Mining Frequent Itemsets from Uncertain Data: Extensions to Constrained Mining and Stream Mining

by

Boyu Hao

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

June 2010

Thesis advisor Author

Dr. Carson K. Leung **Boyu Hao**

Mining Frequent Itemsets from Uncertain Data: Extensions to Constrained Mining and Stream Mining

Abstract

Most studies on frequent itemset mining focus on mining precise data. However, there are situations in which the data are uncertain. This leads to the mining of uncertain data. There are also situations in which users are only interested in frequent itemsets that satisfy user-specified aggregate constraints. This leads to constrained mining of uncertain data. Moreover, floods of uncertain data can be produced in many other situations. This leads to stream mining of uncertain data. In this M.Sc. thesis, we propose algorithms to deal with all these situations. We first design a tree-based mining algorithm to find all frequent itemsets from databases of uncertain data. We then extend it to mine databases of uncertain data for only those frequent itemsets that satisfy user-specified aggregate constraints and to mine streams of uncertain data for all frequent itemsets. Experimental results show the effectiveness of all these algorithms.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First of all, I owe my deepest gratitude to my research supervisor—Dr. Carson K. Leung—for his encouragement, valuable support and advice on my research work. I first met Dr. Leung when he was the professor for my data mining course. His course materials were carefully prepared, well organized, intellectually stimulating, and clearly explained. His enthusiasm and inspiration make me interested in the research area of data mining, which subsequently leads to this M.Sc. thesis research.

I acknowledge Database & Data Mining Laboratory, Department of Computer Science, University of Manitoba for providing me a comfortable research environment, plenty of research equipments, and rapid technical support. I am very glad to study and do research in such a friendly atmosphere. I particularly thank my lab members, Chris Carmichael and Dale Brajczuk. I also acknowledge TRLabs Winnipeg for providing me with a scholarship. It partially covers my living expenses and indeed helps me to focus on this research work.

Thanks to the members of my thesis examination committee, Dr. Xikui Wang, Dr. David H. Scuse, and the chair of my thesis defence, Dr. John E. Anderson, for their precious time to read and examine my thesis.

Last but not the least, I thank my parents and my lovely wife. Their love, care, understanding and support accompany me throughout my graduate study.

Boyu Hao
B.Eng., Heilongjiang University, China, 2002
B.Sc.(Honours), Trent University, Canada, 2004

The University of Manitoba
June 2010

This thesis is dedicated to my parents and my lovely wife.

Chapter 1

Introduction

Frequent itemset mining [AIS93] plays an essential role in data mining, which looks for implicit, previously unknown, and potentially useful information from data. Since Aggrawal et al. [AIS93] introduced the research problem of finding frequent itemsets from traditional databases consisting of precise data, it has been the subject of numerous studies [LNM02, LLN03, Leu04, Bod05, LIC08, YLL08, PG09, PLL$^+$10]. These studies can be broadly divided into two categories: those that focus mainly on performance and those that focus mainly on functionality. The studies in the first category aim to explore how to compute the frequent itemsets as efficiently as possible. The studies in the second category target the questions such as which kinds of patterns to compute and where to mine frequent itemsets.

Regarding the studies that focus on performance, a well-known frequent itemset mining algorithm, called Apriori algorithm [AS94], was proposed in 1994. It depends on a generate-and-test approach. To find frequent itemsets from transaction database, Apriori and its extensions [NLHP98, CKH07] (i.e., Apriori-based algorithms) first generate candidates and then check the occurrences (supports) of these candidates against the transaction database. The candidates with their supports over the user-specified minimum support threshold are counted as valid frequent itemsets.

To avoid memory intensive candidate generation, Han et al. [HPY00] proposed a tree-based frequent itemset mining algorithm called FP-growth (Frequent Pattern growth). The algorithm first scans the transaction database once to find all frequent 1-itemsets and sorts them according to some criteria (e.g., in frequency descending order). Then, it scans the transaction database again to construct an extended prefix-tree, called FP-tree (Frequent Pattern tree), which captures all frequent items in the transaction database. Based on the FP-tree, the FP-growth algorithm mines frequent itemsets by a test-only approach, called frequent pattern growth, which only tests the support for each itemset. As a result, all frequent itemsets are found without candidate generation. As a preview, all the mining algorithms we are going to propose in this thesis use variants of the FP-tree. Our algorithms also avoid generating candidates and only test for support.

Regarding the studies that focus mainly on functionality, they aim to find interesting patterns other than frequent itemsets. Examples of these patterns include correlation [LCZ05], sequences [CWC09], maximal itemsets [Yan04] and closed itemsets [JG06a]. Frequent itemset mining has played an important role in the mining of these patterns. However, most of the studies on frequent itemset mining rely on a computational model, in which data mining algorithm does almost everything and human users are not engaged in the mining process. In other words, these data mining algorithms provide little or no support for user focus. Note that in many real-life applications, the user may be only interested in a tiny portion of mined data. For example, a user may only want to find the frequency information of the snow deeper than 5cm. As another example, the supermarket owner may only want to know the information of the customers with average monthly purchase more than $200. If we apply the regular mining process in these cases, the users need to wait for a long period of time to obtain a big number of frequent itemsets. Among these frequent itemsets, only a tiny fraction may be interesting to the users.

To solve the above problem, **constrained frequent itemset mining** [BAG99, GLW00, LKH06], which aims to find only those frequent itemsets that satisfy the user-specified constraints, has been introduced. Some constrained frequent itemset mining algorithms are based on the Apriori algorithm. For example, CAP [NLHP98] pushes constraints into an Apriori-based algorithm and applies candidate generating-and-testing approach to obtain all valid frequent itemsets. On the other hand, the algorithms such as FPS [LLN02] and $\mathcal{FIC}$ [PHL01] first push the user-specified constraints into the FP-growth based algorithm and then mine frequent itemsets by the frequent pattern growth (divide-and-conquer) approach. As a result, all frequent itemsets which satisfy the user-specified constraints are obtained without candidate generation.

Note that all the studies we have discussed so far are related to the mining of frequent itemsets from traditional *static* databases. Over the past decade, the automation of measurements and data collection has produced tremendously huge amounts of data in many real-life application areas. The recent development and increasing use of a large number of sensors has added to this situation. Consequently, these advances in technology have led to a flood of data. Algorithms for mining these *dynamic* streams are in demand. This calls for **stream mining** [GZK05, LK06a, JG06b, WF06, CH08, LB08, LWW$^+$10]. When comparing with mining from traditional static databases, mining from data streams is more challenging due to the following two properties of data streams:

1. *Data streams are continuous and unbounded.* To find frequent itemsets from data streams, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Hence, we need some techniques to capture the important contents of the streams (e.g., to capture recent data because users are usually more interested in recent data than older ones) and ensure that the captured data can fit into memory.

2. *Data in the streams are not necessarily uniformly distributed and their distributions are usually changing with time.* A currently infrequent itemset may become frequent in the future, and vice versa. So, we have to be careful not to prune infrequent itemsets too early. Otherwise, we may not be able to get complete information such as frequencies of some itemsets (as it is impossible to recall those pruned itemsets).

To find frequent itemsets from data streams, several stream mining algorithms have been proposed. For example, Giannella et al. [GHP$^+$04] proposed a tree-based algorithm called FP-streaming for mining streams of precise data. Leung and Khan [LK06b] proposed two algorithms approxCFPS and exactCFPS to mine constrained frequent itemsets from streams of precise data.

Note that all the aforementioned studies—regardless whether they are Apriori-based or tree-based, mining with constraints or mining without constraints, mining from static databases or mining from data streams—all handle precise data such as databases of market basket transactions, Web logs, and sensor streams. When mining precise data, users definitely know whether an item (or an event) is present in, or is absent from, a transaction in the static databases or dynamic data streams. However, there are situations in which users are uncertain about the presence or absence of some items or events [Agg07, AY08b, CK08, ALWW09, AY09, BKR^{+}09, LC10]. For example, a detective may highly suspect (but not guarantee) that a thief breaks into a house through the window. The uncertainty of such suspicion can be expressed in terms of existential probability. Let us consider a concrete example. I find my car having loud noise when accelerating, so I drive to a garage and seek for help. After a brief check, an automobile mechanic tells me that my car has (a) an 80% likelihood of having a leak in vacuum hose (b) a 30% likelihood of low in engine oil and (c) a 5% likelihood of having a blown exhaust. Each of the above cases is independent, which means the chance of having a leak in vacuum hose is independent of whether the engine oil is low or not. Here, in this uncertain database of car repairing records, each transaction represents one visit of my car to a garage. Of course, my car may have multiple problems at the same time (i.e., multiple items may appear together in the same transaction). Each item (representing a potential problem) in the transaction is associated with an existential probability expressing the likelihood of my car having that problem in that visit. With this notion, each item in a transaction in traditional databases containing precise data can be viewed as an item with a 100% likelihood of being present in the transaction.

In addition to the above examples, there are many other real-life situations (e.g., environmental surveillance, survey research, satellite imaging and candidate election) in which data are uncertain. Hence, mining uncertain data [KP05, ZLY08, VRH$^+$09] is highly in demand. However, most of the previous studies on mining uncertain data focused on data mining tasks like clustering and outlier detection of uncertain data [KP05, AY08a, AY08b, VRH$^+$09]. Recently, there are some studies mined uncertain data for frequent itemsets. For example, Chui et al. [CKH07, CK08] proposed an Apriori-based algorithm called U-Apriori which is a modification of the Apriori algorithm [AS94]. To reduce the number of candidates that need to be counted during the candidate generate-and-test process, the authors also introduced a trimming strategy called LGS-trimming for local trimming, global pruning and single pass patch-up. While the U-Apriori algorithm with LGS-trimming strategy handles uncertain data, the algorithm is Apriori-based (i.e., relies on the candidate generate-and-test paradigm). Hence, some natural questions are: Can we further reduce the number of candidate itemsets to be counted? To a further extent, can we avoid generating candidates at all? When handling precise data, FP-tree based algorithms [HPY00, LLN02] are usually faster than their Apriori-based counterparts [AS94, LLN03]. Is this also the case when handling uncertain data?

1.1 Problem Definition and Thesis Contributions

To answer the above questions, in this thesis, we first propose a tree-based algorithm for further enhancing the performance of U-Apriori algorithm. The key contributions of this part of work include the following: (a) the proposal of an effective tree structure—called UF-tree—for capturing the contents of transactions consisting of uncertain data, and (b) the development of an efficient algorithm—called UF-growth—for mining frequent itemsets from the proposed UF-tree.

The UF-growth algorithm can find all frequent itemsets from uncertain data. However, as in the case for frequent itemset mining from traditional precise data, there are many real-life situations in which the user is interested in only some subsets (and may be some tiny subsets) of all the frequent itemsets. This phenomenon also widely exists in the uncertain cases. For instance, in medicinal diagnosis, a physician may be uncertain about the type of intracranial haemorrhage the patient suffered (uncertain data about each disease). However, he may be only interested in those patients with trauma (instead of all patients) for diagnosing epidural intracranial haemorrhage. As another example, for a product survey, an analyst may be uncertain about the favors of a participator (uncertain data about each reply) but the analyst may be interested in only those replies with certain product or interested in those replies from certain group of participators (e.g., interested in those replies from the participators under 18 years old). This calls for the mining of constrained frequent itemsets from uncertain data [LB09b]. Recently, Leung and Brajczuk [LB09a] designed an algorithm—called U-FPS—to mine uncertain data for frequent itemsets satisfying succinct constraints. Since aggregate constraints like $avg(X.age) \leq 25$ are widely applied in real life. A natural question to ask is: Is it possible to mine uncertain data for only those frequent itemsets that satisfy user-specified aggregate constraints? In response to this question and following the work of developing UF-growth algorithm to mine all frequent itemsets from uncertain data, we propose a tree-based algorithm—called ACUF-growth (indicates Aggregate Constraint UF-growth)—for mining from uncertain data those frequent itemsets that satisfy user-specified aggregate constraints. The key contribution of this part of work in my thesis is the non-trivial integration of (a) constrained mining, (b) mining uncertain data and (c) tree-based mining. The resulting tree-based algorithm avoids the candidate generate-and-test paradigm. It effectively pushes aggregate constraints in the mining process and explores properties of these constraints.

All the above uncertain data mining algorithms—regardless of mining all frequent itemsets or the frequent itemsets which satisfy the user-specified constraints—are based on static databases of uncertain data. It is important to note that the wide use of sensors provides us not only streams of precise data, but also streams of uncertain data. Hence, the mining of frequent itemsets from steams of uncertain data is in demand. In this thesis, we propose and develop two tree-based algorithms—called UF-streaming and SUF-growth—for mining frequent itemsets from streams of uncertain data. The key contributions of this part of work in my thesis include: (a) the proposal of effective tree structures to capture the important contents of transactions in streams of uncertain data and (b) the development of two efficient algorithms to mine frequent itemsets from the transactions captured by these proposed tree structures.

1.2 Thesis Statement

Motivated by the problems and solutions from the previous section, my **thesis statement** is as follows:

> We propose tree-based mining algorithms to fulfill the tasks of (a) finding frequent itemsets from uncertain data, (b) finding from uncertain data those frequent itemsets satisfying aggregate constraints and (c) finding frequent itemsets from streams of uncertain data.

In this thesis, we first propose a UF-growth algorithm that aims to efficiently find frequent itemsets from uncertain data, where each item in the transactions is associated with an existential probability. Since the user may be only interested in some small specific subsets of all the frequent itemsets mined from uncertain data, we then propose an ACUF-growth algorithm to find from uncertain data those frequent itemsets that satisfy user-specified aggregate constraints. Besides the above two algorithms, UF-streaming algorithm and SUF-growth algorithm have also been developed in this thesis for handling frequent itemset mining from streams of uncertain data. Table 1.1 summarizes the salient differences between our algorithms and their most relevant algorithms. Experimental results in Chapter 6 show the effectiveness of all our proposed algorithms.

Table 1.1: Our proposed algorithms vs. the most relevant algorithms

	Tree-based mining	Uncertain mining	Constrained mining	Stream mining
FP-growth [HPY00]	√			
CAP [NLHP98]			√	
FPS [LLN02]	√		√	
FP-streaming [GHP+04]	√			√
U-Apriori [CKH07]		√		
Our proposed UF-growth	√	√		
Our proposed ACUF-growth	√	√	√	
Our proposed UF-streaming	√	√		√
Our proposed SUF-growth	√	√		√

1.3 Thesis Organization

This thesis is organized as follows. The next chapter gives background and related work. In the chapter, we describe algorithms that mine frequent itemsets from traditional databases of precise data, provide background information about uncertain data, and explain algorithms for mining uncertain data. In addition, we also discuss the properties of different types of constraints and present constraint-based frequent itemset mining algorithms. Moreover, we point out differences between traditional static data and dynamic data streams, and present examine algorithms for mining frequent itemsets from streams of precise data.

We propose our frequent itemset mining algorithms in Chapters 3–5. Chapter 3 focuses on the UF-growth algorithm, which is designed to mine frequent itemsets from uncertain data. In the chapter, we discuss two major operations: the construction of UF-trees and the mining of frequent itemsets from UF-trees. The contents of this chapter have been partially published as a refereed paper in the Workshop on Data Mining of Uncertain Data held in conjunction with the Seventh IEEE International Conference on Data Mining [LCH07].

In Chapter 4, we extend the UF-growth algorithm to handle user-specified aggregate constraints. Specifically, we start with a naïve approach and two improved approaches that mine from uncertain data for those frequent itemsets satisfying user-specified aggregate constraints. To make it easier for readers to understand our approach, we present a general skeleton, which is followed by some specific details. Our algorithm, called ACUF-growth, relies on the nice properties of aggregate constraints. To reduce user burden, we classify various aggregate constraints into four classes based on the constraint properties. This chapter has been partially published as a refereed paper in the *Proceedings of the 25th ACM Symposium on Applied Computing* [LHB10].

In Chapter 5, we extend the UF-growth algorithm in another direction—namely, to mine data streams. Specifically, we propose two algorithms: UF-streaming and SUF-growth. The former is an approximate algorithm, which applies an "immediate" mode for mining "frequent" itemsets from streams of uncertain data. The latter is an exact algorithm, which uses an "delayed" mode for mining true frequent itemsets from streams of uncertain data. This chapter has been partially published as a refereed paper in the *Proceedings of 25th IEEE International Conference on Data Engineering* [LH09].

We evaluate in Chapter 6 all these algorithms, which perform uncertain frequent itemset mining, constrained frequent itemset mining from uncertain data and frequent itemset mining from streams of uncertain data by sets of experiments.

Finally, in Chapter 7, we present the conclusions of this thesis and discuss future research work such as the integration of uncertain mining, constrained mining and stream mining.

Chapter 2

Background and Related Work

In this chapter, we provide some background materials and related work that are relevant to the algorithms we propose in this thesis.

2.1 Mining Traditional Databases

Usually, data in traditional databases are precise and users are certain about the contents of these databases. To find frequent itemsets from traditional databases containing precise data, Apriori [AS94] and FP-growth [HPY00] are two popular algorithms.

2.1.1 Apriori Algorithm

In 1994, Agrawal and Srikant [AS94] proposed an algorithm, called Apriori, to find all frequent itemsets from a traditional database. An itemset X is considered frequent if its support equals or exceeds the user-specified minimum support threshold called *minsup*. Here, the support of itemset X can be described as the frequency or the number of occurrences of X in the transaction database. Given a transaction database with precise data and a user-specified minimum support threshold *minsup*, the Apriori algorithm follows a bottom-up generate-and-test framework to find all frequent itemsets. The key ideas of the algorithm can be described as follows. First, it generates set C_1 of candidate itemsets of size 1 (singleton items). Then, it counts the support of each candidate itemset in C_1 to find set L_1 containing all frequent itemsets of size 1. From set L_1, the Apriori algorithm constructs set C_2 of candidate itemsets of size 2 and determines frequent 2-item set L_2 by counting the support of each candidate itemset in C_2. This process is repeated until there are no more candidates (i.e., until C_k is empty for some $k{\geq}1$). Finally, all frequent itemsets are returned as the output of the Apriori algorithm.

2.1.2 FP-growth Algorithm

To improve the mining performance, Han et al. [HPY00] proposed the FP-growth algorithm, which does not require candidate generation. The FP-growth algorithm consists of two main operations: (a) constructing an FP-tree (which is an extended prefix-tree structure that captures the contents of transaction database containing precise data) and (b) growing frequent itemsets. To elaborate, the FP-growth algorithm first scans the database once to obtain frequent items. All infrequent items are removed and all frequent items are sorted in a decreasing frequency order so as to minimize the size of FP-tree. The algorithm then scans the database again to construct an FP-tree. Once the tree is constructed, frequent itemsets are formed by first finding the frequent itemsets of size 1 from the FP-tree and then recursively growing them. In concrete terms, for each frequent item x in the FP-tree, the FP-growth algorithm forms its projected database (i.e., a collection of transactions having $\{x\}$ as its prefix) and builds an $\{x\}$-projected tree for this projected database. Then, for each frequent item y in the $\{x\}$-projected tree, the algorithm forms a projected database for $\{x, y\}$ and builds an $\{x, y\}$-projected tree. This process is then applied recursively to each frequent item in the FP-tree for subsequent projected databases. Hence, the entire mining process can be viewed as a divide-and-conquer approach of decomposing both the mining task and the transaction database according to the frequent itemsets obtained so far. This leads to a focused search of smaller data sets. For better understanding of the structure of FP-tree and the mining approach of the FP-growth algorithm, see Example 2.1.

Example 2.1 We apply the FP-growth algorithm with a minimum support threshold *minsup* = 3 to the following transaction database to find all frequent itemsets.

