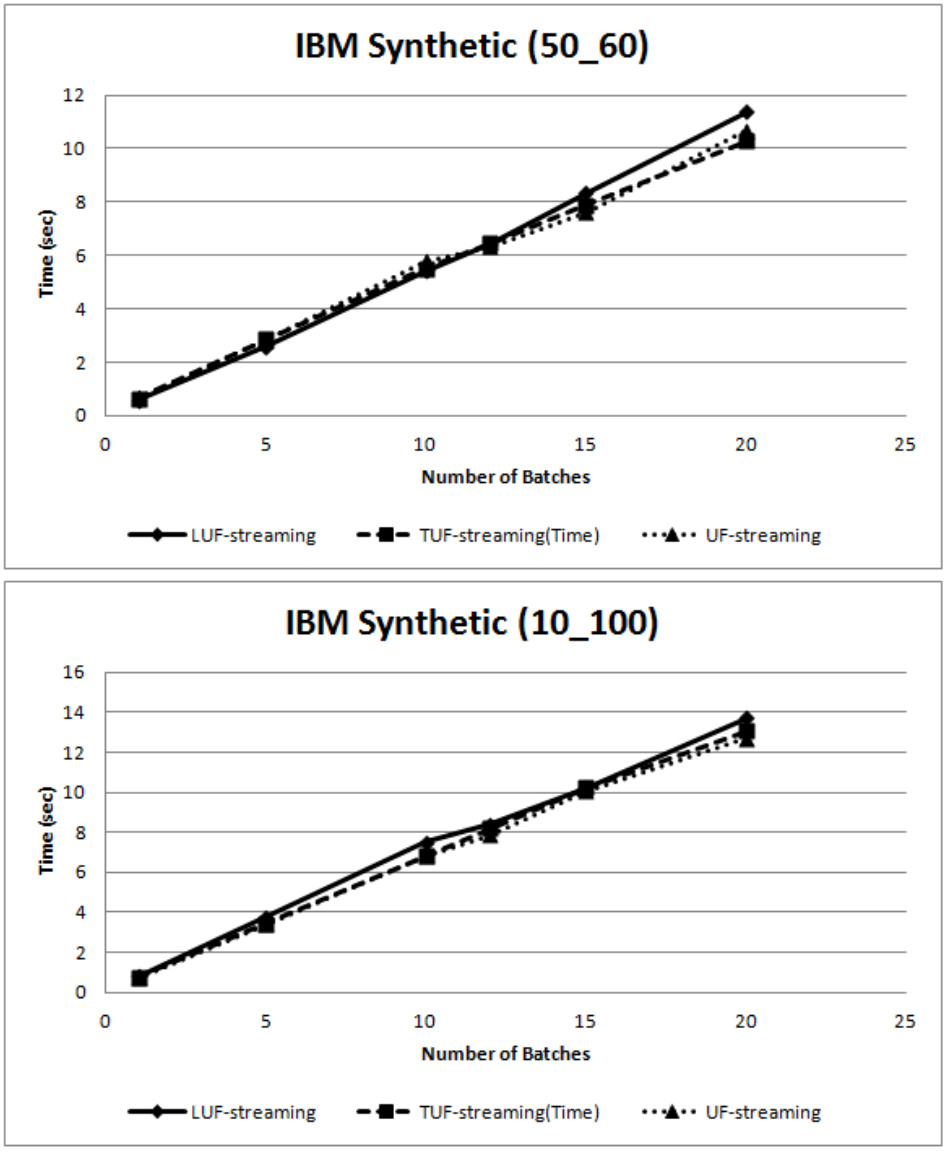


Figure 4.3: Evaluation on Different Numbers of Batches on the IBM Synthetic Data Set (Experiment 4.3)

which leads to a larger TUF-stream tree structure (or LUF-stream structure). Hence, the efficiency of tree updating process becomes more significant. Overall, the TUF-streaming(Naïve) took the longest time. LUF-streaming and TUF-streaming(Time)