Transactions	**Contents**
t_1	$\{a, b, c, d, e\}$
t_2	$\{a, b, c, d, f\}$
t_3	$\{a, b, c, e\}$
t_4	$\{a, c, d\}$

In the first step of the FP-growth algorithm, we count the support for each item in the transaction database to get a:4 (which indicates that $\{a\}$ occurs 4 times in the database), b:3, c:4, d:3, e:2, and f:1. Then, we remove infrequent items e:2 and f:1 and sort the frequent items in a decreasing frequency order: a:4, c:4, b:3, d:3. Based on these frequent items, we build a global FP-tree as shown in Figure 2.1(a). To mine frequent itemsets from this global FP-tree, we first form the projected database for

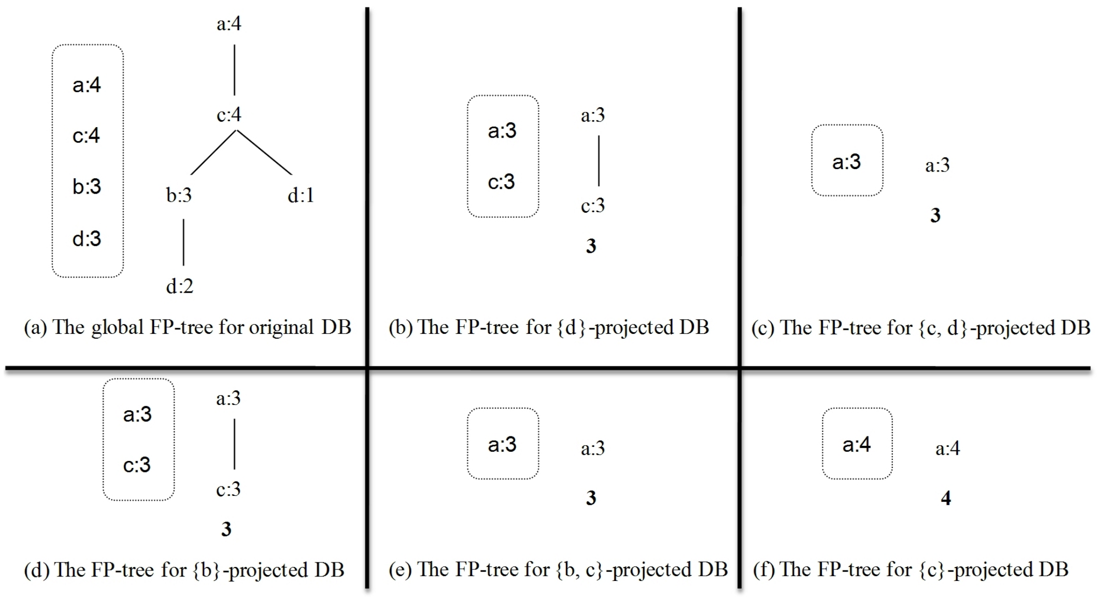

a:4
c:4
b:3
d:3
a:4
c:4
b:3
d:1
d:2
(a) The global FP-tree for original DB
a:3
c:3
a:3
c:3
3
(b) The FP-tree for {d}-projected DB
a:3
a:3
3
(c) The FP-tree for {c, d}-projected DB
a:3
c:3
a:3
c:3
3
(d) The FP-tree for {b}-projected DB
a:3
a:3
3
(e) The FP-tree for {b, c}-projected DB
a:4
a:4
4
(f) The FP-tree for {c}-projected DB

Figure 2.1: The global FP-tree and all projected FP-trees for Example 2.1.

itemset $\{d\}$ and build the $\{d\}$-projected tree as shown in Figure 2.1(b). From this tree, we find itemsets $\{c,\ d\}$:3 and $\{a,\ d\}$:3 are frequent. Furthermore, we form the projected database for itemset $\{c,\ d\}$ and build the $\{c,\ d\}$-projected tree which is shown in Figure 2.1(c). From this tree, we obtain frequent itemset $\{a,\ c,\ d\}$:3. With the same approach, we form the projected databases and build the projected trees for itemsets $\{b\}$, $\{b,\ c\}$ and $\{c\}$, which are shown in Figure 2.1(d), Figure 2.1(e), and Figure 2.1(f) respectively. Finally, we find all frequent itemsets in the transaction database. They are $\{a\}$:4, $\{c\}$:4, $\{b\}$:3, $\{d\}$:3, $\{a,\ c\}$:4, $\{a,\ b\}$:3, $\{a,\ d\}$:3, $\{c,\ b\}$:3, $\{c,\ d\}$:3, $\{a,\ c,\ b\}$:3 and $\{a,\ c,\ d\}$:3. ■

The FP-growth algorithm finds all frequent itemsets from static precise database. However, it cannot handle the situation in which data in transaction database are uncertain. As a preview, our proposed mining algorithms—like FP-growth—also apply the tree-based divide-and-conquer mining approach to find frequent itemsets without candidate generation. Nevertheless, our proposed algorithms are more powerful than the FP-growth algorithm by (a) mining frequent itemsets from uncertain data, (b) mining from uncertain data those frequent itemsets which satisfy user-specific aggregate constraints and (c) mining frequent itemsets from streams of uncertain data.

2.2 Mining Databases of Uncertain Data

Both the Apriori algorithm [AS94] and the FP-growth algorithm [HPY00] were designed to find frequent itemsets from transaction database with precise data—but not uncertain data. A key difference between precise data and uncertain data is that each transaction of the latter one contains items and additional information called existential probabilities. The existential probability $P(x, t_i)$ of an item x in a transaction t_i indicates the likelihood of x being present in t_i. Using the "possible world" interpretation of uncertain data [DYM+05, CKH07, LCH07, LMB08], there are two possible worlds for an item x and a transaction t_i: (a) the possible world W_1 where $x \in t_i$ and (b) the possible world W_2 where $x \notin t_i$. Although it is uncertain which of these two worlds is the true world, the probability of W_1 to be the true world is $P(x, t_i)$ and that of W_2 is $1 - P(x, t_i)$.

To a further extent, there are usually more than one transaction in a transaction database TDB. For instance, for an item x and a TDB consisting of two transaction t_1 and t_2, there are four possible worlds: (a) W_1 where x is in both t_1 and t_2, (b) W_2 where x is in t_1 but not t_2, (c) W_3 where x is in t_2 but not t_1, and (d) W_4 where x is neither in t_1 nor in t_2. Let $\text{prob}(W_j)$ denote the probability of W_j to be the true world. Then $prob(W_1) = P(x, t_1) \times P(x, t_2)$, $prob(W_2) = P(x, t_1) \times [1 - P(x, t_2)]$, $prob(W_3) = [1 - P(x, t_1)] \times P(x, t_2)$, and $prob(W_4) = [1 - P(x, t_1)] \times [1 - P(x, t_2)]$.

Similarly, there are usually more than one domain item in each transaction in a TDB. For instance, for two independent items x, y and a transaction t_i, there are also four possible worlds: (a) W_1 where both x, $y \in t_i$, (b) W_2 where $x \in t_i$ but $y \notin t_i$, (c) W_3 where $x \notin t_i$ but $y \in t_i$, and (d) W_4 where both x, $y \notin t_i$. Then $prob(W_1) = P(x, t_i) \times P(y, t_i)$, $prob(W_2) = P(x, t_i) \times [1 - P(y, t_i)]$, $prob(W_3) = [1 - P(x, t_i)] \times P(y, t_i)$, and $prob(W_4) = [1 - P(x, t_i)] \times [1 - P(y, t_i)]$.

To generalize, there are many items in each of the n transactions in a transaction database TDB (where $|\text{TDB}| = n$). Hence, the expected support of an itemset X in TDB can be computed by summing the support of X in possible world W_j (while taking into account the probability of W_j to be the true world) over all possible worlds:

$$expSup(X) = \sum_j [sup(X) \ in \ W_j \times prob(W_j)], \tag{2.1}$$

where $sup(X)$ denotes the support of X. The probability of W_j to be the true world, denoted by prob(W_j), can be computed as $prob(W_j) = \prod_{i=1}^{n}(\prod_{x \in t_i \ in \ W_j} P(x, t_i) \times \prod_{y \notin t_i \ in \ W_j} [1 - P(y, t_i)])$.

Note that Equation (2.1) can be simplified [DYMV05] to become the following:

$$expSup(X) = \sum_{i=1}^{n} \left(\prod_{x \in X} P(x, t_i) \right). \tag{2.2}$$

With this setting, an itemset X is considered frequent if its expected support equals or exceeds the user-specified minimum support threshold *minsup*.

To extract implicit, previously unknown, and potentially useful information from uncertain data, Chau et al. [CCKN06] proposed an algorithm, called UK-means clustering, which enhances the traditional K-means algorithm to handle uncertain data. The algorithm computes the expected squared distance for line-moving object uncertainty and free-moving object uncertainty [CXP$^+$04] separately and assigns an object to the cluster whose representative has the smallest expected distance to the object. As UK-means algorithm requires expensive integration computation to calculate the expected distance between an object and a cluster for arbitrary probability density functions, Ngai et al. [NKC$^+$06] provided several pruning methods (e.g., min-max-dist pruning) to avoid such an expensive expected distance calculation. Cormode and McGregor [CM08] also did the similar work for proposing uncertain k-center, uncertain k-means, and uncertain k-median algorithms to make all these traditional clustering algorithms handle uncertain data. Besides uncertain clustering algorithms, Aggarwal and Yu [AY08b] examined a density based approach to uncertain data outlier detection. Their approach constructs a density estimate of the underlying uncertain data in various subspaces and uses the density estimate to determine the outliers in the uncertain data. Their approach also removes some uncertainty from underlying data. Although all the above studies apply data mining algorithms to the database with uncertain data to explore useful information, they focus on the uncertain data mining tasks (clustering and outlier detection) other than frequent itemset mining.

For mining frequent itemsets from transaction database with uncertain data, Chui et al. [CKH07, CK08] proposed an Apriori-based algorithm called U-Apriori. Instead of incrementing the support counts of candidate itemsets by their *actual* support (like the Apriori algorithm), the U-Apriori algorithm increments the support counts of candidate itemsets by their *expected* support under the uncertainty model. As a result, if the expected support of a generated candidate itemset is equal or higher than the user-specified minimum support threshold *minsup*, the candidate itemset with uncertain data is frequent. On the other hand, all those candidate itemsets with expected supports less than the *minsup* can considered infrequent.

As indicated by Chui et al., their U-Apriori algorithm suffers from a few problems. First, inherited from the Apriori algorithm, U-Apriori algorithm does not scale well when handling large amounts of data because it also follows a generate-and-test framework. Second, if the existential probabilities of most items in an itemset X are small, increments for each transaction can be insignificantly small. Consequently, many candidates would not be recognized as infrequent until most (if not all) transactions were processed. To improve performance, Chui et al. applied a local trimming, global pruning and single-pass patch-up (LGS-trimming) strategy. The key idea of this strategy is that the algorithm first trims from the original database those items having expected supports less than *minsup* (i.e., trims an item x if $\mathit{expSup}\{x\} < \mathit{minsup}$) and then mines frequent itemsets from the resulting trimmed database. Since an itemset X cannot be frequent if the sum of its expected support $\mathit{expSup}(X)$ and its upper bound of an estimation error $\epsilon(X)$ falls below *minsup* (i.e., X is infrequent if $\mathit{expSup}(X) + \epsilon(X) < \mathit{minsup}$), the algorithm prunes these infrequent itemsets during the mining process. Finally, to avoid missing any frequent itemsets, the algorithm performs the patch up by scanning the original (untrimmed) database one more time to verify that those trimmed/pruned itemsets are indeed infrequent. Although the use of the LGS-trimming strategy in U-Apriori algorithm helps reduce the number of candidate itemsets being counted during the mining process, the U-Apriori algorithm still relies on candidate generate-and-test. As a preview, our proposed algorithms avoid using candidate generate-and-test.

2.3 Mining Databases with Constraints

Constrained frequent itemset mining aims to find those frequent itemsets that satisfy user-specified constraints. Regarding constrained mining, Ng et al. [NLHP98] proposed a constrained frequent itemset mining framework, in which the user can use a rich set of SQL-style constraints to guide the mining process. There is a wide range of constraints which can restrict the mining space. Examples of these constraints include the following: $C_1 \equiv max(X.Price) \leq \$100$, $C_2 \equiv min(X.Distance) \leq 60$km, $C_3 \equiv avg(X.Height) \geq 180$cm and $C_4 \equiv sum(X.Weight) \leq 50$kg. Here, constraint C_1 says that the maximum price of all items in a set X is less than or equal to \$100. Similarly, constraint C_2 says that the minimum distance between all locations in a set X is less than or equal to 60km. Constraint C_3 says that the average height of all individuals in a set X is at least 180cm. And, constraint C_4 says that the total weight of all shipping goods in a set X is lower than or equal to 50kg.

The constraints we discussed above can be categorized into several overlapping classes. For example, constraints C_1 and C_2 are **succinct constraints** because one can directly generate precisely all and only those itemsets satisfying the constraints (e.g., by using member generating function [NLHP98], which does not require generating and excluding itemsets not satisfying the constraints). All constraints C_1,

C_2, C_3 and C_4 are **aggregate constraints** because they involve aggregate functions (e.g., *max*, *min*, *avg*, and *sum*). In order to find frequent itemsets that satisfy these aggregate constraints, we explore properties (e.g., anti-monotonicity, monotonicity) that are possessed by these constraints.

- *Definition 1.* (Anti-monotonicity [NLHP98, Leu09a]). An aggregate constraint C is **anti-monotone** if and only if for any itemset X, whenever X violates C, all supersets of X also violate C.

- *Definition 2.* (Monotonicity [BMS97, Leu09b]). An aggregate constraint C is **monotone** if and only if for any itemset X, whenever X satisfies C, all supersets of X also satisfy C.

Note that constraint C_1 is an example of anti-monotone constraints. If X violates C_1 (i.e., the maximum price > \$100), all supersets of X also violate C_1 (as adding more items to itemset X will not lower the maximum price of the itemset). Conversely, if X satisfies C_1 (i.e., the maximum price $\leq$ \$100), all subsets of X also satisfy C_1 (as deleting any items from itemset X will not increase the maximum price to the itemset). Constraint C_2 is an example of monotone constraints. As a result, if X satisfies C_2 (i.e., the minimum distance $\leq$ 60km), all supersets of X also satisfy C_2 (as adding more locations to X will not increase the minimum distance). Another way to express monotone constraint C_2 is that, if X violates C_2 (i.e., the minimum distance > 60km), all subsets of X also violate C_2 (as deleting any locations from X will not lower the minimum distance). The average constraint C_3 is neither anti-monotone nor monotone because it does not possess the properties in both *Definition 1* (Anti-monotonicity) and *Definition 2* (Monotonicity). The sum constraint C_4 is an anti-monotone constraint because if X violates C_4 (i.e., the sum of weights > 50kg), all supersets of X also violate C_4. It is important to note that the sum constraint C_4 is an anti-monotone constraint if and only if items in itemset X are non-negative (e.g., shipping weight cannot be negative) or non-positive (e.g., diving distance underwater is usually expressed as non-positive number). If items in X can be both positive and negative (e.g., temperature can be both positive and negative), sum constraints may not be anti-monotone.

Since its introduction, constrained frequent itemset mining has been the subject of numerous studies. Ng et al. [NLHP98] introduced an Apriori-based algorithm called CAP for mining frequent itemsets which satisfy user-specific constraints. The CAP algorithm pushes constraints into an Apriori-based algorithm and applies candidate generate-and-test approach to obtain all valid frequent itemsets. Although the CAP algorithm achieves a high degree of pruning for constraints, it is still an Apriori-based algorithm which relies on memory intensive candidate generate-and-test.

For improving performance, Leung et al. [LLN02] proposed and designed an algorithm called FPS to mine frequent itemsets satisfying succinct constraints. The FPS algorithm avoids the generate-and-test paradigm by exploring succinctness properties of the constraints in an FP-tree based framework. To elaborate, FPS algorithm divides all succinct constraints into three subclasses: (a) succinct and anti-monotone constraints (SAM constraints), (b) succinct but not anti-monotone constraints of a form not equivalent to $S.A \supseteq CS$ where CS is a constant set for the attribute A of the set S (SUC constraints) and (c) succinct but not anti-monotone constraints of a form equivalent to $S.A \supseteq CS$ (superset constraints). For handling the SAM constraints, the FPS algorithm keeps only valid items in the initial FP-tree. All invalid items can be safely discarded due to the anti-monotone property (i.e., for any itemset X, whenever X violates C, all supersets of X also violate C). When dealing with the SUC constraints, the FPS algorithm first divides all domain items into two sets—the set $\mathbf{Item^M}$ consisting of all mandatory items and the set $\mathbf{Item^O}$ consisting of all optional items. Since a valid itemset X is composed of mandatory items (i.e., items satisfying the SUC constraint) and possibly some optional items (i.e., items by themselves not satisfying the SUC constraint), the FPS algorithm builds a *modified FP-tree*. A key difference between the original FP-tree and our modified FP-tree is that the latter divides items into two groups (mandatory items and optional items) and put them in such a way that mandatory items appear below optional items. Items within each group are then sorted in descending frequency order. In contrast, the original FP-tree treats all items as one group. So, in our modified FP-tree, the mandatory items are close to the leaves and the optional items are close to the tree root. As a result, whenever the algorithm finishes mining the projected-trees for all mandatory items, all frequent itemsets which satisfy the SUC constraint are obtained because all other frequent itemsets are guaranteed not satisfying the constraint. For better understanding how FPS algorithm handles the SUC constraints, let us see the following Example 2.2.

Example 2.2 Consider the following database:

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

with the following information for each item:

Items	**Price**
a	\$100
b	\$40
c	\$65
d	\$30
e	\$15

Let the minimum support threshold $minsup = 3$ and the constraint C_{SUC} be the SUC constraint $min(X.Price) \leq \$60$. In the first step of FPS algorithm, it checks each of the five domain items a, b, c, d and e against the constraint C_{SUC} and partitions these items into two sets: (a) the *mandatory set* containing items b, d and e (the prices associated with these items are lower than or equal to \$60) and (b) the *optional set* containing items a and c (the prices associated with these items are higher than \$60). After that, the algorithm counts the support for each item in the transaction database to get a:4, b:3, c:4, d:1 and e:3. Then, the FPS algorithm removes infrequent item d:1 and sorts the frequent items in the descending frequency order: a:4, c:4, b:3, e:3. Next, the algorithm builds a modified FP-tree (Figure 2.2(a)) for the frequent items. In this modified FP-tree, all mandatory items b and e appear below optional items a and c. Note that the mandatory item d is not in this modified UF-tree because it is infrequent (i.e., the support of item d is lower than $minsup$—3 in this example). After the global modified FP-tree is build, the FPS algorithm builds projected databases and projected-trees for all mandatory items b and e (as shown in Figure 2.2(b) and

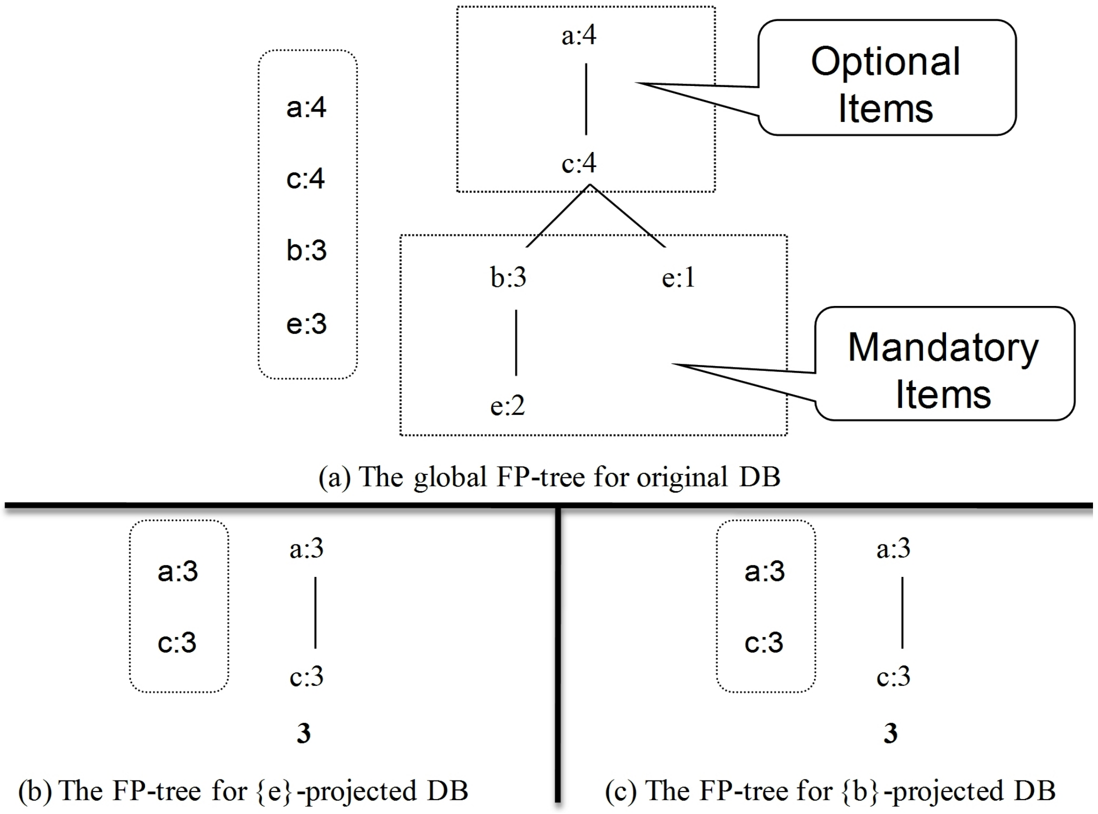

a:4
c:4
b:3
e:3
a:4
c:4
Optional Items
b:3
e:1
e:2
Mandatory Items
(a) The global FP-tree for original DB
a:3
c:3
a:3
c:3
3
(b) The FP-tree for {e}-projected DB
a:3
c:3
a:3
c:3
3
(c) The FP-tree for {b}-projected DB

Figure 2.2: The global modified FP-tree and the projected FP-trees for all mandatory items for Example 2.2.

Figure 2.2(c)) and finds all frequent itemsets which satisfy the SUC constraint C_{SUC}. They are $\{b\}$:3, $\{e\}$:3, $\{a, b\}$:3, $\{a, e\}$:3, $\{c, b\}$:3, $\{c, e\}$:3, $\{a, c, b\}$:3 and $\{a, c, e\}$:3. ■

The approach to handle superset constraints is similar to the one to handle SUC constraints. The key difference between them is the number of mandatory groups. In SUC constrained frequent itemset mining, there is only one set for mandatory items. However, the frequent itemset mining algorithm for superset constraints requires M ($M \geq 1$) sets for mandatory items. A valid itemset is composed of at least one item from each mandatory set and may or may not contain optional items.

To further enhance the functionality of constrained frequent itemset mining algorithms (i.e., FPS algorithm), Leung [Leu04] built a system, called iCFP, for interactive mining of constrained frequent itemsets. Similarly to FPS, the iCFP system also uses a tree-based mining framework which avoids the generate-and-test paradigm. In addition, iCFP system allows human users to impose a certain focus on the mining process by permitting users to dynamically change their constraints (i.e., SAM and SUC constraints) during the mining process. Two kinds of dynamic changes—tightening change and relaxing change—are applied to SAM and SUC constraints separately. The tightening change restricts the new solution space to be a subset of the old space. The result of relaxing change is that the new solution space contains the old space. Other than the above system which is interactive with the users, Leung et al. [LKH06] also designed a tree-based system for mining constrained frequent itemsets from a distributed environment. The system makes use of the constrained local frequent itemsets and the FP-trees that keep all potential global frequent items to efficiently find constrained global frequent itemsets. All constrained frequent itemset mining algorithms we discussed above were designed to find from precise databases those frequent itemsets which satisfy user-specific constraints. They do not focus on (a) aggregate constraints and (b) databases containing uncertain data. As a preview, our proposed algorithms deal with aggregate constraints and/or mine uncertain data.

2.4 Mining Streams of Data

Data streams are different from traditional static data in the following two aspects: (a) data streams are continuous as well as unbounded and (b) data in the streams are not necessarily uniformly distributed. To mime frequent itemsets from streams of precise data, Giannella et al. [GHP+04] proposed and designed a tree-based algorithm called FP-streaming. Given an incoming batch of transactions in a data stream, the first step of FP-streaming is to call the FP-growth algorithm [HPY00] with a threshold that is lower than the usual minimum support threshold *minsup* to find "frequent" itemsets. Let us call this lower threshold *preMinsup*. Then, an itemset is "frequent" if its actual support is no less than *preMinsup*. Note that, although we are interested in truly frequent itemsets (i.e., itemsets with actual support $\geq minsup > preMinsup$), FP-streaming uses *preMinsup* in attempt to avoid pruning an itemset too early. An itemset X having $preMinsup \leq \sup(X) < minsup$ is currently infrequent but may become frequent later. Once the FP-growth algorithm finds all "frequent" itemsets, the second step of FP-streaming algorithm is to store and maintain these itemsets in another tree structure called FP-stream. The key differences between an FP-tree and an FP-stream structure include the following. First, each path in an FP-tree represents a transaction, but each path in an FP-stream structure represents a "frequent" itemset. Second, each node in an FP-tree contains one support value, whereas each node in an FP-stream structure contains a natural or logarithmic tilted-time window table which contains multiple support values, one for each batch of transactions. Since users are often interested in recent data than older data, the FP-stream structure captures only a few recent batches of streaming transactions. As a new batch of transactions flows in, the window slides and the support values of each node shift as well.

Because FP-streaming algorithm uses *preMinsup* instead of exact *minsup*, it is an algorithm for approximate stream mining. To improve performance, Leung and Khan [LK06a] introduced a new tree structure called DSTree (Data Stream Tree) for mining exact frequent itemsets from data streams. In the DSTree, each tree node keeps an item with a list of frequency counts (for recording the frequency of an item in each batch in current window). Items in the tree are arranged according to some canonical order so that the ordering is unaffected by the changes in frequency caused by the continuous nature of streams. When the window slides, transactions in the oldest batch can be deleted by shifting the list of frequency counts. The DSTree also uses a pointer at each node to help shift frequency list (without traversal of the entire tree). Once the DSTree is constructed, it employs a divide-and-conquer approach (similar to the FP-growth algorithm [HPY00]) to find all frequent itemsets from data streams. To gain a better understanding of the DSTree, let us consider Example 2.3.

Example 2.3 Consider the following stream of data:

Batch	**Transactions**	**Contents**
First	t_1	$\{a, b, c, d\}$
	t_2	$\{a, b\}$
	t_3	$\{a, c\}$
Second	t_4	$\{a, b, c\}$
	t_5	$\{b, d\}$
	t_6	$\{a, b, d\}$
Third	t_7	$\{b, d\}$
	t_8	$\{a, b, d\}$
	t_9	$\{a, c\}$

Let the minimum support threshold *minsup* be 3 and let the window size w be 2 batches (indicating that only two batches of transactions are kept). When the

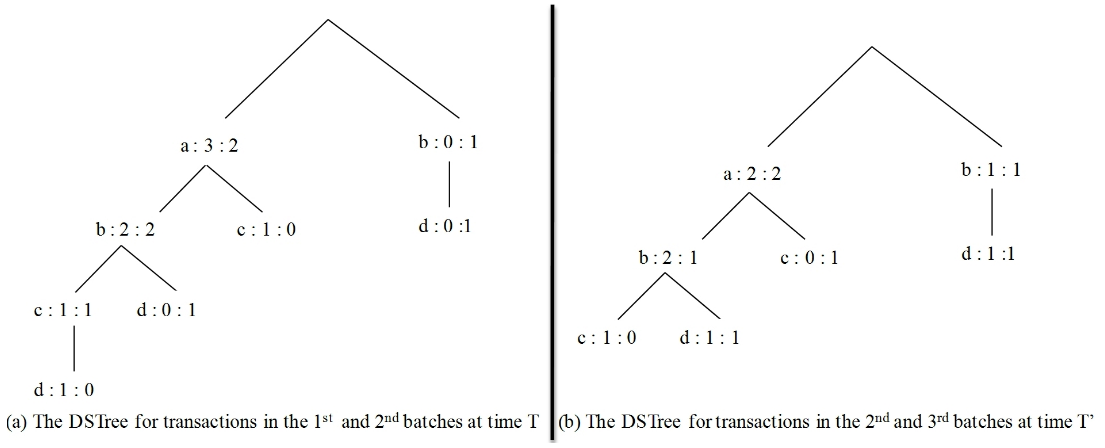

a : 3 : 2
b : 0 : 1
b : 2 : 2
c : 1 : 0
d : 0 :1
c : 1 : 1
d : 0 : 1
d : 1 : 0
(a) The DSTree for transactions in the 1st and 2nd batches at time T
a : 2 : 2
b : 1 : 1
b : 2 : 1
c : 0 : 1
d : 1 :1
c : 1 : 0
d : 1 : 1
(b) The DSTree for transactions in the 2nd and 3rd batches at time T'

Figure 2.3: The DSTree for Example 2.3 (tree nodes with frequency count (0:0) have been removed).

transactions in the first two batches in the data stream flow in (at time T), they are kept in the DSTree with a list of frequency count of 2 entries at each node (as shown in Figure 2.3(a)). For example, the node a:3:2 in Figure 2.3(a) indicates that the frequency of item a is 3 in the first batch and the frequency of item a is 2 in the second batch. When the third batch of streaming data flows in (at time T'), the new transactions in the third batch are inserted into the DSTree. The list of frequency counts is shifted and the frequency counts for the items in first batch are removed. The resulting DSTree which captures the transactions in the second and third batches at time T' is shown in Figure 2.3(b).

To find frequent itemsets at any point of time (i.e., at time T'), the FP-tree based mining process can be used to the DSTree. For instance, at time T', the FP-tree based mining process finds all frequent itemsets $\{a\}$:4, $\{b\}$:5, $\{d\}$:4, $\{a,\ b\}$:3 and $\{b,\ d\}$:4 from the DSTree which captures the transactions in the second and third batches. ■

As a DSTree may consume a large amount of memory by storing all projected databases and projected FP-trees to find frequent itemsets, Leung and Brajczuk [LB08] proposed another tree structure called DSP-tree which employs the key idea of the COFI-tree [EZ03] to find frequent itemsets from streams of precise data in a limited memory space environment. To elaborate, the DSP-tree algorithm first builds a global DSTree and uses it to store data stream information. Then, the algorithm builds a DSP-tree for each frequent item to capture the transactions in each projected database. In a DSP-tree, each tree node contains an item and a counter. Items are arranged in descending local frequency order. The value of the counter for each tree node is initially set to the frequency of the item. This value is decremented during the mining process until it reaches 0. By such a way, the DSP-tree algorithm can find all frequent itemsets from data streams without recursively constructing the projected trees for projected databases. Although all the above algorithms—FP-streaming, the algorithm that uses DSTree, and the algorithm that uses DSP-tree—can find frequent itemsets from streams of precise data, they cannot handle the situation where the data in the streams are uncertain.

2.5 Summary

In this chapter, we reviewed some related work and provided background that are relevant to the remainder of this thesis. We also pointed out the deficiencies in existing work, which motivates the current thesis work.

To mine frequent itemsets from traditional databases of precise data, the Apriori algorithm follows a bottom-up generate-and-test framework. To avoid memory intensive candidate generation, the FP-growth algorithm was proposed. It constructs an FP-tree to store database information and employs a divide-and-conquer approach to find frequent itemsets. Both the Apriori algorithm and the FP-growth algorithm are designed to find frequent itemsets from traditional precise databases. They do not handle uncertain data.

In uncertain databases, each item is associated with an existential probability. To mine frequent itemsets from databases of uncertain data, an Apriori-based algorithm, named U-Apriori, was designed. Similar to the Apriori algorithm, the U-Apriori algorithm also suffers from the memory intensive candidate generate-and-test process.

Constraints are widely used for refining searching space and enhancing user focus. For constrained frequent itemset mining, an Apriori-based algorithm, called CAP, pushes constraints deep inside the mining process and applies candidate generate-and-test approach to obtain all valid frequent itemsets. To improve performance, the FPS algorithm avoids such a generate-and-test paradigm by exploring succinctness properties of the constraints in an FP-tree based framework. Although both CAP and FPS work well with traditional databases of precise data, they cannot deal with uncertain data.

Comparing with traditional static database, data streams are continuous and unbounded. Moreover, data in the streams are not necessarily uniformly distributed. To mine frequent itemsets from data streams, a tree-based algorithm called FP-streaming was proposed and designed. As an approximate algorithm, FP-streaming uses *pre-Minsup* instead of *minsup* to avoid pruning an itemset too early. Moreover, it stores and maintains all mined "frequent" itemsets in a tree structure called FP-stream.

Since FP-streaming does not use *minsup*, it may return too many frequent itemsets by the value of *preMinsup*. To return exact frequent itemsets, the DST algorithm was proposed. However, both FP-streaming and DSTree were designed for streams of precise data, but not streams of uncertain data.

Chapter 3

Mining Frequent Itemsets from Uncertain Data

Recall that U-Apriori is an Apriori-based algorithm for mining frequent itemsets from uncertain data. It is well-known that when handling precise data, the FP-based algorithms are superior in performance to their corresponding Apriori-based counterparts. Hence, in this chapter, we investigate how tree-based frequent itemset mining can be applied to uncertain data. Specifically, we propose a tree-based algorithm, called UF-growth, for mining frequent itemsets from uncertain data. The algorithm consists of two main operations: (a) the construction of a novel tree structure called UF-tree and (b) the mining of frequent itemsets from UF-trees. We also investigate the performance and functionality of our UF-growth algorithm when compared with the U-Apriori algorithm.

3.1 The Construction of UF-trees

The key for many tree-based mining algorithms is how to represent and store data. In our case, how to represent uncertain data in each tree node and how to order these tree nodes are two main challenges. Note that precise data and uncertain data are different in many aspects. For precise data, each item in a database transaction is implicitly associated with a definite certainty of its presence in the transaction. In contrast, for uncertain data, each item is explicitly associated with an existential probability ranging from 0 (indicating that the item is not present in the database transaction) to a value of 1 (indicating that the item is definitely present). When this existential probability is close to 0, the chance of the item presented in database transaction is quite small. On the other hand, the chance of the item presented in database transaction is very high if this existential probability is close to 1. Moreover, the existential probability of an item can vary from one transaction to another. For example, the existential probability of an item a in transaction t_1 may be 0.6 (i.e., item a having a 60% likelihood to be present in t_1) and its existential probability in transaction t_2 may be 0.2 (i.e., item a having a 20% likelihood to be present in t_2). Furthermore, different items may have the same existential probability. For instance, the existential probability of item b and item c in transaction t_1 may be both at 0.3 (i.e., item b and item c both having a 30% likelihood to be present in t_1).

To perform frequent itemset mining from uncertain data, the key modification made by the U-Apriori algorithm to the Apriori algorithm is incrementing the support count of candidate itemsets by the expected support instead of the actual support. A natural question to ask is: Can we apply a similar approach? In other words, instead of storing and incrementing the actual supports of items in transactions at tree nodes (as in FP-growth), can we store and increment the expected supports for uncertain data? To answer this question, let us examine this approach. On the surface, this approach appears to work. For instance, it finds frequent itemsets of size 1 successfully. However, a close examination reveals that such an approach does not completely find all frequent itemsets. Let us elaborate a bit. Recall from Equation (2.2), $expSup(X) = \sum_{i=1}^{n} \left(\prod_{x \in X} P(x, t_i) \right)$. For any singleton itemset x (i.e., itemsets of size 1), the equation can be simplified into the following:

$$expSup(X) = \sum_{i=1}^{n} P(x, t_i). \tag{3.1}$$

This means the expected support of a singleton itemset can be computed by using the sum of expected support. So, the approach works when mining frequent itemsets of size 1 because each tree node keeps the sum of expected support of an item. However, the approach does not work when mining frequent itemsets of size greater than 1 because the expected support of any itemset is computed as the sum of *products* of the expected support of items in such an itemset. If we only store and increment the expected supports, we miss the information for the product. Therefore, to effectively represent uncertain data and find all frequent itemsets, we propose a variant of the FP-tree (i.e., a variant of an extended prefix-tree structure). We call it an **UF-tree**. Each node in the UF-tree stores (a) an item, (b) its expected support and (c) the number of occurrence of such expected support for such an item.

Now, let us turn our focus to the ordering of tree nodes in the UF-tree. From Equation (2.2), we know that the expected support of any itemset can be computed as the sum of products of the expected supports of items in the itemset. Note that the order of any portion in a product does not influence the result of the product. From this point of view, no matter how we order the tree nodes in the UF-tree, we can find the expected support of an itemset with the same efficiency. However, we observed from many other tree-based mining algorithms (i.e., FP-growth) that the ordering of tree nodes affects the tree size. Smaller trees consume less memory space than bigger ones. Bigger trees require more computation to travel the tree nodes than the smaller ones. We also know that when we handle precise data, ordering the items with descending frequency order can increase the chance of tree path sharing, which leads to reduction in the size of the tree. Applying a similar approach to our UF-tree, we order the tree nodes in descending order of accumulated expected supports to increase the chance of path sharing and reduce the size of the UF-tree.

With the above uncertain data representation and the ordering of the tree nodes, let us take a close look at how our proposed **UF-growth** algorithm constructs the UF-tree. The algorithm first scans the database once and accumulates the expected support of each item. Hence, it finds all frequent items (i.e., items having expected support $\geq$ *minsup*). It then sorts these frequent items in descending order of accumulated expected supports. After that, the algorithm scans the database the second time and inserts each transaction into the UF-tree in a similar fashion as in the construction of an FP-tree. Specifically, for each transaction, frequent items are sorted according to the descending global expected support order and infrequent items are removed. New transactions are added at the root level. At each level, nodes of the new transaction are compared with children (or descendant) nodes. If the same item and same expected support exist in both the new transaction and the children (or descendant) nodes, the transaction is merged with the node at the highest support level. The occurrence count of the merged node is then incremented by 1. The remainder of the transaction (items with their expected support) is then added to the merged node, and this process is repeated recursively until all common items and expected supports are found. Any remaining items and expected supports of the transaction are added as a new branch to the last merged node. It is important to note that only the same items with same expected support can be merged in the UF-tree. The same items with different expected supports have to be in the different branches. For instance, there is a node a:0.9:2 (i.e., a indicates the item, 0.9 indicates the expected support, and 2 indicates the count for the occurrence of the item in expected support 0.9) in the UF-tree. An incoming item a with expected support 0.9 may be merged with this tree node. However, an incoming item a with expected support 0.6 or an incoming item b with expected support 0.9 cannot be merged with it. With the above tree construction process, our UF-tree processes the following nice property:

- The occurrence count of a node is greater than or equal to the sum of occurrence counts of all its children nodes.

On the surface, the UF-tree may appear to require a large amount of memory. However, it is important to note that the UF-tree—like the FP-tree—is an extended prefix-tree structure that captures the contents of transactions. In the worst case, the number of nodes in a UF-tree is the same as the sum of items in all transactions in the original database of uncertain data. Moreover, thanks to advances in modern technology, we are able to make the same realistic assumption as in many studies [PHM00, CZ03, GHP+04] that we have enough main memory space in the sense that the trees can fit into the memory. In the situations where the trees cannot fit into memory, recursive projections and partitioning are required to break the trees into smaller pieces.

To get a better understanding about the construction of a UF-tree, let us consider the following example.

Example 3.1 Consider the following database transactions consisting of uncertain data:

Transactions	**Contents**
t_1	$\{a{:}0.9,\ b{:}0.8,\ c{:}0.7,\ d{:}0.6,\ f{:}0.8\}$
t_2	$\{a{:}0.9,\ c{:}0.7,\ d{:}0.6,\ f{:}0.1\}$
t_3	$\{b{:}0.9,\ c{:}0.5,\ e{:}0.4\}$
t_4	$\{b{:}0.9,\ e{:}0.2\}$
t_5	$\{a{:}0.9,\ c{:}0.7,\ d{:}0.6,\ e{:}0.3\}$
t_6	$\{a{:}0.3,\ b{:}0.2\}$

In this database of uncertain data, each transaction contains items and their corresponding existential probabilities. For example, there are five items a, b, c, d and f in first transaction t_1, in which the existential probabilities of these items are 0.9, 0.8, 0.7, 0.6 and 0.8 respectively.

Let the user-specified support threshold *minsup* be set to 1.0. The UF-tree can be constructed as follows. First, our UF-growth algorithm scans the database once and accumulates the expected support of each item. Hence, it finds all frequent items and sorts them in descending order of (accumulated) expected supports. Specifically, we obtain frequent items a, b, c and d (with their corresponding accumulated expected supports of 3.0, 2.8, 2.6 and 1.8), which are sorted in descending order of their expected support values. We represent these items and their expected supports as a:3.0, b:2.8, c:2.6 and d:1.8. The expected support of each of these frequent items

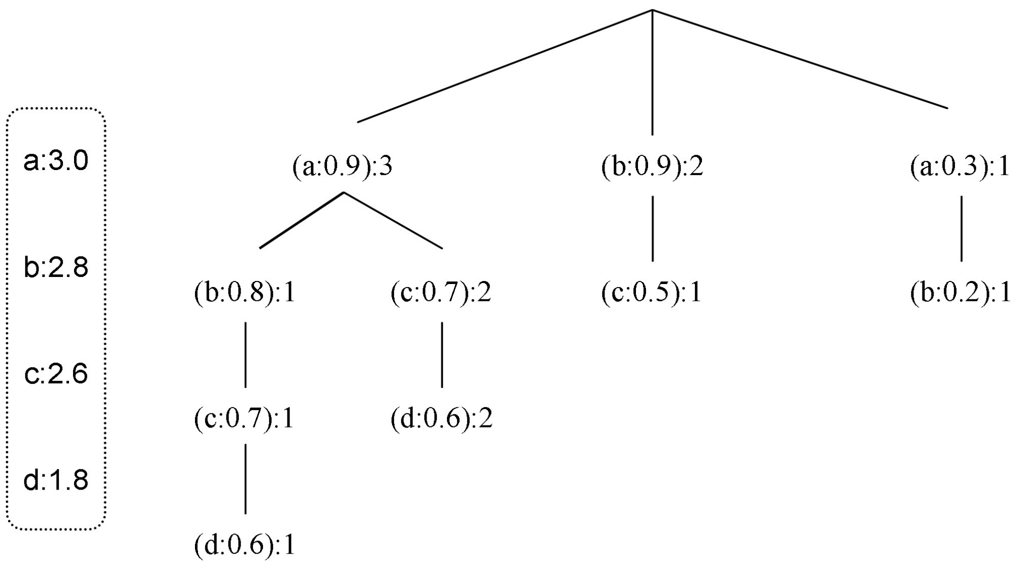


Figure 3.1: The UF-tree for Example 3.1.