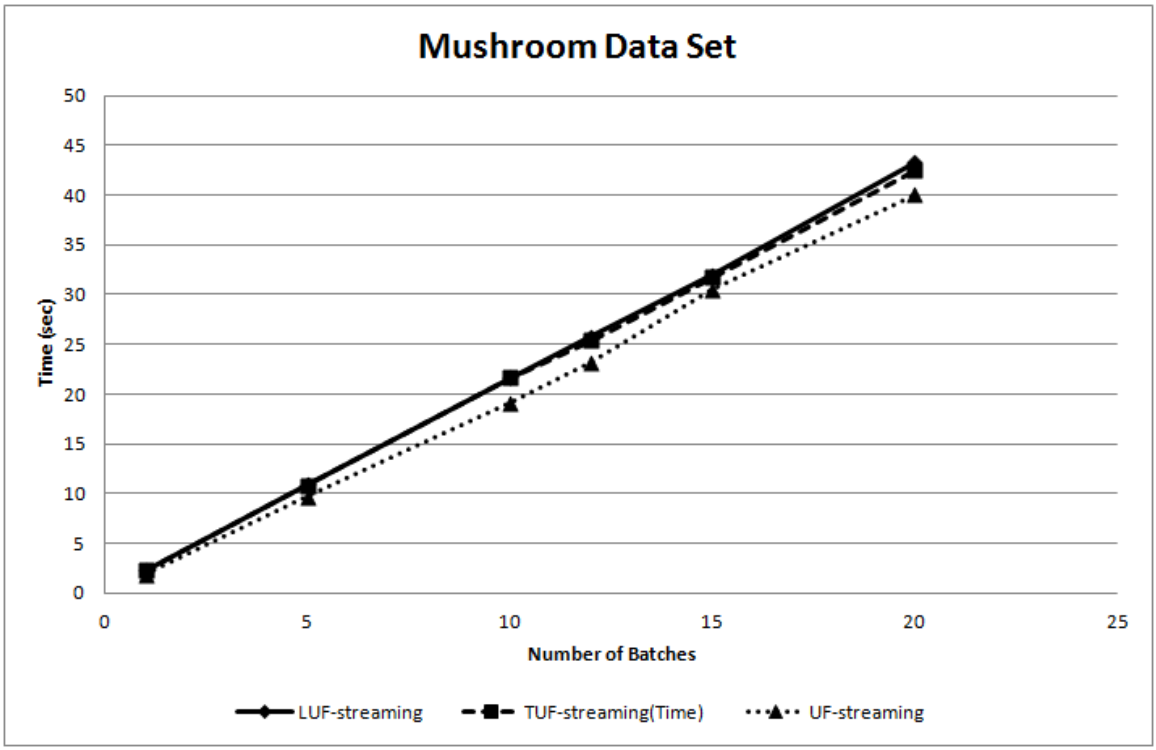


Figure 4.4: Evaluation on Different Numbers of Batches on the Mushroom Data Set (Experiment 4.3)

took the shortest time.

Experiment 4.3 Both TUF-streaming(Time) and LUF-streaming were observed to give an overall best performance in terms of runtime. We used the same approach as previous experiments to compare these two algorithms with an existing algorithm. First, we ran all algorithms against the IBM synthetic data set (Figure 4.3) and Mushroom data set (Figure 4.4) on increasing number of batches. One may argue that, as different window models are applied to these algorithms, the mining results may be different. The result of this experiment shows the ability of our algorithm in handling data streams. It also gives readers an idea about the runtime in terms of

magnitude when compared with others.