$\geq$ *minsup* $= 1.0$. On the other hand, items e and f having accumulated expected support of $0.9 <$ *minsup* are removed because they are infrequent.

Then, our UF-growth algorithm scans the database the second time and inserts each transaction into the UF-tree. The algorithm first inserts the contents of the first transaction t_1 into the tree, and results in a tree branch $\langle(a{:}0.9){:}1$, $(b{:}0.8){:}1$, $(c{:}0.7){:}1$, $(d{:}0.6){:}1\rangle$. It then inserts the contents of the second transaction t_2 into the UF-tree. Since the expected support of item a in t_2 is the same as the expected support of item a in an existing branch (i.e., the branch for t_1), this node can be shared. So, the algorithm increments the occurrence count for the tree node $(a{:}0.9)$ to 2, and adds the remainder of t_2—namely, $\langle(c{:}0.7){:}1$, $(d{:}0.6){:}1\rangle$—as a child of the node $(a{:}0.9){:}2$. As a result, we get the tree branch $\langle(a{:}0.9){:}2$, $(c{:}0.7){:}1$, $(d{:}0.6){:}1\rangle$. Afterwards, the UF-growth algorithm inserts the contents of the third transaction

t_3 as a new branch $\langle(b{:}0.9){:}1, (c{:}0.5){:}1\rangle$ because the node $(b{:}0.9){:}1$ cannot be shared with the node $(a{:}0.9){:}2$. Transactions t_4, t_5 and t_6 are then inserted into the UF-tree in a similar fashion. For instance, for t_4, the node $(b{:}0.9)$ is incremented to 2; for t_5, nodes $(a{:}0.9){:}2$, $(c{:}0.7){:}1$ and $(d{:}0.6){:}1$ are all incremented by 1; for t_6, the UF-growth algorithm inserts a new branch $\langle(a{:}0.3){:}1, (b{:}0.2){:}1\rangle$ because the expected support of the item a in t_6 (equal to 0.3) is different from the expected support of the node $(a{:}0.9)$ in the tree. Consequently, at the end of the tree construction process, we get the UF-tree shown in Figure 3.1 capturing the contents of the above database with uncertain data. ■

3.2 The Mining of Frequent Itemsets from UF-trees

Once the UF-tree is constructed, our proposed UF-growth algorithm recursively mines frequent itemsets from this tree in a similar fashion as in the FP-growth algorithm except for the following:

- When forming a UF-tree for the projected database for an itemset X, we need to keep track of the existential probability of X. The existential probability of X can be calculated as the multiplying results of the existential probabilities of each items in X.
- When computing the expected support of an "extension" of an itemset X (say, $X \cup \{y\}$), we need to first multiply the existential probability of X by the existential probability of y, and then multiply this results by their occurrence

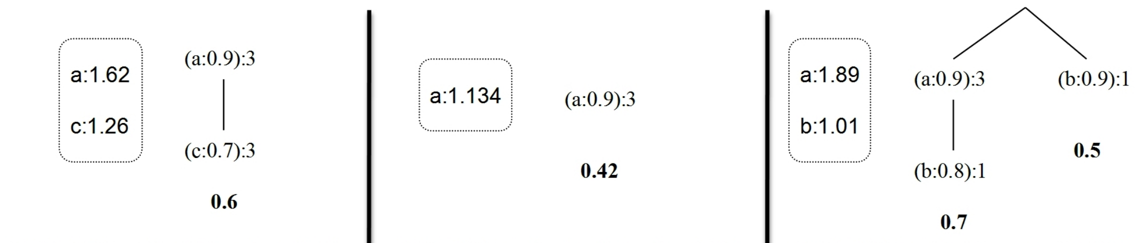
in each tree path. Finally, we sum all the multiplying results. Note that, if an itemset Z is an "extension" of an itemset X, then X is a prefix of Z. In other words, the "extension" Z can be formed by appending items to X (e.g., if $X = \{a, c\}$, then $\{a, c, d\}$, $\{a, c, e\}$ and $\{a, c, d, e\}$ are some "extensions" of X but $\{a, b, c\}$ is not).

To elaborate, the algorithm first finds the frequent itemsets of size 1 (singleton items) from the UF-tree, calculates the expected supports and keeps tracking the existential probability for each of them. For each frequent item x in the UF-tree, the UF-growth algorithm forms its projected database (i.e., a collection of transactions having $\{x\}$ as its prefix) and builds a $\{x\}$-projected tree for this projected database. For each frequent item y in the $\{x\}$-projected tree, the algorithm forms the itemset $\{x, y\}$ and calculates the expected support for this itemset. Here, the expected support for itemset $\{x, y\}$ can be calculated as follows. In each path of the $\{x\}$-projected tree, the algorithm multiplies the existential probability of item x by the existential probability of item y. Then, the algorithm multiplies this result by the occurrence of itemset $\{x, y\}$. Finally, the algorithm sums the multiplying results from all paths to obtain the expected support of itemset $\{x, y\}$. If this expected support $\geq$ *minsup*, itemset $\{x, y\}$ is frequent. In contrast, if this expected support $<$ *minsup*, itemset $\{x, y\}$ is infrequent.

Next, for each frequent item z in the $\{x, y\}$-projected tree, the algorithm calculates the expected support using the same approach we discussed above. This process is then applied recursively to each frequent item in the UF-tree for subsequent projected database. Similar to the FP-growth algorithm, the entire mining process of UF-growth algorithm can be viewed as a divide-and-conquer approach of decomposing both the mining task and the transaction database according to the frequent itemsets obtained so far. This leads to a focused search of smaller data sets. For better understanding the mining approach of UF-growth algorithm, let us see Example 3.2.

Example 3.2 Once the UF-tree for Example 3.1 is constructed, our proposed UF-growth algorithm recursively mines frequent itemsets from this tree with $minsup = 1.0$ as follows. It starts with item d (with $expSup(d)$ =1.8). The algorithm extracts from two tree paths—namely, (1) $\langle(a{:}0.9), (b{:}0.8), (c{:}0.7)\rangle$ with the occurrence count of $(d{:}0.6)$ equal to 1 (implying that items a, b, c and d occur together once in the original database) and (2) $\langle(a{:}0.9), (c{:}0.7)\rangle$ with the occurrence count of $(d{:}0.6)$ equal to 2 (implying that items a, c and d occur together twice in the original database)—and forms the d-projected database. The expected support of itemset $\{a, d\} = 0.6 \times 0.9 \times 3 = 1.62 \geq minsup$. Similarly, the expected support of itemset $\{c, d\} = 0.6 \times 0.7 \times 3 = 1.26 \geq minsup$. So, both itemsets $\{a,\ d\}$ and $\{c,\ d\}$ are frequent. However, itemset $\{b,\ d\}$ is infrequent because expected support of itemset $(\{b, d\}) = 0.6 \times 0.8 \times 1 = 0.48 < minsup$. Thus, b is removed from the $\{d\}$-projected database, which then consists of a single path $0.6\times$ $\langle(a{:}0.9){:}3, (c{:}0.7){:}3\rangle$. The UF-tree for such a projected database is shown in Figure 3.2(a).

Then, our UF-growth algorithm extracts from the UF-tree for the $\{d\}$-projected database to form the $\{c,\ d\}$-projected database, which consists of item a (representing the frequent itemset $\{a,\ c,\ d\}$) with $expSup(\{a, c, d\}) = 0.42 \times 0.9 \times 3 = 1.134$, where 0.42 represents $expSup(\{c,\ d\})$ for each of the 3 occurrences of itemset $\{c,\ d\}$. The resulting $\{c,\ d\}$-projected database and $\{c,\ d\}$-projected UF-tree are shown in



(a) The UF-tree for {d}-projected DB (b) The UF-tree for {c, d}-projected DB (c) The UF-tree for {c}-projected DB

Figure 3.2: The projected UF-trees for projected databases for Example 3.2.

Figure 3.2(b).

Next, the algorithm deals with item c. It extracts from three tree paths—namely, (1) $\langle(a{:}0.9), (b{:}0.8)\rangle{:}1$, (2) $\langle(a{:}0.9)\rangle{:}2$ and (3) $\langle(b{:}0.9)\rangle{:}1$—and forms the $\{c\}$-projected database. Note that items in the first two tree paths are both associated with the same item with the same existential probability (i.e., $c{:}0.7$), whereas items in the last path are associated with $c{:}0.5$. All this information is captured by a UF-tree shown in Figure 3.2(c). From this tree, the algorithm finds frequent itemsets $\{a,\ c\}$ and $\{b,\ c\}$, where $expSup(\{a, c\}) = 0.7 \times 0.9 \times 3 = 1.89$ and $expSup(\{b, c\}) = (0.7 \times 0.8 \times 1) + (0.5 \times 0.9 \times 1) = 1.01$. After that, our proposed UF-growth algorithm extracts from the UF-tree for the $\{c\}$-projected database to form the $\{b,\ c\}$-projected database, which consists of item a. Next, the algorithm calculates the expected support for itemset $\{a,\ b,\ c\}$ as $0.56 \times 0.9 \times 1 = 0.504 < minsup$. As a result, the itemset $\{a,\ b,\ c\}$ is infrequent.

Finally, the algorithm deals with item b. The algorithm extracts from tree paths and calculates the expected support of itemset $\{a,\ b\}$ as $expSup(\{a, b\}) = 0.8 \times 0.9 \times 1 + 0.2 \times 0.3 \times 1 = 0.78 < minsup$, which means itemset $\{a,\ b\}$ is infrequent.

To summarize, by applying our proposed UF-growth algorithm to the UF-tree (shown in Figure 3.1) that captures the contents of uncertain database in Example 3.1, we have found frequent itemsets $\{a\}$:3.0, $\{b\}$:2.8, $\{c\}$:2.6, $\{d\}$:1.8, $\{a, c\}$:1.89, $\{a, d\}$:1.62, $\{a, c, d\}$:1.134, $\{b, c\}$:1.01 and $\{c, d\}$:1.26. ■

3.3 Summary

In this chapter, we proposed the UF-growth algorithm, which is a tree-based algorithm designed to mine frequent itemsets from uncertain data. Unlike the U-Apriori algorithm relying on memory intensive candidate generate-and-test approach, our proposed UF-growth algorithm can find all frequent itemsets from uncertain data without candidate generation.

Our UF-growth algorithm consists of two major operations. First, it constructs a UF-tree, which contains all frequent items from the uncertain database has been constructed by two database scans. Each node in the UF-tree contains not only an item and its expected support, but also the number of occurrence of such expected support for such an item. In addition, we order the tree nodes in descending order of accumulated expected supports to increase the chance of path sharing.

Second, our UF-growth algorithm recursively mines frequent itemsets from the UF-tree by a divide-and-conquer approach. The algorithm first finds all frequent items of size 1 from the global UF-tree. For each frequent item of size 1, the algorithm then forms its projected database and builds its corresponding projected tree to find its "extensions" of size 2 (i.e., frequent itemsets of size 2 using such an item as prefix). This process is then applied recursively to find all frequent itemsets from the

UF-tree. Note that during the frequent itemset mining operation, we need keep track the existential probability and the occurrence count for an itemset to compute the expected support of its “extensions”.

Chapter 4

Mining Constrained Frequent Itemsets from Uncertain Data

From the previous chapter, we know that the UF-growth algorithm can find all frequent itemsets from uncertain data. However, in many real-life applications, a user may be only interested in a tiny portion of frequent itemsets mined from uncertain data. We also know that the aggregate constraints—min, max, avg and sum—are widely used for specifying user interests. A natural question to ask is: Can we find a way to mine from uncertain data for frequent itemsets that satisfy the user-specified aggregate constraints?

4.1 A Naïve Approach and Two Improved Solutions

To answer the question, a naïve approach is to find all frequent itemsets first, and then apply a post-processing step to check the frequent itemsets against the user-specified aggregate constraints. All the frequent itemsets that do not satisfy the constraints will be filtered out. Specifically, for finding the constrained frequent itemsets (i.e., frequent itemsets that satisfy the constraints) from uncertain data, the algorithm using the above approach needs to first compute the expected support of each itemset and then compare these supports with the *minsup* to find those frequent itemsets (i.e., itemsets with expected support $\geq$ *minsup*). After that, the algorithm checks each of the frequent itemsets with the user-specified aggregate constraints to find those frequent itemsets which satisfy the constraints.

One clear weakness associated with the naïve approach is that the algorithm needs to compute the expected support of each itemset, regardless whether the itemset satisfies the user-specified aggregate constraints or not. It is important to note that expected support checking is orthogonal to constraint checking. As the algorithm aims to find constrained frequent itemsets, checking the expected support for those itemsets which do not satisfy the user-specified aggregate constraints wastes time and resources. Hence, we propose an improved solution—called **CBSChecking** (indicating Constraint checking Before applying the expected Support Checking)—which performs constraint checking before the expected support checking. To elaborate, for finding the constrained frequent itemsets from uncertain data, CBSChecking first computes the expected support of each itemset. Then, the algorithm checks these itemsets with the user-specified aggregate constraints to find those itemsets which satisfy the constraints. Finally, the algorithm checks the expected supports of those constrained itemsets (i.e., itemsets that satisfy the aggregate constraints) with the *minsup* to find those frequent itemsets (i.e., itemsets with expected support $\geq$ *minsup*) which satisfy the aggregate constraints.

Following the above direction and inspired by the U-FPS algorithm which handles the frequent itemset mining with succinct constraints from uncertain data, we could push the constraint checking earlier. We can apply constraint checking before computing the expected supports for itemsets, which leads to a new solution, called **CBSComputing** (indicating Constraint checking Before expected Support Computing), with the following operations:

1. For each itemset, check if it satisfies the user-specified aggregate constraints.

2. Compute the expected supports for all valid itemsets which satisfy the aggregate constrains.

3. Check the expected supports of those valid itemsets with *minsup* to find those frequent itemsets (i.e., itemsets with expected support $\geq$ *minsup*) which satisfy the aggregate constraints.

Pushing aggregate constraint checking early could save lots of computational costs, especially when the selectivity of the constraints is low (i.e., when a small portion of frequent itemsets satisfies the constraints). However, we still need to perform constraint checking for every itemset. Can we further reduce the computational costs by exploring the properties of the user-specified aggregate constraints so that we do not need to check every itemset in order to determine whether or not it satisfies the constraints? If we can, how to do it?

4.2 General Skeleton of the ACUF-growth Algorithm

To answer the above questions, we propose a tree-based algorithm called **ACUF-growth** (indicates Aggregate Constraint UF-growth) to efficiently mine from uncertain data those frequent itemsets which satisfy the user-specified aggregate constraints. Our algorithm first scans the database with uncertain data once to accumulate the expected support of each of the domain items and finds those frequent ones. Then, the algorithm arranges these frequent items according to some order R. After that, the ACUF-growth algorithm scans the database the second time to build a modified UF-tree. Recall that the UF-tree is a tree structure to mine frequent itemsets from uncertain data. In the UF-tree, each node stores (a) an item, (b) its expected support and (c) the number of occurrences of such expected support for such an item. The order of the tree nodes in the UF-tree is based on the descending order of accumulated expected supports. In our modified UF-tree, each tree node contains exactly the same contents as the UF-tree. However, the order of the tree nodes is based on the order R, which is the only difference between a UF-tree and our modified UF-tree.

During the modified UF-tree construction process, our ACUF-growth algorithm only inserts those frequent items in each database transaction into the tree. As discussed above, the items in the tree are arranged according to some order R. Here, a new transaction is merged with a child (or descendant) node of the root of the modified UF-tree only if the same item and the same expected support exist in both the transaction and the child (or descendant) node. Note that the occurrence count of a node is at least the sum of occurrence counts of all its children nodes.

Once the modified UF-tree is constructed, our ACUF-growth algorithm recursively mines constrained frequent itemsets from this tree in a depth-first divide-and-conquer manner. Specifically, the algorithm first forms a modified UF-tree for the projected database of an item x (according to the order R) and keeps track of the expected support of x. From this tree, the algorithm recursively finds all constrained frequent itemsets containing x in a similar manner (i.e., by recursively forming a modified UF-tree for the projected database of $\{x, y\}$ where y is an item in the $\{x\}$-projected database). This process is repeated for other items. By exploring nice properties of aggregate constraints, we avoid unnecessary constraint checking and formation of projected databases.

4.3 Specific Details of the ACUF-growth Algorithm

Note that examples of aggregate constraints include max, min, avg, and sum. Here, we discuss specific details of applying our ACUF-growth algorithm on each of them. For this part, we focus on (a) the order R for arranging domain items and (b) nice properties of different types of aggregate constraints based on R.

4.3.1 Applying the ACUF-growth Algorithm on Max and Min Constraints

Max and Min Constraints with Anti-monotonicity Property

A max constraint C_{MAX} can be presented as the form $max(X.attr)\ \theta\ const$, where $attr$ is an attribute of an itemset X and $const$ is a constant. The operator θ can be $\leq$, $<$, $\geq$ or $>$. For example, constraint $max(X.Price) < \$60$ says the maximum price of all items in a set X is less than \$60 and the constraint $max(X.Weight) \geq 100$kg indicates the maximum weight of all individuals in a set X is at least 100kg.

For a max constraint C_{MAX}, let us first consider the situation in which the operator θ is $\leq$ or $<$ (say, constraint $C_{MAX_1} \equiv max(X.attr) \leq const$). In such a situation, for any itemset X, whenever X violates max constraint C_{MAX_1}, all supersets of X also violate C_{MAX_1} (as adding more items to itemset X will not lower the maximum attribute value to the itemset). In other words, if any itemset X satisfies the max constraint C_{MAX_1}, all subsets of X also satisfy C_{MAX_1} (as deleting any items from itemset X will not increase the maximum attribute value to the itemset). Based on the above properties of the max constraint C_{MAX_1}, we know that it is an anti-monotone constraint.

Hence, for the max constraint C_{MAX_1}, our proposed ACUF-growth algorithm arranges the domain items in non-ascending order R^{-} of $attr$ values (e.g., $x_i.attr \geq x_{i+1}.attr$). By such a way, our algorithm does not need to check constraint C_{MAX_1} against all the items. It stops as soon as it finds the first valid item x_i because all remaining items x_{i+j} (for all $j \geq 1$) are guaranteed to be valid (as they have lower *attr* values due to the order of R^-). Moreover, the ACUF-growth algorithm only needs to "extend" (e.g., construct modified UF-trees for projected databases) valid items x_{i+j} (for all $j \geq 0$) because any "extension" of an invalid item would have a higher *attr* value due to the order of R^-. Furthermore, no further constraint checking is required for "extensions" of any valid items. These "extensions" are guaranteed to be valid as they have the same maximum *attr* values as the valid item due to the order of R^-.

Similar to max constraints, a min constraint C_{MIN} can be presented as the form $min(X.attr)\ \theta\ const$, where *attr* is an attribute of an itemset X and *const* is a constant. The operator θ can be $\leq$, $<$, $\geq$ or $>$. For example, the constraint $min(X.Height) \geq$ 180cm indicates the minimum height of all individuals in a set X is more than or equal to 180cm. When the operator θ is $\geq$ or $>$ in a min constraint $C_{MIN_1} \equiv min(X.attr) \geq const$, the min constraint C_{MIN_1} possesses the same anti-monotonicity property as the max constraint C_{MAX_1}. As a result, for the min constraint C_{MIN_1}, when our ACUF-growth arranges the domain items in non-descending order R^+ of *attr* values, we get exactly same benefits (i.e., does not need to check constraint C_{MIN_1} against all the items) as the max constraint C_{MAX_1} to perform the frequent itemset mining.

For better understanding the procedures of our ACUF-growth algorithm handling the max constraint C_{MAX_1} and min constraint C_{MIN_1}, let us see Example 4.1.

Example 4.1 Consider the following transaction database consisting of uncertain data:

Transactions	**Contents**
t_1	$\{a$:0.8, b:0.7, c:0.8, d:1.0, e:0.5$\}$
t_2	$\{a$:0.8, b:0.7, c:0.8, e:0.9, f:0.1$\}$
t_3	$\{a$:0.7, c:0.6, d:1.0$\}$
t_4	$\{a$:0.8, b:0.7, d:1.0$\}$
t_5	$\{a$:0.7, f:0.3$\}$

with the following information for each item:

Items	**Price**
a	\$30
b	\$12
c	\$72
d	\$45
e	\$84
f	\$9

In the above database of uncertain data, each transaction contains items and their corresponding existential probabilities (e.g., in t_1, the existential probabilities of domain items a, b, c, d and e are 0.8, 0.7, 0.8, 1.0 and 0.5 respectively).

Let the user-specified threshold *minsup* be set to 1.0 and the aggregate constraint be $max(X.Price) < \$60$. To find all frequent itemsets which satisfy the aggregate constraint, our proposed ACUF-growth algorithm first scans the database once and accumulates the expected support of each domain item (a:3.8, b:2.1, c:1.4, d:3.0, e:1.4 and f:0.4). Then, the algorithm builds a global modified UF-tree consisting of only five frequent items (a, b, c, d and e). Item f is infrequent because the expected support of $f = 0.4 < minsup = 1.0$. With the aggregate constraint $max(X.Price) < \$60$, our algorithm arranges the items in non-ascending order R^- of price (i.e., e, c, d, a, b) from leaves to the root in the modified UF-tree. The resulting modified UF-tree is shown in Figure 4.1(a).

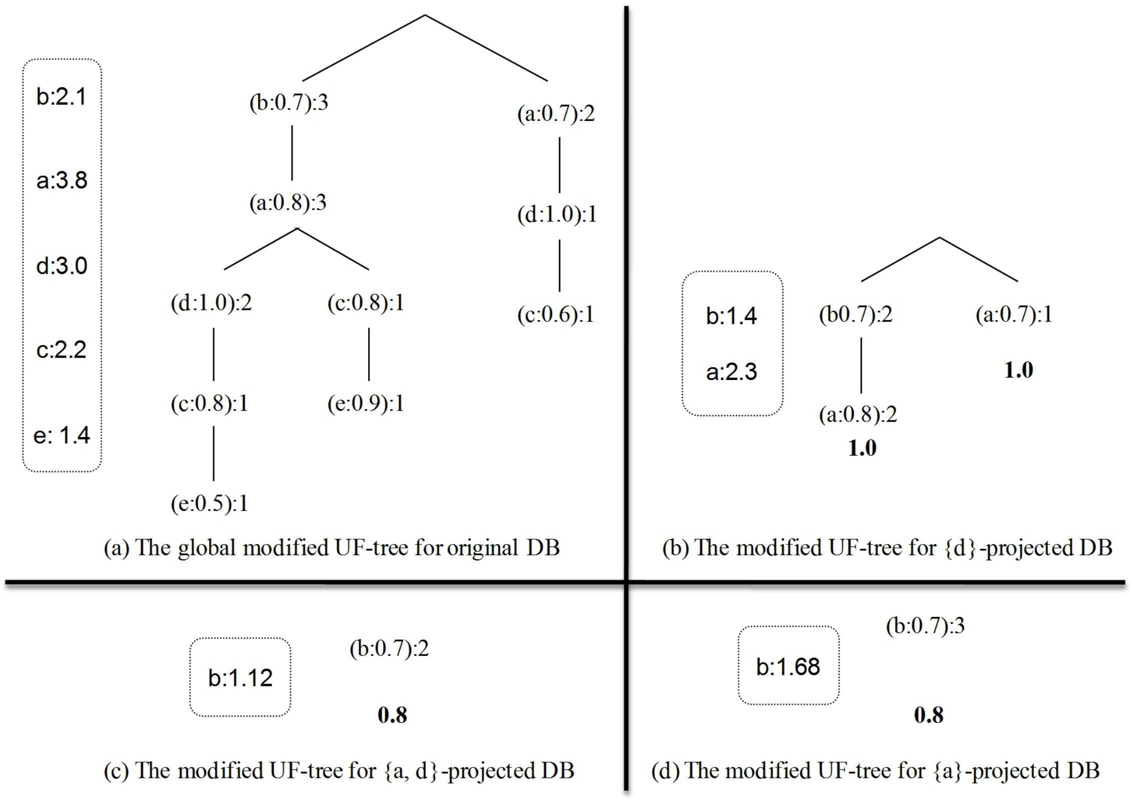


Figure 4.1: The global modified UF-tree and all projected modified UF-trees for Example 4.1.

Our ACUF-growth algorithm recursively mines frequent itemsets satisfying the max constraint $max(X.Price) < \$60$ as follows. It checks the constraint against all items in the global modified UF-tree (i.e., *e*, *c*, *d*,...) until it finds the first valid item *d*. Then, the mining process only needs to "extend" (i.e., to build the projected databases) all the valid items and no more constraint checking is required. To elaborate, for the first valid item *d*, the algorithm extracts the following tree paths:

- $\langle(b{:}0.7), (a{:}0.8)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 2, and
- $\langle(a{:}0.7)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 1.

With the above information, the ACUF-growth algorithm forms the $\{d\}$-projected database and the corresponding $\{d\}$-projected modified UF-tree as shown in Figure 4.1(b). From this projected tree, the algorithm finds valid frequent itemsets $\{a, d\}$ with expected support *expSup*($\{a, d\}$) $= (2\times0.8\times1.0)+(1\times0.7\times1.0) = 2.3$. Next, our ACUF-growth algorithm forms the projected database for itemset $\{a, d\}$ by extracting the tree path $\langle(b : 0.7)\rangle$ with the occurrence count of ($\{a, d\}$:0.8) equal to 2. The resulting $\{a, d\}$-projected tree is shown in Figure 4.1(c). From this tree, the algorithm finds the valid frequent itemset $\{a, b, d\}$ with *expSup*($\{a, b, d\}$) $= 2\times0.7\times0.8 = 1.12$.

After that, our proposed ACUF-growth algorithm finds the valid frequent itemset $\{b, d\}$ from the $\{d\}$-projected modified UF-tree. The expected support of itemset $\{b, d\}$ can be computed as *expSup*($\{b, d\}$) $= 2\times0.7\times1.0 = 1.4$. Finally, the ACUF-growth algorithm forms the $\{a\}$-projected database and constructs the corresponding $\{a\}$-projected modified UF-tree (as shown in Figure 4.1(d)) which only contains the item b with occurrence count of ($\{a\}$:0.8) equal to 3. From this tree, the algorithm finds the last frequent itemset $\{a, b\}$ with expected support *expSup*($\{a, b\}$) $= 3 \times 0.7 \times 0.8 = 1.68$. Consequently, our proposed ACUF-growth finds frequent itemsets $\{d\}$:3.0, $\{a, d\}$:2.3, $\{a, b, d\}$:1.12 $\{b, d\}$:1.4, $\{a\}$:3.8, $\{a, b\}$:1.68 and $\{b\}$:2.1 that satisfy the aggregate constraint $max(X.Price) <$ \$60. ■

Max and Min Constraints with Monotonicity Property

Now, let us discuss the situation when operator θ is $\geq$ or $>$ in a max constraint C_{MAX}. For a max constraint $C_{MAX_2} \equiv max(X.attr) \geq const$, any itemset X satisfies constraint C_{MAX_2}, all supersets of X also satisfy constraint C_{MAX_2} (as adding more items to X will not decrease the maximum attribute value of the itemset). In other words, if any itemset X violates constraint C_{MAX_2}, all subsets of itemset X also violate constraint C_{MAX_2} (as deleting any items from itemset X will not increase the maximum attribute value to the itemset). Based on the above properties of the max constraint C_{MAX_2}, we know that it is a monotone constraint.

Hence, for the max constraint C_{MAX_2}, our algorithm arranges the domain items in non-ascending order R^{-} of *attr* values (e.g., $x_i.attr \geq x_{i+1}.attr$). By such a way, our ACUF-growth algorithm keeps checking constraint C_{MAX_2} against the domain items until it finds the first invalid one x_k. All remaining items x_{k+j} are guaranteed to be invalid because they have lower *attr* values due to the order of R^{-}. Like the above procedures for C_{MAX_1}, the ACUF-growth algorithm also only "extends" (e.g., constructs modified UF-trees for projected databases) valid items x_j (for all $1 \leq j < k$). Furthermore, as the "extensions" of these valid items are guaranteed to be valid (due to the monotonicity property), no further constraint checking is required for these "extensions". Different from the constraints possessing anti-monotonicity property, the invalid items in the tree are still useful because they can be parts of the valid itemsets when our algorithm processes the monotone constraints. For example, for monotone constraint $max(X.Weight) \geq 100$kg, item a with weight equal to 120kg is a valid item. Item b with weight equal to 70kg is an invalid item. We cannot delete item b from the modified UF-tree because itemset $\{a, b\}$ is a valid itemset (maximum weight $= 120$kg ≥ 100kg). Same procedures apply to any constraint C_{MIN_2} of the form $min(X.attr) \leq const$ when items are arranged in non-descending order R^{+} of *attr* values. Let us see the following example for mining from uncertain data those frequent itemsets which satisfy aggregate constraints possessing monotonicity property.

Example 4.2 Reconsider the transaction database consisting of uncertain data in Example 4.1 with *minsup* = 1.0 and the following auxiliary information:

Items	Weight
a	85kg
b	95kg
c	80kg
d	55kg
e	50kg
f	45kg

Let the user-specific aggregate constraint be $min(X.Weight) \leq$ 60kg. To find all frequent itemsets which satisfy the above min constraint, our proposed ACUF-growth algorithm scans the database once to accumulate the expected support of each domain item (a:3.8, b:2.1, c:1.4, d:3.0, e:1.4 and f:0.4) and arranges the frequent items in non-descending order R^+ of weight (i.e., e, d, c, a, b). Then, the algorithm constructs a modified UF-tree (as shown in Figure 4.2(a)) by inserting these frequent items as the non-descending order R^+ from tree leaves to tree root.

To mine frequent itemsets from the modified UF-tree, our ACUF-growth algorithm first checks the constraint $min(X.Price) \leq$ 60kg against all items in the global modified UF-tree (i.e., e, d, c, ...) until it finds the first invalid item c. Then, the algorithm only "extends" (builds the projected databases) those valid items (i.e., item e and item d). No more constraint checking is required because of the monotonicity property and the non-descending order R^+ of weight values.

For the first valid item e, the algorithm extracts the following tree paths:

- $\langle(a{:}0.8), (c{:}0.8)\rangle$ with the occurrence count of (e:0.5) equal to 1, and
- $\langle(a{:}0.8), (c{:}0.8)\rangle$ with the occurrence count of (e:0.9) equal to 1.

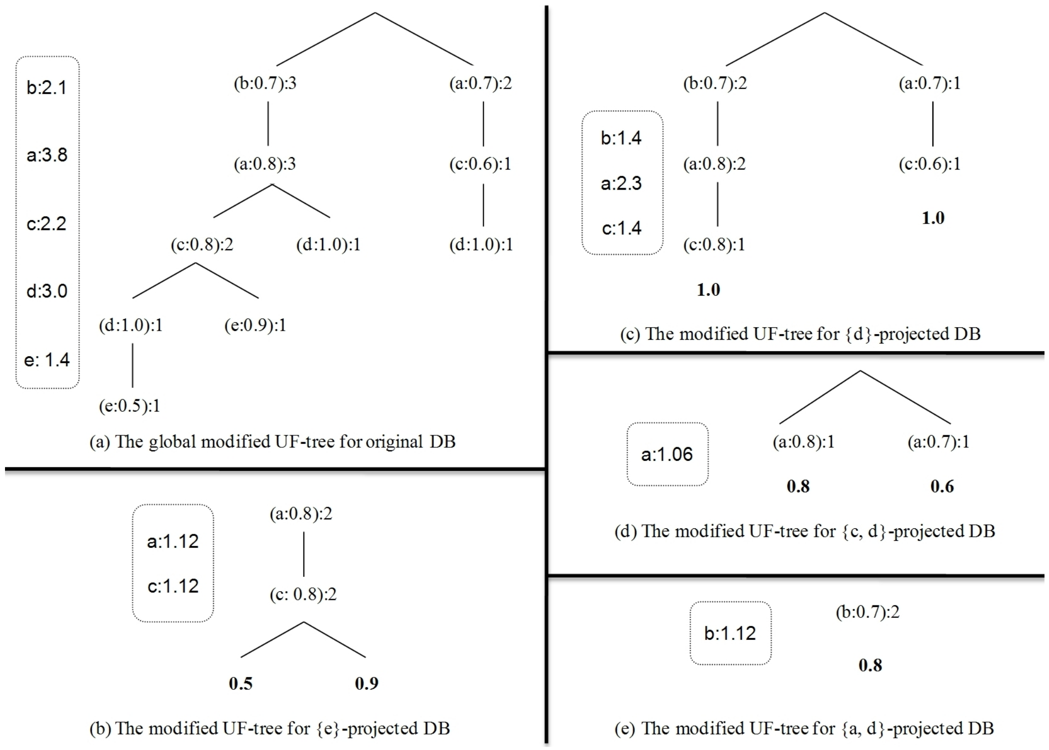


Figure 4.2: The global modified UF-tree and all projected modified UF-trees for Example 4.2.

The ACUF-growth algorithm forms the $\{e\}$-projected database and constructs the $\{e\}$-projected modified UF-tree as shown in Figure 4.2(b). From this tree, the algorithm finds the frequent itemsets $\{a,\ e\}$ and $\{c,\ e\}$ which satisfy the min constraint $min(X.Price) \leq 60$kg. The expected support for itemset $\{a,\ e\}$ can be computed as $expSup(\{a, e\}) = (1 \times 0.8 \times 0.5) + (1 \times 0.8 \times 0.9) = 1.12$ and the expected support for itemset $\{c,\ e\}$ can be computed as $expSup(\{c, e\}) = ((1 \times 0.8 \times 0.5) + (1 \times 0.8 \times 0.9) = 1.12$.

Next, our proposed algorithm extracts from the following tree paths to form $\{d\}$-projected database and construct the corresponding $\{d\}$-projected modified UF-tree as shown in Figure 4.2(c):

- $\langle(b{:}0.7), (a{:}0.8), (c{:}0.8)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 1,
- $\langle(b{:}0.7), (a{:}0.8)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 1, and
- $\langle(a{:}0.7), (c{:}0.6)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 1.

From this projected tree, the algorithm first finds valid frequent itemset $\{c, d\}$ with $expSup(\{c, d\}) = ((1 \times 0.8 \times 1.0) + (1 \times 0.6 \times 1.0) = 1.4$. Then, it builds the $\{c, d\}$-projected modified UF-tree for the $\{c, d\}$-projected database as shown in Figure 4.2(d). From this $\{c, d\}$-projected tree, our ACUF-growth algorithm finds valid frequent itemset $\{a, c, d\}$ with $expSup(\{a, c, d\}) = ((1 \times 0.8 \times 0.8) + (1 \times 0.7 \times 0.6) = 1.06$.

From the $\{d\}$-projected modified UF-tree (as shown in Figure 4.2(c)), the algorithm also finds frequent itemset $\{a, d\}$ with $expSup(\{a, d\}) = ((2 \times 0.8 \times 1.0) + (1 \times 0.7 \times 1.0) = 2.3$. The algorithm then "extends" valid item $\{d\}$ to itemset $\{a, d\}$ and constructs $\{a, d\}$-projected modified UF-tree as shown in Figure 4.2(e). From this $\{a, d\}$-projected tree, our algorithm finds the valid frequent itemset $\{a, b, d\}$ with $expSup(\{a, b, d\}) = 2 \times 0.7 \times 0.8 = 1.12$.

The last frequent itemset which satisfies constraint $min(X.Price) \leq 60$kg can be obtained from $\{d\}$-projected modified UF-tree. It is the itemset $\{b, d\}$ with expected support $expSup(\{b, d\}) = 2 \times 0.7 \times 1.0 = 1.4$. Hence, our proposed ACUF-growth finds all valid (i.e., satisfy the aggregate constraint $min(X.Price) \leq 60$kg) frequent itemsets as $\{e\}$:1.4, $\{c, e\}$:1.12, $\{a, e\}$:1.12, $\{d\}$:3.0, $\{c, d\}$:1.4, $\{a, c, d\}$:1.06, $\{a, d\}$:2.3, $\{a, b, d\}$:1.12 and $\{b, d\}$:1.4. ■

4.3.2 Applying the ACUF-growth Algorithm on Average Constraint

Now, let us discuss the average constraint C_{AVG}. Similar to C_{MAX} and C_{MIN}, C_{AVG} can be presented as the form $avg(X.attr)\ \theta\ const$ where $attr$ is an attribute of an itemset X and $const$ is a constant. The operator θ can be $\leq$, $<$, $\geq$ or $>$. For example, constraint $avg(X.Temperature) \geq -10$°C says the average temperature of all locations in a set X is more than or equal to -10°C and the constraint $avg(X.Size)$ $< 200\text{m}^2$ indicates the average size of all houses in a set X is less than 200 square meters.

For an average constraint C_{AVG}, let us first consider the situation in which the operator θ is $\geq$ or $>$. Normally, with any arbitrary item order, constraint $C_{AVG_1} \equiv$ $avg(X.attr) \geq const$ is neither anti-monotone nor monotone. However, if we arrange domain items in non-ascending order R^- of $attr$ values (e.g., $x_i.attr \geq x_{i+1}.attr$), constraint C_{AVG_1} would possess a nice property (i.e., convertible anti-monotonicity property) where all "extensions" of an itemset X would violate C_{AVG_1} whenever X violates C_{AVG_1}. With such a property, our ACUF-growth algorithm keeps checking average constraint C_{AVG_1} against the domain items until it finds the first invalid one x_k because all remaining items x_{k+j} (for all $j \geq 1$) are guaranteed to be invalid. Again, the ACUF-growth algorithm only needs to "extend" (e.g., form modified UF-trees for projected databases) valid items x_j (for all $1 \leq j < k$) because any "extension" of an invalid item would have a lower average $attr$ value due to the order of R^-. However, further constraint checking is required in subsequent projected databases of the "extensions" of these valid items because these "extensions" may be invalid.

When the operator θ is $\leq$ or $<$ in an average constraint (e.g., constraint $C_{AVG_2} \equiv avg(X.attr) \leq const$), the constraint C_{AVG_2} is also neither anti-monotone nor monotone. However, if we arrange domain items in non-descending order R^+ of *attr* values (e.g., $x_i.attr \leq x_{i+1}.attr$), constraint C_{AVG_2} would possess exactly the same convertible anti-monotonicity property as constraint C_{AVG_1} (i.e., all "extensions" of an itemset X would violate C_{AVG_2} whenever X violates C_{AVG_2}). As a result, our proposed ACUF-growth algorithm performs the same procedures (as the procedures for average constraint C_{AVG_1}) to the average constraint C_{AVG_2} to find from uncertain data those frequent itemsets which satisfy C_{AVG_2}.

For better understanding how our proposed ACUF-growth algorithm handles average constraints, let us see Example 4.3.

Example 4.3 Consider the following transaction database consisting of uncertain data:

Transactions	**Contents**
t_1	$\{a{:}0.8,\ b{:}0.7,\ c{:}0.8,\ d{:}1.0,\ e{:}0.2\}$
t_2	$\{a{:}0.8,\ b{:}0.7,\ c{:}0.8,\ e{:}0.3,\ f{:}0.1\}$
t_3	$\{a{:}0.7,\ c{:}0.6,\ d{:}0.9,\ e{:}0.4\}$
t_4	$\{a{:}0.8,\ b{:}0.7,\ d{:}1.0,\ f{:}0.1\}$

with the following auxiliary information for each item:

Items	**Temperature**
a	–20°C
b	–35°C
c	+5°C
d	–5°C
e	–15°C
f	0°C

In the above database of uncertain data, each transaction contains items and their corresponding existential probabilities (e.g., in t_1, the existential probabilities of domain items a, b, c, d and e are 0.8, 0.7, 0.8, 1.0 and 0.2 respectively).

By setting the user-specified threshold *minsup* be 1.0 and average constraint be $avg(X.Temperature) \geq –10°C$, our ACUF-growth algorithm scans the database once, accumulates the expected support of each domain item (a:3.1, b:2.1, c:2.2, d:2.9, e:0.9 and f:0.2), and builds a global modified UF-tree (see Figure 4.3(a)) consisting of only four frequent items a, b, c and d. With average constraint $avg(X.Temperature) \geq -10°C$, our algorithm arranges the items in non-ascending order R^- of temperature values (i.e., c, d, a, b) from leaves to root in the modified UF-tree.

Once the global modified UF-tree has been constructed, our ACUF-growth algorithm recursively mines frequent itemsets which satisfy the average constraint $avg(X.Temperature) \geq –10°C$ from the tree as follows. It first checks the average constraint against all items in the global modified UF-tree (i.e., c, d, a, ...) until it finds the first invalid item a. Then, the algorithm "extends" (builds the projected databases) only valid items. When "extending" valid item c, the algorithm extracts the following tree paths:

- $\langle(d{:}1.0), (a{:}0.8), (b{:}0.7)\rangle$ with the occurrence count of $(c{:}0.8)$ equal to 1,
- $\langle(a{:}0.8), (b{:}0.7)\rangle$ with the occurrence count of $(c{:}0.8)$ equal to 1, and
- $\langle(d{:}0.9), (a{:}0.7)\rangle$ with the occurrence count of $(c{:}0.6)$ equal to 1.