4.2.2 Evaluation on Different Minimum Support Thresholds

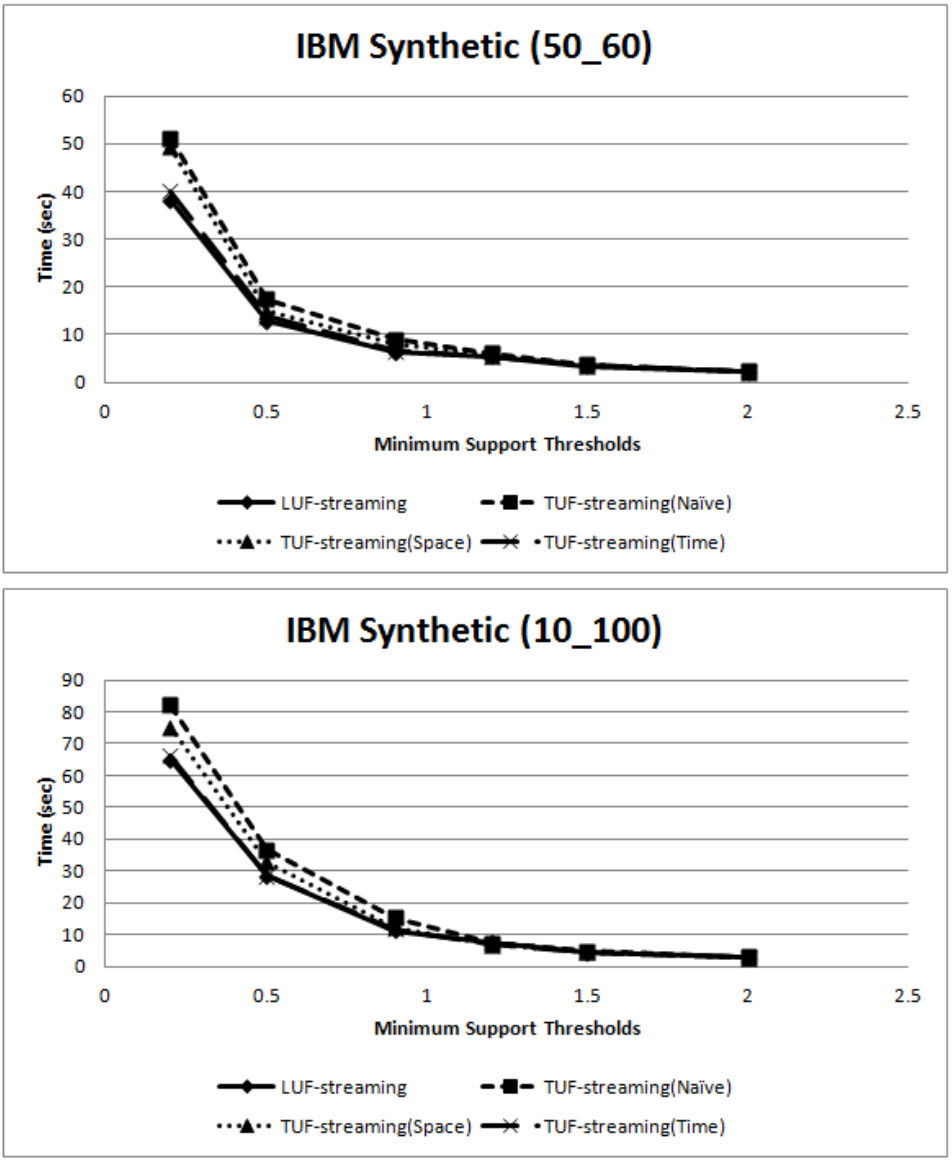


Figure 4.5: Evaluation on Different Minimum Support Thresholds on the IBM Synthetic Data Set (Experiment 4.4)

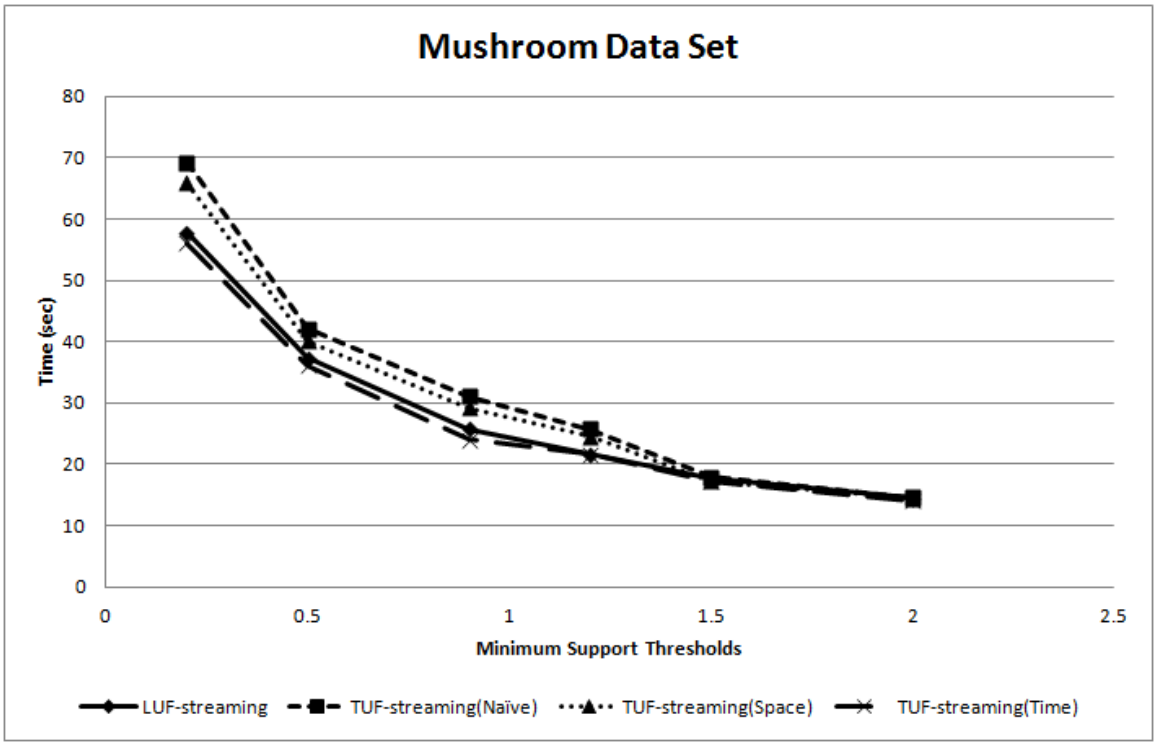


Figure 4.6: Evaluation on Different Minimum Support Thresholds on the Mushroom Data Set (Experiment 4.4)

Experiment 4.4 In this experiment, we evaluated all of our proposed algorithms in different minimum support thresholds on both IBM synthetic data set (Figure 4.5), and Mushroom data set (Figure 4.6). In both figures, the x-axis is the increasing minimum support threshold, and the y-axis is the execution time. We set the number of data batches as to 10 for each run. And, again, to make a fair evaluation, we purposely set the α value to 1 for all TUF-streaming family algorithms. We observed from the evaluation results that the LUF-streaming and the TUF-streaming(Time) took the shortest time among the tested algorithms.