In addition, ACUF-growth also constructs a modified UF-tree for the $\{c\}$-projected database as shown in Figure 4.3(b). The algorithm then checks the average constraint $avg(X.Temperature) \geq –10°C$ against itemsets $\{c, d\}$, which is valid with an expected support $expSup(\{c, d\}) = (1 \times 1.0 \times 0.8) + (1 \times 0.9 \times 0.6) = 1.34$. After "extending" valid frequent itemset $\{c, d\}$, the algorithm constructs the $\{c, d\}$-projected database

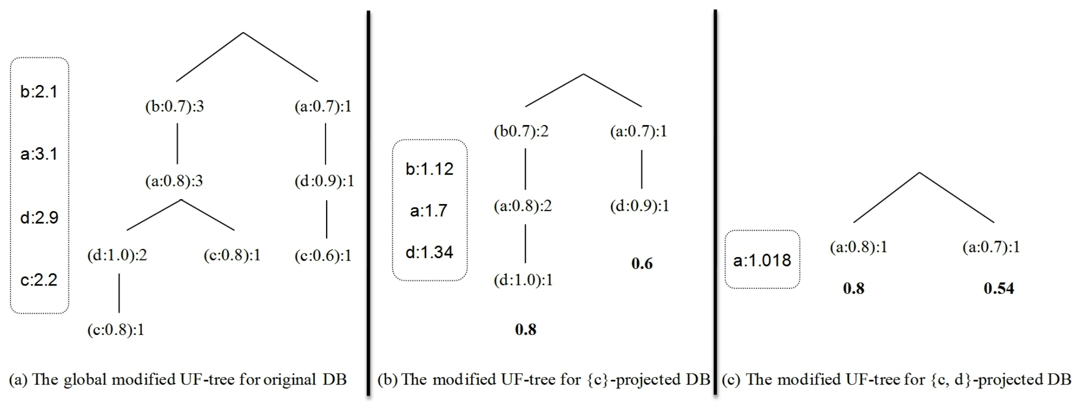


Figure 4.3: The global modified UF-tree and all projected modified UF-trees for Example 4.3.

and its corresponding $\{c, d\}$-projected modified UF-tree as shown in Figure 4.3(c). The ACUF-growth algorithm then applies constraint checking and finds valid frequent $\{c, d, a\}$ with expected support $expSup(\{c, d, a\}) = (1 \times 0.8 \times 0.8) + (1 \times 0.7 \times 0.54) = 1.018$ from $\{c, d\}$-projected tree.

Next, our ACUF-growth algorithm finds the frequent itemset $\{c, a\}$ from the $\{c\}$-projected modified UF-tree with expected support $expSup(\{c, a\}) = (2 \times 0.8 \times 0.8) + (1 \times 0.7 \times 0.6) = 1.7$. After checking itemset $\{c, a\}$ against the average constraint $avg(X.Temperature) \geq -10°\text{C}$, we know that it is another valid frequent itemset. However, the "extension" of itemset $\{c, a\}$—itemset $\{c, a, b\}$—is invalid. Then, the algorithm checks the average constraint against the next item b in $\{c\}$-projected modified UF-tree (Figure 4.3(b)) and finds itemset $\{c, b\}$ is invalid. No further constraint checking is required to any other itemsets in the {c}-projected database because $-10°\text{C} > avg(\{c, b\}.Temperature) > avg(\{c, x\}.Temperature)$ for any item x ordered after item b according to the non-ascending order R^{-}.

Afterwards, the algorithm checks the average constraint $avg(X.Temperature) \geq$ –10°C against item d in the global modified UF-tree in Figure 4.3(a) and "extends" it. However, none of its "extensions" is valid. Then, the ACUF-growth algorithm checks item a in the global modified UF-tree against the average constraint and finds item a is invalid. No further constraint checking is applied to other items in the global modified UF-tree. Consequently, our proposed algorithm finds frequent itemsets $\{c\}$:2.2, $\{c,\ d\}$:1.34, $\{c,\ d,\ a\}$:1.018, $\{c,\ a\}$:1.7 and $\{d\}$:2.9 that satisfy aggregate average constraint $avg(X.Temperature) \geq$ –10°C. ■

4.3.3 Applying the ACUF-growth Algorithm on Sum Constraint

So far, we have discussed how our proposed ACUF-growth algorithm finds from uncertain data those frequent itemsets which satisfy user-specific max, min and average constraints. Now, let us turn our focus to sum constraint C_{SUM}. Similar to other constraints, the sum constraint C_{SUM} can be presented as the form $sum(X.attr)\ \theta$ *const*, where *attr* is an attribute of an itemset X and *const* is a constant. The operator θ can be $\leq$, $<$, $\geq$ or $>$. For example, constraint $sum(X.Rainfall) \geq$ 100mm says the total rainfall of all locations in a set X is more than or equal to 100mm and the constraint $sum(X.Diving) \geq$ –200m indicates the total diving distance of an individual in all locations in a set X is more than or equal to 200 meters (under water 200m).

As we know, no matter what the attributes of domain items have positive, negative or mix (of positive and negative) values, the max, min and average constraints possess nice properties (i.e., anti-monotonicity, monotonicity, and convertible anti-monotonicity) as long as we form the items with some specific orders (i.e., non-descending order R^+ or non-ascending order R^- of the attribute values). However, the property of the sum constraint is related to the values of the domain item attributes. Specifically, the constraint C_{SUM} possesses nice property if and only if the values of domain item attributes are all positive or all negative. In other words, with any arbitrary item order, constraint C_{SUM} is not anti-monotone, monotone, convertible anti-monotone or convertible monotone if the values of domain item attributes are mixed with both positive and negative numbers. As a result, in this section, we only consider the situations for the sum constraint in which the values of domain item attributes are all positive or all negative.

Firstly, we consider a sum constraint C_{SUM_1} of the form $sum(X^+.attr) \geq const$ (where X^+ is an itemset with all positive *attr* values). If we arrange domain items in non-ascending order R^- of *attr* values (e.g., $x_i.attr \geq x_{i+1}.attr$), sum constraint C_{SUM_1} would possess convertible monotonicity property. In such a situation, all "extensions" of an itemset X would satisfy sum constraint C_{SUM_1} whenever itemset X satisfies C_{SUM_1}. Here, our proposed ACUF-growth algorithm keeps performing constraint checking until it finds the first invalid item x_k because all remaining items x_{k+j} (for all $j \geq 1$) are guaranteed to be invalid (as they have lower positive *attr* values). Unlike the procedures for all max, min and average constraints, the ACUF-growth algorithm "extends" (e.g., forms modified UF-trees for projected databases) valid items as well as invalid items. While further constraint checking is unnecessary for "extensions" of the valid items (because these "extensions" are guaranteed to be valid due to the convertible monotonicity property), further constraint checking is needed for "extensions" of the invalid items (because some of these"extensions" may be valid). Same procedures apply to a sum constraint C_{SUM_2} of the form $sum(X^{-}.attr) \leq const$ (where X^{-} is an itemset with all negative *attr* values) when items are arranged in non-descending order R^{+} of *attr* values.

Now, let us consider a sum constraint C_{SUM_3} of the form $sum(X^{-}.attr) \geq const.$ If we arrange domain items in non-ascending order R^{-} of *attr* values (e.g., $x_i.attr \geq x_{i+1}.attr$), constraint C_{SUM_3} would possess convertible anti-monotonicity property that all "extensions" of an itemset X would violate the sum constraint C_{SUM_3} whenever X violates C_{SUM_3}. With such a property, our ACUF-growth algorithm keeps checking sum constraint C_{SUM_3} against the domain items until it finds the first invalid one x_k because all remaining items x_{k+j} (for all $j \geq 1$) are guaranteed to be invalid. In this situation, the ACUF-growth algorithm only needs to "extend" (e.g., form modified UF-trees for projected databases) valid items x_j (for all $1 \leq j < k$) because any "extensions" of an invalid item would have a lower sum *attr* value due to the negative values of all attributes. However, further constraint checking is required in subsequent projected databases of the "extensions" of these valid items because these "extensions" may violate the sum constraint C_{SUM_3}. For a sum constraint C_{SUM_4} of the form $sum(X^{+}.attr) \leq const$, we can perform the same procedures (as the procedures for C_{SUM_3}) when items are arranged in non-descending order R^{+} of *attr* values because it also possesses convertible anti-monotonicity property.

For better understanding the procedures of our proposed ACUF-growth algorithm handling the sum constraints in all positive domain item attributes and all negative domain item attributes, let us see the following two examples.

Example 4.4 Reconsider the transaction database in Example 4.3 with *minsup* = 1.0 and the following auxiliary information.

Items	**Rainfall**
a	60mm
b	50mm
c	100mm
d	105mm
e	110mm
f	115mm

Let the user-specific aggregate constraint be *sum*(*X.Rainfall*) $\geq$ 100mm. Here, all the *attr* values (i.e., the rainfall) are positive. To find all frequent itemsets which satisfy the above sum constraint, our proposed ACUF-growth algorithm scans the database once to accumulate the expected support of each domain item a:3.1, b:2.1, c:2.2, d:2.9, e:0.9 and f:0.2. Then, the algorithm builds a global modified UF-tree (see Figure 4.4(a)) consisting of only four frequent items a, b, c and d. With sum constraint *sum*(*X.Rainfall*) $\geq$ 100mm, our algorithm arranges the frequent items in descending order R^- of rainfall values (i.e., d, c, a, b) from leaves to root in the modified UF-tree.

Our ACUF-growth algorithm recursively mines frequent itemsets satisfying the sum constraint *sum*(*X.Rainfall*) $\geq$ 100mm as follows. It checks the sum constraint against all items in the global modified UF-tree (i.e., d, c, a, b) until it finds the first invalid item a. The remaining item (i.e., item b in this example) is guaranteed to be invalid due to the non-ascending order R^-. Then, the algorithm "extends" (builds the projected databases) for all valid and invalid items. When "extending" valid item d, the algorithm extracts the following tree paths:

- $\langle (c{:}0.8), (a{:}0.8), (b{:}0.7)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 1,
- $\langle (a{:}0.8), (b{:}0.7)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 1, and
- $\langle (c{:}0.6), (a{:}0.7)\rangle$ with the occurrence count of $(d{:}0.9)$ equal to 1.

From the information obtained above, the algorithm constructs a modified UF-tree for the $\{d\}$-projected database as shown in Figure 4.4(b).

Note that for the sum constraint$sum(X.Rainfall) \geq 100$mm, further constraint checking for valid items (i.e., item d and item c) and their "extensions" is unnecessary because of the convertible monotonicity property. Hence, our proposed ACUF-growth algorithm "extends" valid itemset $\{d, c\}$ and calculates its expected support as $expSup(\{d, c\}) = (1 \times 0.8 \times 1.0) + (1 \times 0.6 \times 0.9) = 1.34$. After that, the algorithm constructs the $\{d, c\}$-projected database and its corresponding $\{d, c\}$-projected modified UF-tree as shown in Figure 4.4(c). The ACUF-growth algorithm then finds valid frequent $\{d, c, a\}$ with expected support $expSup(\{d, c, a\}) =(1 \times 0.8 \times 0.8) + (1 \times 0.7 \times 0.54) = 1.018$ from $\{d, c\}$-projected tree.

Next, our ACUF-growth algorithm "extends" valid itemset $\{d, a\}$ and calculates its expected support as $expSup(\{d, a\}) = (2 \times 0.8 \times 1.0) + (1 \times 0.7 \times 0.9) = 2.23$. After that, the algorithm constructs the $\{d, a\}$-projected database and its corresponding $\{d, a\}$-projected modified UF-tree which only contains one item b (as shown in Figure 4.4(d)). Our proposed ACUF-growth algorithm then finds valid frequent $\{b, a, d\}$ with expected support $expSup(\{b, a, d\}) = 2 \times 0.7 \times 0.8 = 1.12$ from $\{d, a\}$-projected tree.

Then, the algorithm extracts the following tree paths to form a modified UF-tree for the $\{c\}$-projected database as shown in Figure 4.4(e):

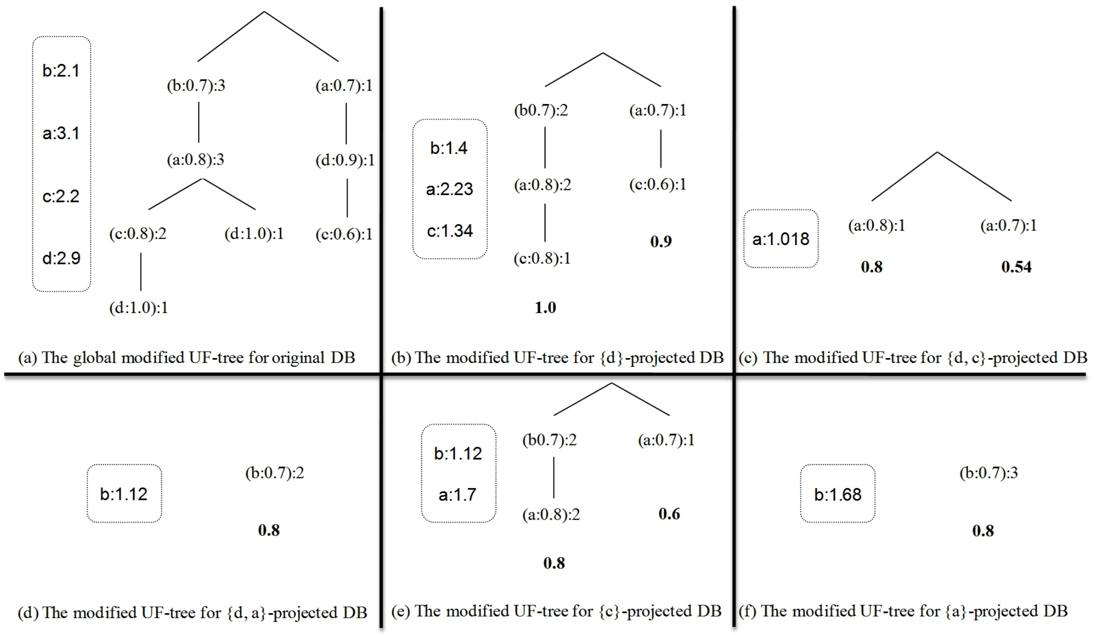

b:2.1
a:3.1
c:2.2
d:2.9
(b:0.7):3
(a:0.7):1
(a:0.8):3
(d:0.9):1
(c:0.8):2
(d:1.0):1
(c:0.6):1
(d:1.0):1
(a) The global modified UF-tree for original DB
b:1.4
a:2.23
c:1.34
(b0.7):2
(a:0.7):1
(a:0.8):2
(c:0.6):1
(c:0.8):1
0.9
1.0
(b) The modified UF-tree for {d}-projected DB
a:1.018
(a:0.8):1
(a:0.7):1
0.8
0.54
(c) The modified UF-tree for {d, c}-projected DB
b:1.12
(b:0.7):2
0.8
(d) The modified UF-tree for {d, a}-projected DB
b:1.12
a:1.7
(b0.7):2
(a:0.7):1
(a:0.8):2
0.6
0.8
(e) The modified UF-tree for {c}-projected DB
b:1.68
(b:0.7):3
0.8
(f) The modified UF-tree for {a}-projected DB

Figure 4.4: The global modified UF-tree and all projected modified UF-trees for Example 4.4.

- $\langle(a{:}0.8), (b{:}0.7)\rangle$ with the occurrence count of $(c{:}0.8)$ equal to 2, and
- $\langle(a{:}0.7)\rangle$ with the occurrence count of $(c{:}0.6)$ equal to 1.

From this $\{c\}$-projected modified UF-tree, our algorithm finds the valid itemset $\{c, a\}$ and calculates its expected support as $expSup(\{c, a\}) = (2 \times 0.8 \times 0.8) + (1 \times 0.7 \times 0.6) = 1.7$. Note that the itemset $\{c, a, b\}$ is valid with the sum constraint $sum(X.Rainfall) \geq 100$mm. However, it is infrequent because its expected support equals to $2 \times 0.7 \times 0.64 = 0.896$ which is lower than the user-specific $minsup = 1.0$. From $\{c\}$-projected modified UF-tree, our algorithm also "extends" valid frequent itemsets $\{c, b\}$ with expected support $expSup(\{c, b\}) = 2 \times 0.7 \times 0.8 = 1.12$.

Afterwards, the algorithm constructs a modified UF-tree for the $\{a\}$-projected database (as shown in Figure 4.4(f)) by extracting from tree path $\langle(b{:}0.7)\rangle$ with the occurrence count of $(a{:}0.8)$ equal to 2. Since item a is invalid for the sum constraint $sum(X.Rainfall) \geq 100$mm, the algorithm checks the sum constraint against its "extension" itemset $\{a, b\}$ and calculates its expected support as $expSup(\{a, b\}) = 3 \times 0.7 \times 0.8 = 1.68$. Hence, our proposed algorithm finds frequent itemsets $\{d\}$:2.9, $\{d, c\}$:1.34, $\{d, c, a\}$:1.018, $\{d, a\}$:2.23, $\{d, a, b\}$:1.12, $\{c\}$:2.2, $\{c, a\}$:1.7, $\{c, b\}$:1.12 and $\{a, b\}$:1.68 that satisfy aggregate sum constraint $sum(X.Rainfall) \geq 100$mm. ■

Example 4.5 Reconsider the transaction database in Example 4.3 with the following auxiliary information:

Items	**Diving distance (underwater)**
a	–260m
b	–230m
c	–60m
d	–120m
e	–100m
f	–360m

Let the user-specific minimum support threshold $minsup = 1.0$ and aggregate sum constraint be $sum(X.Diving) \geq -200$m. All the *attr* values in this example are negative. To find all frequent itemsets which satisfy the above sum constraint, our proposed ACUF-growth algorithm scans the database once to accumulate the expected support of each domain item a:3.1, b:2.1, c:2.2, d:2.9, e:0.9 and f:0.2. Among them, items a, b, c and d are frequent. Then, the algorithm builds a global modified UF-tree (as shown in Figure 4.5(a)) consisting of only the frequent items. With sum constraint $sum(X.Diving) \geq -200$m, our algorithm arranges the items in non-ascending order R^{-} of diving distance values (i.e., c, d, b, a) from leaves to root in the modified UF-tree.

In the next step, our proposed ACUF-growth algorithm recursively mines frequent

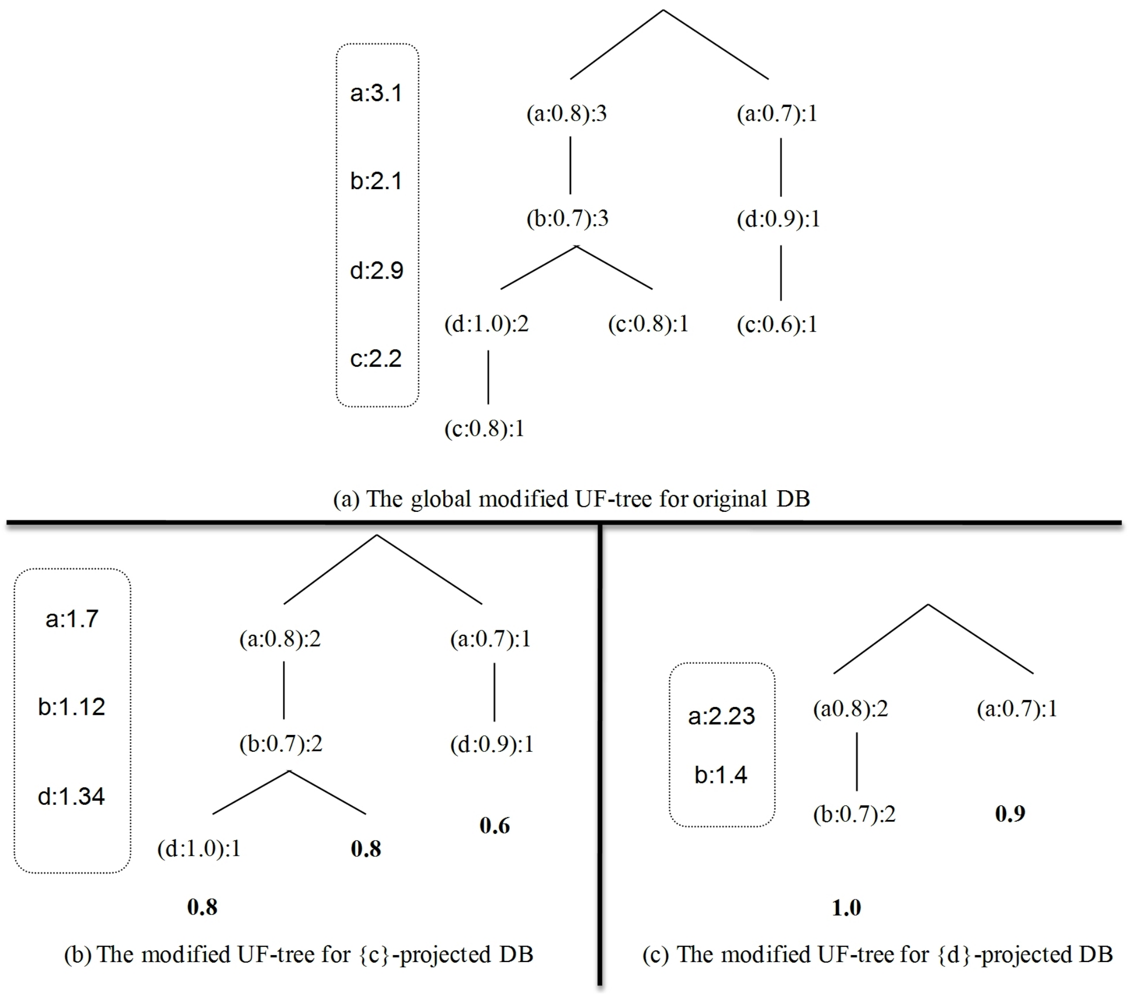


Figure 4.5: The global modified UF-tree and all projected modified UF-trees for Example 4.5.

itemsets satisfying the sum constraint $sum(X.Diving) \geq -200$m. It first checks the sum constraint against all items in the global modified UF-tree until it finds the first invalid item b. The remaining item (i.e., item a) is guaranteed to be invalid due to the non-ascending order R^{-}. Then, the algorithm "extends" (builds the projected databases) for only those valid items (i.e, items c and item d). When "extending" valid item c, the algorithm extracts the following tree paths:

- $\langle(d{:}1.0), (b{:}0.7), (a{:}0.8)\rangle$ with the occurrence count of $(c{:}0.8)$ equal to 1,
- $\langle(b{:}0.7), (a{:}0.8)\rangle$ with the occurrence count of $(c{:}0.8)$ equal to 1, and
- $\langle(d{:}0.9), (a{:}0.7)\rangle$ with the occurrence count of $(c{:}0.6)$ equal to 1.