Experiment 4.5 This experiment compared TUF-steaming(Time), LUF-streaming,

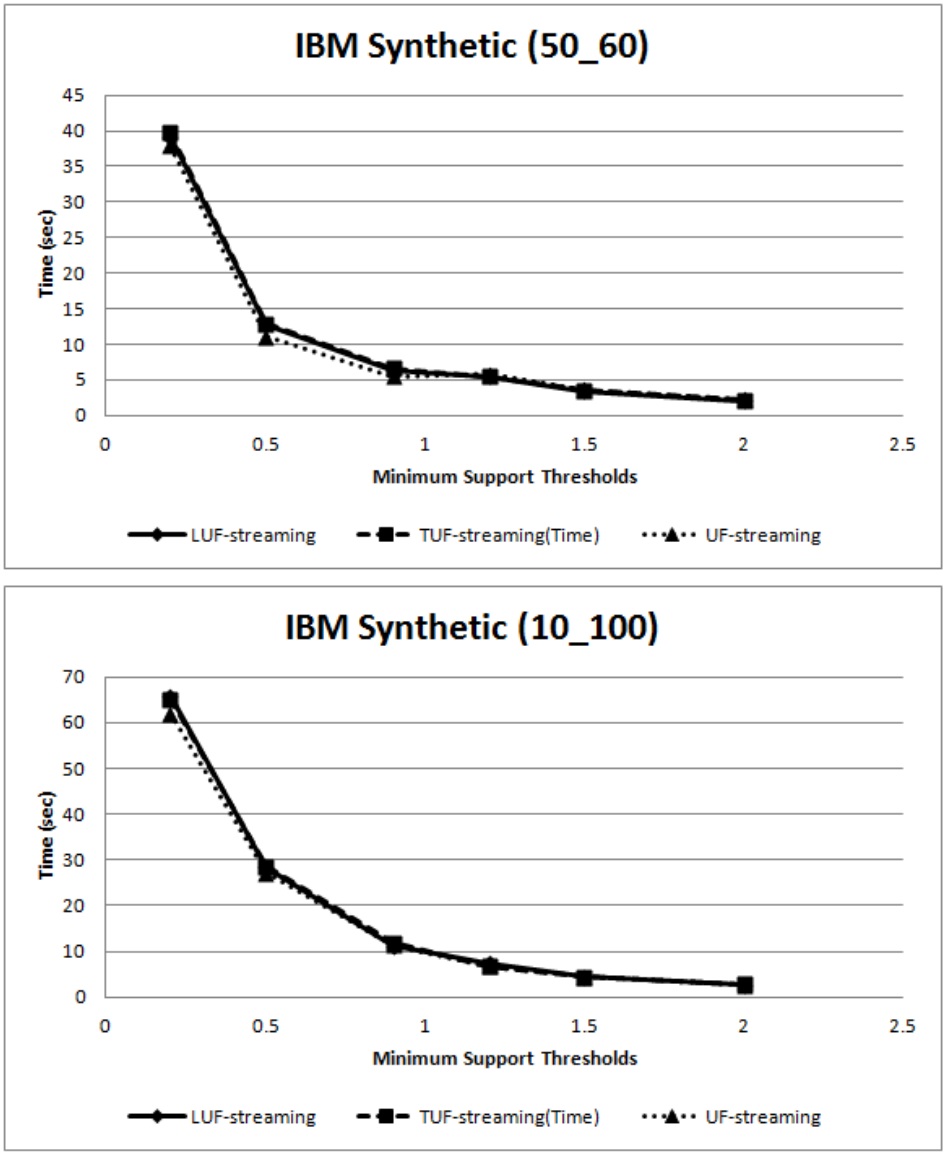


Figure 4.7: Evaluation on Different Minimum Support Thresholds on the IBM Synthetic Data Set (Experiment 4.5)

and UF-streaming running on the IBM synthetic data set (Figure 4.7) and Mushroom data set (Figure 4.8) with different minimum support thresholds. Recall from Experiment 4.3 that the number of frequent patterns returned by UF-streaming is different

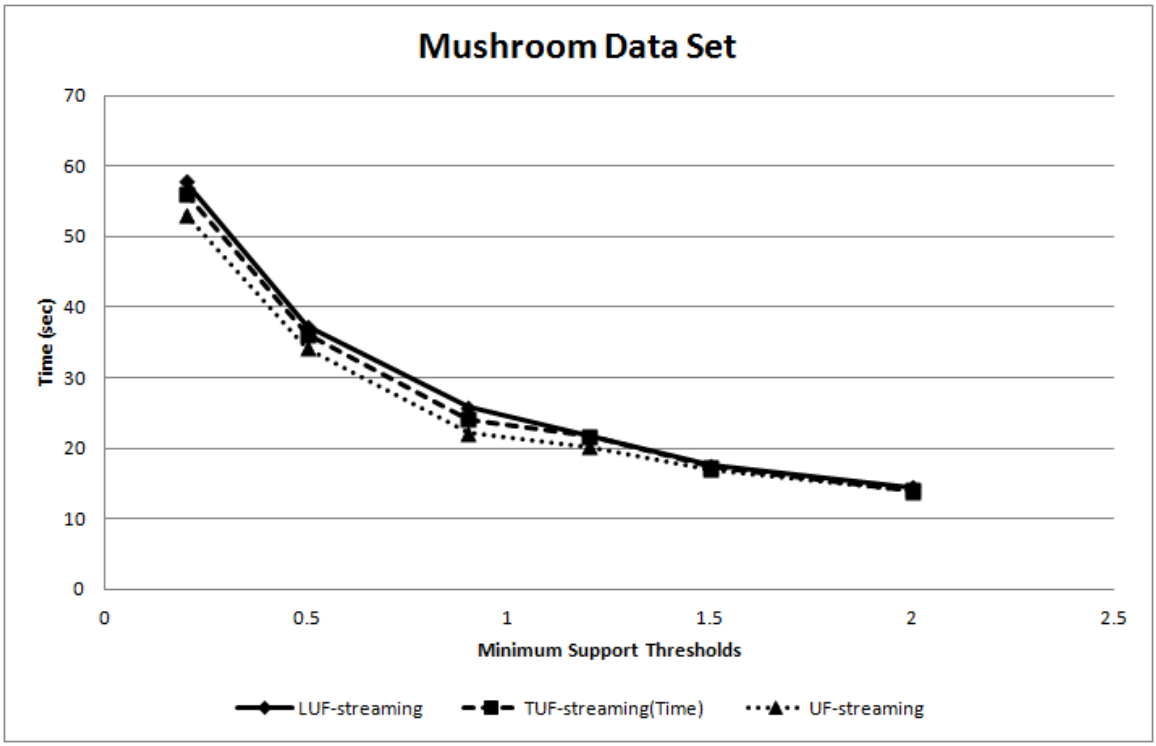


Figure 4.8: Evaluation on Different Minimum Support Thresholds on the Mushroom Data Set (Experiment 4.5)

from the number of frequent patterns returned by our proposed algorithms. This was because the UF-streaming applies a sliding window to discard data. However, we observed that our algorithms perform only slightly slower than UF-streaming.

4.3 Summary

In this chapter, we evaluated our five proposed algorithms analytically and experimentally. For analytical evaluation, we analyzed the memory usage as well as runtime complexity (in terms of the number of subfrequent patterns mined from each batch of the uncertain streams). In terms of memory usage, LUF-streaming

and TUF-streaming(Space) use the least amount of memory because they both keep only one single support value for every batch of the potentially infinite streams. TUF-streaming(Time), on the other hand, requires slightly more memory space than TUF-streaming(Space) due to the extra variable "LV" (last visit) in each tree node of TUF-stream. TUF-streaming(Naïve) uses the largest amount of memory because it keeps the entire window (the time-fading window for TUF-streaming(Naïve)).

In terms of runtime, both LUF-streaming and TUF-streaming(Time) require the shortest runtime due to the enhancement of the tree updating process. Both of them only need to visit/update each tree node corresponding to a frequent pattern mined from the current batch.