From the information obtained above, the algorithm constructs a modified UF-tree for the $\{c\}$-projected database as shown in Figure 4.5(b). From this tree, the algorithm finds frequent itemset $\{c,\ d\}$ (with expected support $expSup(\{c,\ d\}) = (1 \times 1.0 \times 0.8) + (1 \times 0.9 \times 0.6) = 1.34$) satisfying constraint $sum(X.Diving) \geq -200$m. None of other frequent itemsets is valid.

Next, the algorithm extracts the following tree paths and forms a modified UF-tree for the $\{d\}$-projected database as shown in Figure 4.5(c):

- $\langle((b{:}0.7), (a{:}0.8)\rangle$ with the occurrence count of $(d{:}1.0)$ equal to 2, and
- $\langle(a{:}0.7)\rangle$ with the occurrence count of $(d{:}0.9)$ equal to 1.

The algorithm checks the sum constraint $sum(X.Diving) \geq -200$m against itemsets $\{d,\ b\}$ and $\{d,\ a\}$ and finds both of them are invalid. No more itemsets should be checked because the "extensions" of invalid items (itemsets) are guaranteed to be invalid. As a result, our proposed algorithm finds all valid (satisfying the aggregate constraint $sum(X.Diving) \geq -200$m) frequent itemsets $\{c\}$, $\{c,\ d\}$ and $\{d\}$ with their corresponding expected supports 2.2, 1.34 and 2.9 respectively. ■

4.4 The ACUF-growth Algorithm and Aggregate Constraint Properties

In the previous section, we have discussed the detailed procedures of applying our proposed ACUF-growth algorithm on max, min, avg and sum constraints to find those valid frequent itemsets from uncertain data. Generally, there are four constraint properties (i.e., anti-monotonicity property, monotonicity property, convertible anti-monotonicity property and convertible monotonicity property) which can be possessed by aggregate constraints. To get a clear view of how our ACUF-growth algorithm achieves the above constraint properties, we classify the forms of aggregate constraints into the following four constraint property categories:

- **Category 1: Constraints with anti-monotonicity property.** For max constraints of the form $max(X.attr) \leq const$ and min constraints of the form $min(X.attr) \geq const$, when our proposed ACUF-growth algorithm arranges domain items in a monotonic increasing or decreasing order R_1 of *attr* values such that invalid items come before/below valid items in the modified UF-tree, these constraints possess anti-monotonicity property.
- **Category 2: Constraints with monotonicity property.** For max constraints of the form $max(X.attr) \geq const$ and min constraints of the form $min(X.attr) \leq const$, when our proposed ACUF-growth algorithm arranges domain items in a monotonic decreasing or increasing order R_2 of *attr* values such that valid items come before/below invalid items in the modified UF-tree, these constraints possess monotonicity property.

- **Category 3: Constraints with convertible anti-monotonicity property.** For average constraints of any form, the sum constraints of the form $sum(X^{-}.attr) \geq const$ (where X^{-} is an itemset with all negative *attr* values) and the sum constraints of the form $sum(X^{+}.attr) \leq const$ (where X^{+} is an itemset with all positive *attr* values), when our proposed ACUF-growth algorithm arranges domain items in an order R_2 of *attr* values such that valid items come before/below invalid items in the modified UF-tree, these constraints possess convertible anti-monotonicity property.

- **Category 4: Constraints with convertible monotonicity property.** For sum constraints of the form $sum(X^{+}.attr) \geq const$ (where X^{+} is an itemset with all positive *attr* values) and the form $sum(X^{-}.attr) \leq const$ (where X^{-} is an itemset with all negative *attr* values), when our proposed ACUF-growth algorithm arranges domain items in an order R_2 of *attr* values such that valid items come before/below invalid items in the modified UF-tree, these constraints possess convertible monotonicity property.

We summarize our classification of aggregate constraints in Table 4.1. The above classification can help users to easily determine the most suitable algorithm procedure to handle aggregate constrained mining.

4.5 Summary

To find from uncertain data those frequent itemsets satisfying aggregate constraints, we first presented a naïve approach in this chapter. This approach finds all

Table 4.1: The classification of aggregate constraints by constraint properties

Constraint properties	**The forms of aggregate constraints**	**The orders of domain items by ACUF-growth**
anti-monotonicity	$max(X.attr) \leq const$ $min(X.attr) \geq const$	Invalid items before/below valid items in the modified UF-tree (order R_1)
monotonicity	$max(X.attr) \geq const$ $min(X.attr) \leq const$	Valid items before/below invalid items in the modified UF-tree (order R_2)
convertible anti-monotonicity	$avg(X.attr) \geq const$ $avg(X.attr) \leq const$ $sum(X^{-}.attr) \geq const$ $sum(X^{+}.attr) \leq const$	
convertible monotonicity	$sum(X^{+}.attr) \geq const$ $sum(X^{-}.attr) \leq const$	

frequent itemsets first, and then applies a post-processing step to check the frequent itemsets against constraints to find the valid ones. One clear weakness for such an approach is that, the algorithm computes the expected support for every itemset, regardless whether it satisfies the constraints or not. To avoid the above weakness, we proposed two improved solutions named CBSChecking and CBSComputing. Both of them can save some computational costs by pushing the constraint checking early. However, we still need to perform constraint checking for every itemset. Our ACUF-growth algorithm can be a good solution. To make a clear view of our algorithm, we discussed the general skeleton and the detailed procedures of applying it on max, min, avg and sum constraints to find those valid frequent itemsets from uncertain data. By utilizing the nice constraint properties (i.e., anti-monotonicity property, monotonicity property, convertible anti-monotonicity property and convertible monotonicity property), our algorithm does not need to check every itemset in order to determine whether or not it satisfies the aggregate constraints, which saves a lot of computational costs. By ordering the domain items based on their attribute values, we classified the forms of aggregate constraints into four different constraint property categories. Such a classification helps us understand the relation between our ACUF-growth algorithm and aggregate constraint properties. It also acts as a guide to determine the most suitable algorithm procedure to handle aggregate constrained mining.

Chapter 5

Mining Frequent Itemsets from Streams of Uncertain Data

So far, we have discussed the procedures of applying our UF-growth algorithm to find frequent itemsets from static uncertain data and the procedures of applying our ACUF-growth algorithm to find from static uncertain data those frequent itemsets which satisfy user-specific aggregate constraints (i.e., max, min, average and sum constraints). Note that all the above algorithms we proposed work well when the uncertain data are static. However, these algorithms cannot perform the frequent itemset mining for uncertain data streams. In this chapter, we propose an approximate algorithm called UF-streaming and an exact algorithm called SUF-growth to efficiently find frequent itemsets from streams of uncertain data.

5.1 An Approximate Algorithm for Mining Streams of Uncertain Data

Recall that FP-streaming algorithm [GHP$^+$04] is an approximate mining algorithm to find frequent itemsets from streams of precise data. It mainly includes two operations: (a) calling the FP-growth algorithm with a threshold *preMinsup* that is lower than the usual minimum support threshold *minsup* to find "frequent" itemsets and (b) storing and maintaining these "frequent" itemsets in a tree structure called FP-stream. (Recall from Chapter 2.4 that an itemset is "frequent" is its support is no less than *preMinsup*. Note that all truly frequent itemsets—i.e., itemsets with support $\geq$ *minsup* $>$ *preMinsup*—must be "frequent" , but not vice versa.) When the focus turns to mining frequent itemsets from streams of uncertain data, some nature questions to ask are: Can we use the similar approach to FP-streaming to handle frequent itemset mining from streams of uncertain data? If it is possible, how to perform the mining and where to store and maintain the "frequent" itemsets?

To answer the above questions, we propose an approximate algorithm—called **UF-streaming**—for mining frequent itemsets from streams of uncertain data. This algorithm can be described as follows. When the first batch of transactions in a stream of uncertain data flows in, our proposed UF-streaming algorithm applies the UF-growth algorithm to this batch. (Recall from Chapter 3 that UF-growth is an efficient tree-based algorithm for mining frequent itemsets from static databases of uncertain data.) Since data in the streams are not necessarily uniformly distributed, an itemset X that is infrequent in the current batch may be frequent in subsequent batches in the current sliding window (which may make X a frequent itemset in the current window). As data streams are continuous and unbounded, we can no longer go back to the current batch and reconsider itemset X once we move to the subsequent batches. Consequently, if the expected support of itemset X is currently slightly lower than *minsup*, we better keep X. Otherwise, we may miss X (a possibly frequent itemset). Hence, we apply the UF-growth algorithm with *preMinsup* (a threshold that is lower than the usual minimum support threshold *minsup*) to find "frequent" itemsets. Specifically, our proposed UF-streaming algorithm first constructs a UF-tree with those "frequent" items and then recursively mines "frequent" itemsets from this tree by divide-and-conquer approach. Note that in this process, an itemset is "frequent" if its expected support is no less than *preMinsup*. Although we are interested in truly frequent itemsets (i.e., itemsets having expected support $expSup \geq minsup > preMinsup$), we use *preMinsup* to avoid pruning an itemset too early. An itemset X having $preMinsup \leq expSup(X) < minsup$ is currently infrequent but may become frequent later. As a result, itemset X is not pruned.

After finding the "frequent" itemsets from incoming batch, our proposed UF-streaming algorithm stores and maintains these itemsets in another tree structure called **UF-stream**. In our UF-stream structure, each path represents a "frequent" itemset. Common items in itemsets share the tree path in a similar fashion as in the FP-tree or the UF-tree. Each node in this UF-stream structure contains (a) an item and (b) a window table (containing a list of w expected support values, one for each batch of transactions). As users are often interested in recent data than older data, our UF-stream structure focuses on capturing only w most recent batches of transactions in the stream. So, when a new batch of transactions flows in, the window slides and the expected support value of each node in the UF-stream structure shifts.

We repeat the above mining process when each of the subsequent batches of transactions in the stream of uncertain data arrives. In other words, our proposed UF-streaming algorithm first calls UF-growth algorithm to compute "frequent" itemsets from a new batch of streaming uncertain data. Our algorithm then stores those "frequent" itemsets in the UF-stream structure, slides the window, and shifts the w expected support values of each node in the UF-stream structure so as to ensure that it always captures the contents of the w most recent batches of transactions in the steam of uncertain data. To gain a better understanding of this algorithm, let us consider the following Example 5.1.

Example 5.1 Consider the following stream of uncertain data:

Batch	Transactions	Contents
First	t_1	{a:0.8, b:0.6, d:0.9, e:0.5, f:0.1}
	t_2	{b:0.7, c:0.2, d:0.8, e:0.1}
	t_3	{a:0.8, b:0.6, e:0.6}
Second	t_4	{a:0.3, b:0.6, d:0.4, e:0.5}
	t_5	{a:0.8, b:0.6}
	t_6	{a:0.8, d:0.8, e:0.9, f:0.2}
Third	t_7	{a:0.8, b:0.6, e:0.1}
	t_8	{a:0.8, b:0.6, f:0.3}
	t_9	{b:0.2, d:1.0}

In the above database of uncertain data, each transaction contains items and their corresponding existential probabilities. For example, in the first batch, the first transaction t_1 includes 5 items a, b, d, e and f. The existential probabilities of these items are 0.8, 0.6, 0.9, 0.5 and 0.1 respectively. Note that the existential probabilities of the same item may vary from one transaction to another (e.g., the existential probability of item d in transaction t_1 is 0.9 whereas that in t_2 is 0.8) and the different items may have the same existential probability in the same transaction (e.g., the existential probabilities of item b and item e are both 0.6 in transaction t_3).

UF-tree Constructing: Let the user-specified support threshold *minsup* be set to 0.9. Our proposed UF-streaming algorithm applies UF-growth to the first batch of transactions in the stream of uncertain data with a *preMinsup* lower than *minsup*. In this example, let *preMinsup* be 0.8. Our UF-streaming algorithm constructs a UF-tree for the first batch of the transactions as follows. First, it scans the current batch and accumulates expected support of each item. It finds all "frequent" items a, b, d and e (i.e., the expected support of each of these items $\geq$ *preMinsup*) with their corresponding accumulated expected supports 1.6, 1.9, 1.7 and 1.2. Item c and item f having accumulated expected supports 0.2 and 0.1 (lower than *preMinsup*) are removed because they are infrequent in the first batch.

When scanning the current batch of transactions, our proposed algorithm also inserts the frequent items from each transaction into the UF-tree. Firstly, the algorithm inserts the frequent items in t_1 into the tree, and results in a tree branch $\langle(a{:}0.8){:}1$, $(b{:}0.6){:}1$, $(d{:}0.9){:}1$, $(e{:}0.5){:}1\rangle$. It then inserts the frequent items in the second transaction t_2 into the UF-tree. A new branch $\langle(b{:}0.7){:}1$, $(d{:}0.8){:}1$, $(e{:}0.1){:}1\rangle$ is added to the tree. When we insert the frequent items in t_3 into the tree, we find the expected supports of items a and b in t_3 are the same as the expected supports of them in the tree branch for t_1, which means these two nodes can be shared. So, the algorithm increases the occurrence counts for the tree nodes $(a{:}0.8)$ and $(b{:}0.6)$ to 2. Item e is the rest frequent item in t_3 and we add it as a child of node $(b{:}0.6)$. As a result, we get the tree branch $\langle(a{:}0.8){:}2$, $(b{:}0.6){:}2$, $(e{:}0.6){:}1\rangle$. Now, all the transactions in

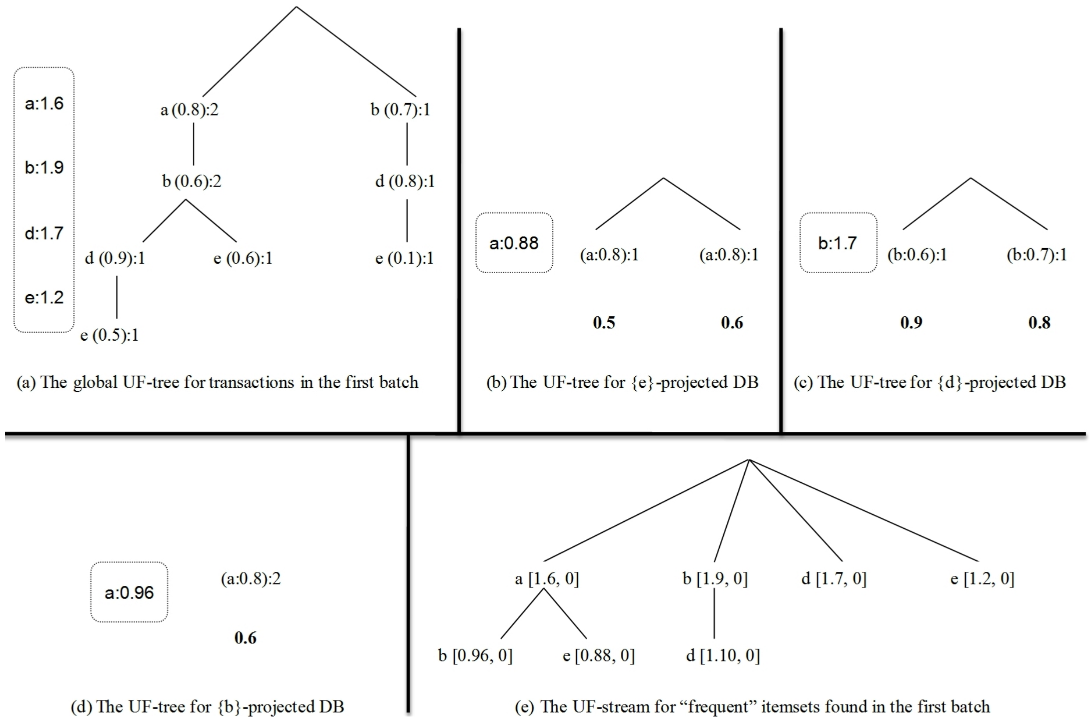


Figure 5.1: The UF-trees and the UF-stream structure for the first batch for Example 5.1.

the first batch are added into the UF-tree and we get the global UF-tree shown in Figure 5.1(a) which captures all frequent items in the first batch of uncertain data.

"Frequent" Itemset Mining: Once the global UF-tree is constructed for the first batch, our proposed UF-streaming algorithm recursively mines "frequent" itemsets from this tree with *preMinsup* = 0.8. The procedures can be briefly described as follows. The algorithm starts with item e (with $expSup(\{e\}) = 1.2$). Note that item e appears in all the three tree branches, as follows:

- $\langle(a{:}0.8), (b{:}0.6), (d{:}0.9)\rangle$ occurring once with $(e{:}0.5)$,
- $\langle(a{:}0.8), (b{:}0.6)\rangle$ occurring once with $(e{:}0.6)$, and
- $\langle(b{:}0.7), (d{:}0.8)\rangle$ occurring once with $(e{:}0.1)$.

Our algorithm forms the $\{e\}$-projected database and builds the $\{e\}$-projected UF-tree (as shown in Figure 5.1(b)) with them. From the $\{e\}$-projected UF-tree, we can calculate the expected support of itemset $\{a, e\}$ as $(1 \times 0.8 \times 0.5 + 1 \times 0.8 \times 0.6) = 0.88 \geq$ *preMinsup*. This means that itemset $\{a, e\}$ is "frequent" (i.e., possibly frequent because *minsup* $>$ *expSup*$(\{a, e\}) \geq$ *preMinsup*). The expected support of itemset $\{b, e\} = (1 \times 0.6 \times 0.5 + 1 \times 0.6 \times 0.6 + 1 \times 0.7 \times 0.1) = 0.73 <$ *preMinsup* and the expected support of itemset $\{d, e\} = (1 \times 0.9 \times 0.5 + 1 \times 0.8 \times 0.1) = 0.53 <$ *preMinsup*. As a result, itemsets $\{b, e\}$ and $\{d, e\}$ are infrequent in the first batch (they are not shown in $\{e\}$-projected database).

Similarly, our UF-streaming algorithm extracts appropriate paths and forms the $\{d\}$-projected database and $\{b\}$-projected database. With $\{d\}$-projected database and $\{b\}$-projected database, the algorithm constructs $\{d\}$-projected UF-tree and $\{b\}$-projected UF-tree as shown in Figure 5.1(c) and Figure 5.1(d). From these two trees, we can calculate the expected support of the itemset $\{b, d\}$ as *expSup*$(\{b, d\}) = (1 \times 0.6 \times 0.9 + 1 \times 0.7 \times 0.8) = 1.10 \geq$ *preMinsup* and the expected support of itemset $\{a, b\}$ as *expSup*$(\{a, b\}) = 2 \times 0.8 \times 0.6 = 0.96 \geq$ *preMinsup*. Note that the expected support of itemset $\{b, d\} = 1.10$ and the expected support of $\{a, b\} = 0.96$ are both higher than the *minsup*, which means the itemset $\{b, d\}$ and itemset $\{a, b\}$ are true frequent (comparing with "frequent"). After computing expected supports for all itemsets, our UF-streaming algorithm reports all "frequent" itemsets in the first batch of the steam of uncertain data. They are $\{a\}$, $\{a, b\}$, $\{a, e\}$, $\{b\}$, $\{b, d\}$, $\{d\}$ and $\{e\}$ (with their corresponding expected supports of 1.6, 0.96, 0.88, 1.8, 1.10, 1.7 and 1.2 respectively).