For experimental evaluation, we compared the proposed algorithms with the existing UF-streaming [LH09] algorithm. We examined the effect on runtimes by varying the number of data batches and minimum support thresholds. Experimental results on both IBM synthetic data set and Mushroom data show that TUF-streaming(Time) and LUF-streaming require the shortest runtime. Moreover, when the number of batches increased, runtimes for all algorithms increased almost linearly. When the minimum support thresholds increased, the number of frequent patterns decreased, and thus required runtime decreased.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Frequent pattern mining is an important research area in the database and data mining field. It finds out patterns (sets of data) that appear frequently in databases. It has been applied into many real-life applications (e.g., supermarket sales databases, weather report databases, environment databases). Although a lot of research works have been done on frequent pattern mining, most of them are focused on *precise data* where users are certain about the presence of items in databases. However, in many real-life situations, users are uncertain about the presence or absence of items in the database (e.g., quantum physics databases, thermal sensors databases, environment surveillance databases and so on). This type of data is defined as *uncertain data* where each item appears with a "likelihood" value (i.e., uncertainty value). Moreover, usually, the databases in real-life applications are not static; there are new data flowing in continuously at a rapid rate. To mine this type of dynamic databases is

called data stream mining. For this M.Sc. research work, we explored the combination of these three research areas (frequent pattern mining, uncertain data mining, and stream mining) and came up with algorithms for mining frequent patterns from streams of uncertain data.

In this M.Sc. thesis, we proposed the following five algorithms for uncertain data stream frequent pattern mining.

- **LUF-streaming**, which is an algorithm that applies the landmark window model (Section 3.1),
- **TUF-streaming(Naïve)**, which is an algorithm that applyies the time-fading window model (Section 3.2.1),
- **TUF-streaming(Space)**, which extends TUF-streaming(Naïve) by reducing memory usage (Section 3.2.2),
- **TUF-streaming(Time)**, which extends TUF-streaming(Naïve) by speeding up the runtime (Section 3.2.3), and
- **TUF-streaming(Delay)**, which extends TUF-streaming(Naïve) by handling "delayed" streams (Section 3.3).

We conducted both analytical and experimental evaluation on these algorithms. The evaluation results showed the space and time effectiveness of the above algorithms. For instance, in terms of memory usage, both LUF-streaming and TUF-streaming(Space) use the least amount of memory. In terms of algorithm execution efficiency, LUF-streaming and TUF-streaming(Time) has the shortest running time.

In the end, our evaluation shows that LUF-streaming and TUF-streaming(Time) has the best overall performance, We also used TUF-streaming(Time) and LUF-streaming algorithms to compare with existing algorithm UF-streaming. The evaluation results show that both TUF-streaming and LUF-streaming only sacrificed slightly in terms of running time to store a potentially infinite amount of data stream information, while UF-streaming applies a sliding window model (which discards old data streams).

Recall in Section 1.2, we asked eight questions about applying different window models for uncertain data stream mining. In this thesis, we provided answers for all of them:

1. *Are there any other models than the sliding window model that we can use?* In this thesis, we proposed the landmark window model and the time-fading window model.

2. *Can we apply other windowing models to discover frequent patterns from dynamic streams of uncertain data?* We successfully applied the landmark window model and the time-fading window model to solve the problem.

3. *How to construct a data structure that helps discover frequent patterns from dynamic streams of uncertain data?* For the LUF-streaming, the corresponding data structure is called LUF-stream. For the TUF-streaming algorithm family, the corresponding data structure is called TUF-stream.

4. *How does the new data structure efficiently store more information than applying the sliding window model?* Each of the proposed algorithms modifies the equation of finding support values of frequent patterns (i.e., Equation (3.1) and

Equation (3.5))

5. *How can we handle potentially infinite streams of incoming data?* By modifying the equations for calculating support values of frequent patterns, the algorithm does not necessarily need infinite memory to store frequent pattern information from an infinite number of data streams. Specifically, LUF-streaming, TUF-streaming(Space), and TUF-streaming(Time) only require one support value for each frequent pattern in the data structure at any point of time.

6. *Will the mining result stay the same when we apply other window models?* Since the differences between different windowing techniques, the mining results are different from each other. The user needs to choose from all algorithms that we proposed depending on different situations and requirements.

7. *How will other window models affect the mining algorithm in terms of runtime performance?* Based on the evaluation result that we observed, LUF-streaming and TUF-streaming(Time) have the best performance in terms of runtime.

8. *How will other window models affect the mining algorithm in terms of memory usage?* Based on the evaluation result that we observed, LUF-streaming and TUF-streaming(Space) have the best performance in terms of runtime.

In conclusion, we successfully developed algorithms to solve the problem of uncertain data stream frequent pattern mining by applying the landmark window model (the LUF-streaming) and the time-fading window model (the TUF-streaming algorithm family). TUF-streaming(Time) and TUF-streaming(Space) increase the performance in both memory usage and algorithm execution efficiency. TUF-streaming(De-

lay) handles delayed data streams. Evaluation results show the efficiency of all the algorithms, and the compares with existing algorithm.

5.2 Future Work

For the future work of this M.Sc. thesis, we plan to extend the algorithm in the following three directions: (1) data dependency, (2) data streams contrast mining, and (3) hyperlinked-structure-based mining.

This thesis research is done under an assumption that all uncertain data that we used is independent from each other, in which we can consider each data individually. However, if there are some dependency relationships between the data, we could use this relationship to improve the performance. For example, in weather report database, if the possibility of "rainy" is 55% then we can deduce that the possibility of "sunny" together with the possibility of "cloudy " is $100\% - 55\% = 45\%$. The research of this topic requires a significant modification on the data structure since the representation of dependency relationship needs to be efficient and it has to be able to coordinate with the data structure for storing all frequent patterns.

The second future research direction is contrast pattern mining. In many real-life situations, usually people not only want to know what pattern is frequent, they also want to know the difference between two collections of frequent patterns. In other words, they want to compare and contrast the mined frequent patterns. This leads to contrast pattern mining, which is becoming an important research direction in data mining. In this M.Sc. thesis, we processed data streams, which can be considered as several collections of frequent patterns. The contrast mining on these streams is a

topic that is worth further research.

Last but not the least, the third future research direction is the hyperlinked-structure-based mining. In this thesis, we described and compared two major types data mining algorithms: the Apriori-based and the tree-based. Our proposed algorithms use tree-based mining. However, hyperlinked-structure-based mining exists, and the UH-mine [ALWW09] is an alternative algorithm for mining frequent patterns from *static* uncertain data. We could extend UH-mine for dynamic stream mining. For this M.Sc. thesis, we focus on the tree-based algorithm. A challenge is that the underneath data structures between the tree-based algorithms and the hyperlinked-structure-based algorithms are completely different.

Bibliography

[ALWW09] Charu C. Aggarwal, Yan Li, Jianyong Wang, and Jing Wang. Frequent pattern mining with uncertain data. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2009), Paris, France*, pages 29–38. ACM, 2009.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB 1994), Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann Publishers Inc., 1994.

[Bod05] Ferenc Bodon. A trie-based APRIORI implementation for mining frequent item sequences. In *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations (OSDM 2005), Chicago, IL, USA*, pages 56–65. ACM, 2005.

[CE06] Feng Cao and Martin Ester. Density-based clustering over an evolving data stream with noise. *Science*, (2):326–337, 2006.

[CGG10] Toon Calders, Calin Garboni, and Bart Goethals. Efficient pattern mining of uncertain data with sampling. In *Proceedings of the 14th Pacific Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD 2010), Hyderabad, India*, pages 480–487. Springer-Verlag, 2010.

[CGWD10] Malu Castellanos, Chetan Gupta, Song Wang, and Umeshwar Dayal. Leveraging web streams for contractual situational awareness in operational BI. In *Proceedings of the 2010 International Conference on Extending Database Technology/International Conference on Database Theory (EDBT/ICDT 2010) Workshops, Lausanne, Switzerland*, pages 7:1–8. ACM, 2010.

[CK08] Chun-Kit Chui and Ben Kao. A decremental approach for mining frequent itemsets from uncertain data. In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD 2008), Osaka, Japan*, pages 64–75. Springer-Verlag, 2008.

[CKH07] Chun-Kit Chui, Ben Kao, and Edward Hung. Mining frequent itemsets from uncertain data. In *Proceedings of the 11th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD 2007), Nanjing, China*, pages 47–58. Springer-Verlag, 2007.

[CNOT07] Yueguo Chen, Mario A. Nascimento, Beng Chin Ooi, and Anthony K. H. Tung. SpADe: On shape-based pattern detection in streaming time series. In *Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007), Istanbul, Turkey*, pages 786–795. IEEE, 2007.

[Cuz09] Alfredo Cuzzocrea. CAMS: OLAPing multidimensional data streams efficiently. In *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2009), Linz, Austria*, pages 48–62. Springer-Verlag, 2009.

[DYMV05] Xiangyuan Dai, Man Lung Yiu, Nikos Mamoulis, and Michail Vaitis. Probabilistic spatial queries on existentially uncertain data. In *Proceedings of the 9th International Symposium on Spatial and Temporal Databases (SSTD 2005), Angra dos Reis, Brazil*, pages 400–417. Springer-Verlag, 2005.

[EZ09] C. I. Ezeife and Dan Zhang. TidFP: Mining frequent patterns in different databases with transaction ID. In *Proceedings of the 11th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2009), Linz, Austria*, pages 125–137. Springer-Verlag, 2009.

[FA10] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[GBK10] Anamika Gupta, Vasudha Bhatnagar, and Naveen Kumar. Mining closed itemsets in data stream using formal concept analysis. In *Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2010), Bilbao, Spain*, pages 285–296. Springer-Verlag, 2010.

[GHP+04] Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S. Yu. Mining frequent patterns in data streams at multiple time granularities. *Data Mining: Next Generation Challenges and Future Directions*, pages 191–212, 2004.

[GZK05] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34:18–26, June 2005.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29:1–12, May 2000.

[JG06] Nan Jiang and Le Gruenwald. Research issues in data stream association rule mining. *SIGMOD Rec.*, 35:14–19, March 2006.

[KD11] Yun Sing Koh and Gillian Dobbie. SPO-Tree: Efficient single pass ordered incremental pattern mining. In *Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2011), Toulouse, France*, pages 265–276. Springer-Verlag, 2011.

[LCH07] Carson Kai-Sang Leung, Christopher L. Carmichael, and Boyu Hao. Efficient mining of frequent patterns from uncertain data. In *Proceedings of the Seventh IEEE International Conference on Data Mining Workshops (ICDM 2007), Washington, DC, USA*, pages 489–494. IEEE Computer Society, 2007.

[Leu04] Carson Kai-Sang Leung. Interactive constrained frequent-pattern mining system. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS 2004), Coimbra, Portugal*, pages 49–58. IEEE Computer Society, 2004.

[Leu11] Carson Kai-Sang Leung. Mining uncertain data. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(4):316–329, 2011.

[LH09] Carson Kai-Sang Leung and Boyu Hao. Mining of frequent itemsets from streams of uncertain data. In *Proceedings of the 2009 IEEE 25th International Conference on Data Engineering (ICDE 2009), Shanghai, China*, pages 1663–1670. IEEE Computer Society, 2009.

[LIC08] Carson Kai-Sang Leung, Pourang P. Irani, and Christopher L. Carmichael. WiFIsViz: Effective visualization of frequent itemsets. In *Proceedings of the 2008 IEEE International Conference on Data Mining (ICDM 2008), Pisa, Italy*, pages 875–880. IEEE Computer Society, 2008.

[LJ11] Carson Kai-Sang Leung and Fan Jiang. Frequent pattern mining from time-fading streams of uncertain data. In *Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2011), Toulouse, France*, pages 252–264. Springer-Verlag, 2011.

[LJH11] Carson Kai-Sang Leung, Fan Jiang, and Yaroslav Hayduk. A landmark-model based system for mining frequent patterns from uncertain data streams. In *Proceedings of the 15th International Database Engineering and Applications Symposium (IDEAS 2011), Lisbon, Portugal*, pages 249–250. ACM, 2011.

[LLN03] Laks V. S. Lakshmanan, Carson Kai-Sang Leung, and Raymond T. Ng. Efficient dynamic mining of constrained frequent sets. *ACM Trans. Database Syst.*, 28:337–389, December 2003.

[LMB08] Carson Kai-Sang Leung, Mark Anthony F. Mateo, and Dale A. Brajczuk. A tree-based approach for frequent pattern mining from uncertain data. In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD 2008), Osaka, Japan*, pages 653–661. Springer-Verlag, 2008.

[LS11] Carson Kai-Sang Leung and Lijing Sun. Equivalence class transformation based mining of frequent itemsets from uncertain data. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC 2011), TaiChung, Taiwan*, pages 983–984. ACM, 2011.

[MSL08] George A. Mihaila, Ioana Stanoi, and Christian A. Lang. Anomaly-free incremental output in stream processing. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM 2008), Napa Valley, California, USA*, pages 359–368. ACM, 2008.

[NLHP98] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. *SIGMOD Rec.*, 27:13–24, June 1998.

[OSS02] OSSM: A segmentation approach to optimize frequency counting. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002), San Jose, CA, USA*, pages 583–592. IEEE Computer Societ, 2002.

[PG09] Ardian Kristanto Poernomo and Vivekanand Gopalkrishnan. Towards efficient mining of proportional fault-tolerant frequent itemsets. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2009), Paris, France*, pages 697–706. ACM, 2009.

[PLL+10] Marc Plantevit, Anne Laurent, Dominique Laurent, Maguelonne Teisseire, and Yeow WEI Choong. Mining multidimensional and multilevel sequential patterns. *ACM Trans. Knowl. Discov. Data*, 4:1–37, January 2010.

[YCLZ04] Jeffery Xu Yu, Zhihong Chong, Hongjun Lu, and Aoying Zhou. False positive or false negative: mining frequent itemsets from high speed transactional data streams. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB 2004), Toronto, Canada*, pages 204–215. VLDB Endowment, 2004.

[YLL08] Jeffrey Xu Yu, Zhiheng Li, and Guimei Liu. A data mining proxy approach for efficient frequent itemset mining. *The VLDB Journal*, 17:947–970, July 2008.

[YMD+09] Man Lung Yiu, Nikos Mamoulis, Xiangyuan Dai, Yufei Tao, and Michail Vaitis. Efficient evaluation of probabilistic advanced spatial queries on existentially uncertain data. *IEEE Transactions on Knowledge and Data Engineering*, 21:108–122, January 2009.