Let the window size $w = 2$ batches. In the second step of our proposed UF-streaming algorithm, we store these "frequent" itemsets in a UF-stream structure shown in Figure 5.1(e). The node $a[1.6,\ 0]$ in this UF-stream structure represents the "frequent" itemset $\{a\}$ with an expected support of 1.6 in the first batch of the stream of uncertain data (and an expected support of 0 in the second batch as we have not yet read/mined the second batch). Similarly, the node $b[0.96,\ 0]$ on the leftmost path $\langle a[1.6,\ 0],\ b[0.96,\ 0]\rangle$ represents the "frequent" itemset $\{a,\ b\}$ with an expected support of 0.96 in the first batch of the stream of uncertain data.

When the second batch of the data flows in, our proposed UF-streaming algorithm applies the same mining procedure as the first batch to this batch. To elaborate, it first constructs a new global UF-tree (Figure 5.2(a)) with those "frequent" items (i.e., $a,\ b,\ d,\ e$) on the second batch. It then applies the UF-growth algorithm to this tree as follows. It starts from item e (with $\mathit{expSup}(\{e\}) = 1.4$). The algorithm extracts the following two tree branches and forms the $\{e\}$-projected database:

- $\langle(a{:}0.3),\ (b{:}0.6),\ (d{:}0.4)\rangle$ occurring once with $(e{:}0.5)$, and
- $\langle(a{:}0.8),\ (d{:}0.8)\rangle$ occurring once with $(e{:}0.6)$.

After that, the algorithm constructs the $\{e\}$-projected UF-tree (as shown in Figure 5.2(b)). From this tree, our proposed the UF-streaming algorithm calculates the expected support for itemset $\{a,\ e\}$ as $\mathit{expSup}(\{a,\ e\}) = (1 \times 0.3 \times 0.5 + 1 \times 0.8 \times 0.9) = 0.87$ and the expected support for itemset $\{d,\ e\}$ as $\mathit{expSup}(\{d,\ e\}) = (1 \times 0.4 \times 0.5 + 1 \times 0.8 \times 0.9) = 0.92$. With the $\mathit{preMinsup} = 0.8$ and $\mathit{minsup} = 0.9$, we know that itemset $\{a,\ e\}$ is "frequent" and itemset $\{d,\ e\}$ is true frequent. After mining the entire UF-tree which captures the items in the second batch, our proposed

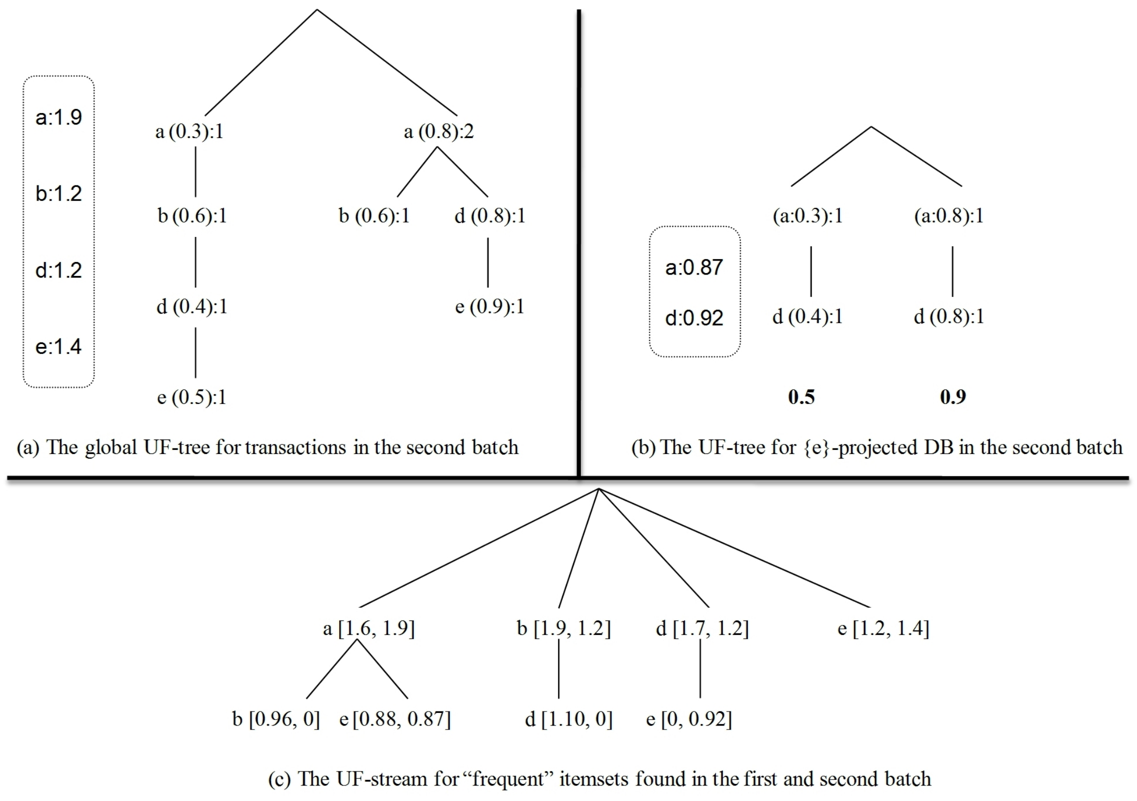


Figure 5.2: The UF-trees for the second batch and the UF-stream structure for the first and second batches for Example 5.1.

UF-streaming algorithm obtains the " frequent" itemsets $\{a\}$, $\{a, e\}$, $\{b\}$, $\{d\}$, $\{d, e\}$ and $\{e\}$ (with their corresponding expected supports of 1.9, 0.87, 1.2, 1.2, 0.92 and 1.4).

After finding all "frequent" itemsets from the second batch, our proposed UF-streaming algorithm updates the existing UF-stream structure by inserting these new "frequent" itemsets into it. The resulting UF-stream structure, as shown in Figure 5.2(c), consists of eight nodes. Note that the node for $\{a\}$ is now changed from $a[1.6,\ 0]$ (as in Figure 5.1(e)) to $a[1.6,\ 1.9]$ (as in Figure 5.2(c)) as we now know that the expected supports of $\{a\}$ both batches: 1.6 in the first batch and 1.9 in the second batch. A new node $e[0, 0.92]$ represents a new "frequent" itemset $\{d, e\}$ that has no expected support in the first batch and an expected support of 0.92 in the second batch. Similarly, the node $b[0.96, 0]$ means the "frequent" itemset $\{a, b\}$ has an expected support of 0.96 in the first batch and no expected support in the second batch.

Applying the similar approach to the first and second batches of stream of uncertain data, our proposed UF-streaming algorithm constructs the UF-trees and updates the UF-stream structure when the third batch of the data flows in. Specifically, it first finds the "frequent" items a, b, d in the third batch and constructs a new global UF-tree (Figure 5.3(a)) with them. Then, it forms the $\{b\}$-projected database and constructs the corresponding $\{b\}$-projected UF-tree (as shown in Figure 5.3(b)). By recursively growing "frequent" itemsets, the algorithm obtains all "frequent" itemsets $\{a\}$, $\{a, b\}$, $\{b\}$, and $\{d\}$ (with their corresponding expected supports of 1.6, 0.96, 1.4 and 1.0). After that, our proposed UF-streaming algorithm slides the window (of size $w = 2$ batches) and shifts the expected support values of each node in the UF-stream structure. The resulting UF-stream structure, as shown in Figure 5.3(c), captures the expected support values for "frequent" itemsets found in the second and third batches. (Note that nodes with zero expected support, such as d[0, 0], can be removed from the UF-stream structure.) ■

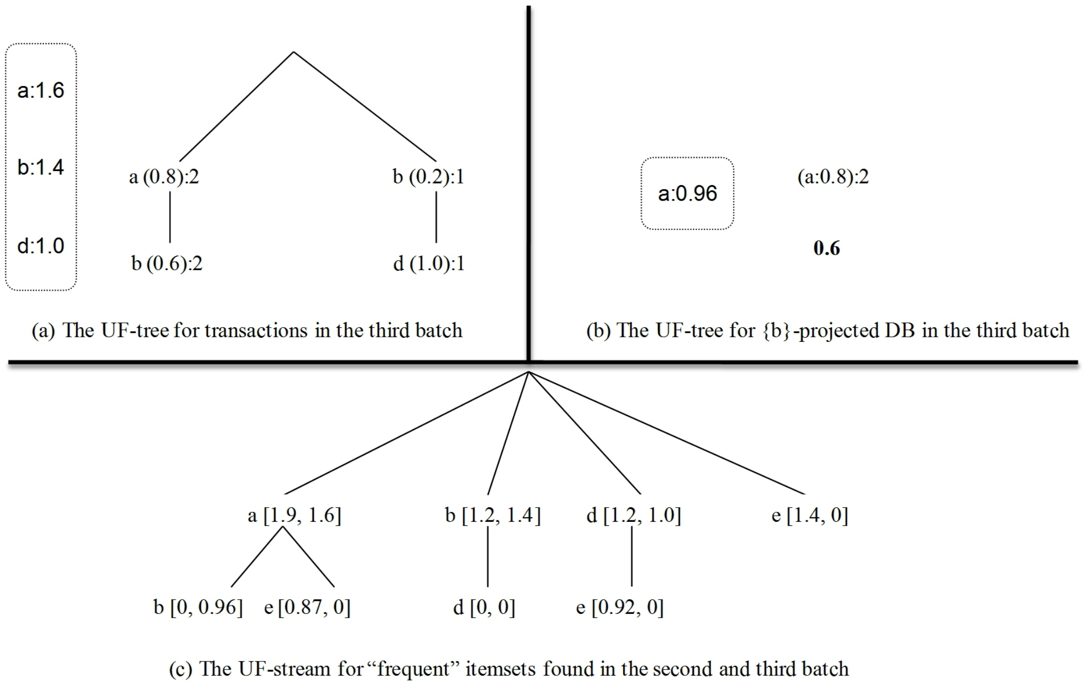

a:1.6
b:1.4
d:1.0
a (0.8):2
b (0.6):2
b (0.2):1
d (1.0):1
(a) The UF-tree for transactions in the third batch
a:0.96
(a:0.8):2
0.6
(b) The UF-tree for {b}-projected DB in the third batch
a [1.9, 1.6]
b [1.2, 1.4]
d [1.2, 1.0]
e [1.4, 0]
b [0, 0.96]
e [0.87, 0]
d [0, 0]
e [0.92, 0]
(c) The UF-stream for "frequent" itemsets found in the second and third batch

Figure 5.3: The UF-trees for the third batch and the UF-stream structure for the second and third batches for Example 5.1.

5.2 An Exact Algorithm for Mining Streams of Uncertain Data

In the previous section, we proposed the UF-streaming algorithm for mining frequent itemsets from streams of uncertain data. There are some potential problems associated with such an approximate algorithm. First of all, the UF-streaming algorithm calls the UF-growth algorithm with *preMinsup* which is lower than the actual *minsup*. As a result, the algorithm finds "frequent" itemsets (i.e., itemsets with expected support $expSup \geq preMinsup$). And we know that some of these itemsets are not truly frequent (e.g., some itemsets may have expected support $expSup < minsup$).

Consequently, to find truly frequent itemsets, one needs to apply a post-processing step. Moreover, the success of UF-streaming strongly depends on the value of *preMinsup*. If it is too high (e.g., too close to *minsup*), we may lose frequency information of some itemsets. To another extreme, if it is too low, a lot of redundant itemsets (e.g., those itemsets with expected support higher than *preMinsup* but lower than *minsup*) may be generated and stored. So, similar to *minsup*, it is not easy to find an appropriate value for *preMinsup*. Second, the UF-streaming algorithm requires an extra data structure (the UF-stream structure) to store the mined itemsets. Third, the UF-streaming algorithm uses an "immediate" mode for mining, which leads to lots of computation wasting, especially when many batches flow in before the user requests for the mining results (frequent itemsets). Let us consider an example with a sliding window of size $w = 3$ batches. If the user requests the mining results at the end of the 100th batch, the UF-streaming algorithm has already computed "frequent" itemsets for each of the 100 batches, out of which only those from the last three batches are needed for the mining results. Computation on the first 97 batches is wasted.

In this section, we propose an exact algorithm, called **SUF-growth**, which further improves our UF-streaming algorithm (by avoiding the aforementioned potential problems) for mining frequent itemsets from streams of uncertain data. Our proposed algorithm not only returns to the user all and only those true frequent itemsets (i.e., those with expected support $expSup \geq minsup$) by using *minsup* instead of *preMinsup* but also saves the memory space which is used to save the UF-stream structure by using only one new tree structure called **SUF-tree** to perform the frequent itemset mining. Unlike the UF-streaming algorithm that uses the "immediate" mode for mining, the new SUF-growth algorithm applies a "delayed" mode for mining. With such a mode, a lot of computation could be saved, especially when many batches flow in before the user requests for the mining results (frequent itemsets). To better understand the advantage of the "delayed" mode, let us revisit the aforementioned example with a sliding window of size $w = 3$ batches. If the user requests the mining results at the end of the 100th batch, when we apply "immediate" mode based mining algorithm such as UF-streaming, we know that the algorithm has already computed "frequent" itemsets for each of the 100 batches, out of which only those from the last three batches are needed for the mining results. Computation on the first 97 batches is wasted. However, when we apply "delayed" mode based mining algorithm such as our proposed SUF-growth algorithm, we only focus on the last three batches of uncertain data. As a result, a lot of unnecessary computation has been saved.

Now, let us explain how our proposed SUF-growth algorithm finds all the frequent itemsets from streams of uncertain data using the SUF-tree. The key idea of our proposed SUF-growth algorithm can be described as follows. The algorithm first constructs an SUF-tree, and then extracts relevant paths from this SUF-tree (which is a global tree) to recursively form smaller UF-trees for projected databases. Because the data streams are dynamic and not necessarily uniformly distributed (their distributions are usually changing by time), expected supports of items are continuously affected by the arrival of new batches and the removal of the contents of older batches. Arranging items in frequency-dependent order in the SUF-tree may lead to swapping—which, in turn, can cause merging and splitting—of tree nodes when the global frequencies of items change. Hence, in the SUF-tree, items are arranged according to some canonical order (e.g., lexicographic order), which can be specified by the user prior to the construction of the SUF-tree or the mining process. Because the data streams are continuous and unbounded, the SUF-tree can be constructed using only one scan of the streams of uncertain data, and the resulting SUF-tree captures the contents of the streams. Moreover, the SUF-tree preserves the usual tree properties. They are: (a) the occurrence count of a node is at least as high as the sum of occurrence counts of its children and (b) the ordering of items is unaffected by the continuous changes in the expected support values of items.

To record and update the information at each tree node, the SUF-tree keeps a list of occurrence counts (instead of only one occurrence count as in the UF-tree). Each count in this list captures the occurrence of the item in each of the corresponding batch. By doing so, when the window slides (i.e., when new batches arrive and older batches are deleted), information can be updated easily. Specifically, whenever a new batch of transactions flow in, the occurrence count of the node $(x{:}expSup(x))$ in the new batch is appended to the list for such a node. In other words, the last entry of the list for such a node then shows the occurrence of such a node in the current batch. Afterwards, when the next batch of transactions flow in, the contents in the list are shifted forward. The last entry shifts and becomes the second-last entry. This shifting leaves room (the last entry) for the newest batch. At the same time, the occurrence count corresponding to the oldest batch in the window is discarded. This has the same effect as deleting from the window those transactions in the oldest batch.

Theoretically, to effectively shift the list of occurrence counts, one may need to traverse all the tree nodes and shift the list of entries in each node. Practically, we do not need to do so. Instead, we use a pointer to indicate the last update of each node. For example, let us consider a node (x:*expSup*(x)). Suppose we are processing the j-th batch of the streaming uncertain data. If the pointer in the node points to the entry representing the $(j-1)$-th batch, then this indicates that the node has just been visited when processing the $(j-1)$-th batch. Otherwise, the pointer points to a much earlier entry representing the $(j-k)$-th batch for some $k > 1$, which indicates that the node has not been visited since then and the occurrence count of such a node for the entries in-between should be 0 (i.e., the node does not occur in $(j-k+1)$-th, ..., $(j-1)$-th batches). By doing so, we avoid traversing all the nodes in the SUF-tree.

Next, let us focus on how to perform the actual mining with our SUF-growth algorithm. With the SUF-tree, the actual mining of frequent itemsets from streams of uncertain data is "delayed" until it is needed. In other words, once the SUF-tree is constructed, it is always kept up-to-date when the window slides. Consequently, one can mine frequent itemsets from this up-to-dated SUF-tree in a fashion similar to the UF-growth algorithm using an appropriate *minsup*. More specifically, mining with the SUF-tree employs a divide-and-conquer approach. The algorithm forms projected databases (e.g., $\{d\}$-projected database, $\{d, c\}$-projected database, $\{d, b\}$-projected database, etc.) by traversing the paths upwards only. Since items are consistently arranged according to some canonical order, one can guarantee the inclusion of all frequent items using just upward traversals. There is also no worry about possible omission or doubly-counting of frequent items during the mining process. As the SUF-tree is always kept up-to-date, all frequent itemsets in current stream can be found effectively. To get a better understanding of our proposed SUF-growth algorithm, let us consider the following example.

Example 5.2 Reconsider the stream of uncertain data as shown in Example 5.1. Let *minsup* be 0.9 and the window size w be 2 batches (indicating that only two batches of streaming transactions are kept in the tree at each time point T).

In the first step, our SUF-growth algorithm constructs an SUF-tree by inserting the items in the first and second batches of transactions into it. Figure 5.4 shows the resulting global SUF-tree at time T which captures the transactions in the first and second batches. Note that each node in the SUF-tree contains (a) an item, (b) its expected support and (c) a list of occurrence count (instead of just one occurrence count). For instance, tree node (b:0.6)[2,1] means with expected support 0.6, item b occurs in the first batch two times and occurs in the second batch one time. Also, we keep all (both frequent and infrequent) items. For example, item f is infrequent in the first batch and in the second batch. However, we still keep it in the SUF-tree for transactions in the first and second batches (at time T). As discussed above, the order of items in the SUF-tree is based on some canonical order. In this example, we use the lexicographic order (i.e., item a, item b, item c, ..., item f) from the tree top to the tree leaves.

When the third batch of transactions flow in, our proposed SUF-growth algorithm updates the list of occurrence counts by discarding the occurrence count corresponding to the oldest batch (i.e., the first batch) and adding the occurrence count corresponding to the new batch (i.e., the third batch) to the last entry of the list. The resulting SUF-tree which captures the transactions in the second and third batches at time T'

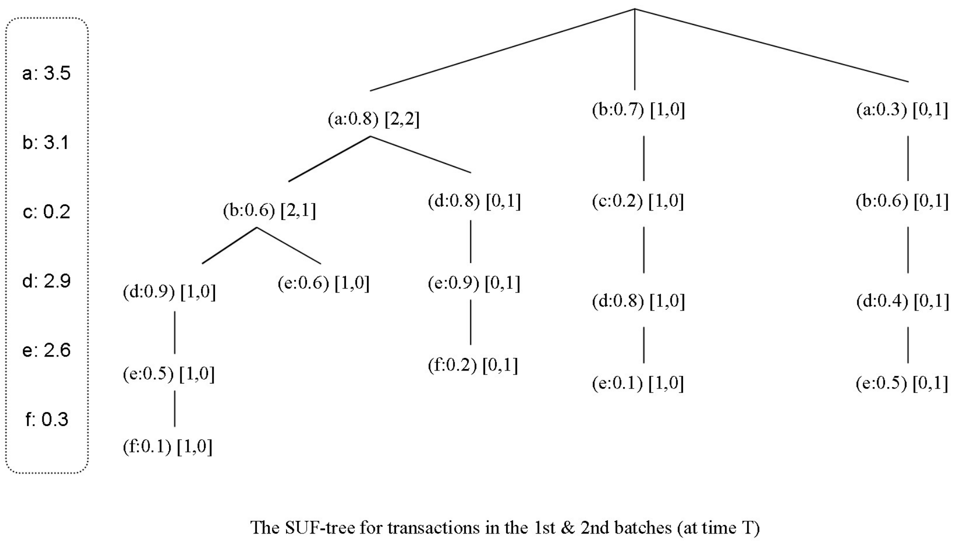


Figure 5.4: The global SUF-tree for the first and second batches for Example 5.2.

is shown in Figure 5.5(a). Note that all tree nodes with occurrence count [0,0] have been removed from the SUF-tree which captures the transactions in the second and third batches.

After the SUF-tree is constructed, our focus turns to how to discover the frequent itemsets from the tree. Note that our proposed SUF-growth algorithm is "delay" mode based mining algorithm, which means we are only interested in the current SUF-tree (i.e., in this example, the SUF-tree which captures the second and the third batches at time T'). Now, let us recursively apply the divide-and-conquer based mining approach to find all frequent itemsets for this tree. Firstly, our proposed SUF-growth algorithm calculates the expected support for each item in the global SUF-tree. Note that the occurrence of any tree node can be computed by summing the occurrence counts in the list for that tree node. For instance, the occurrence of

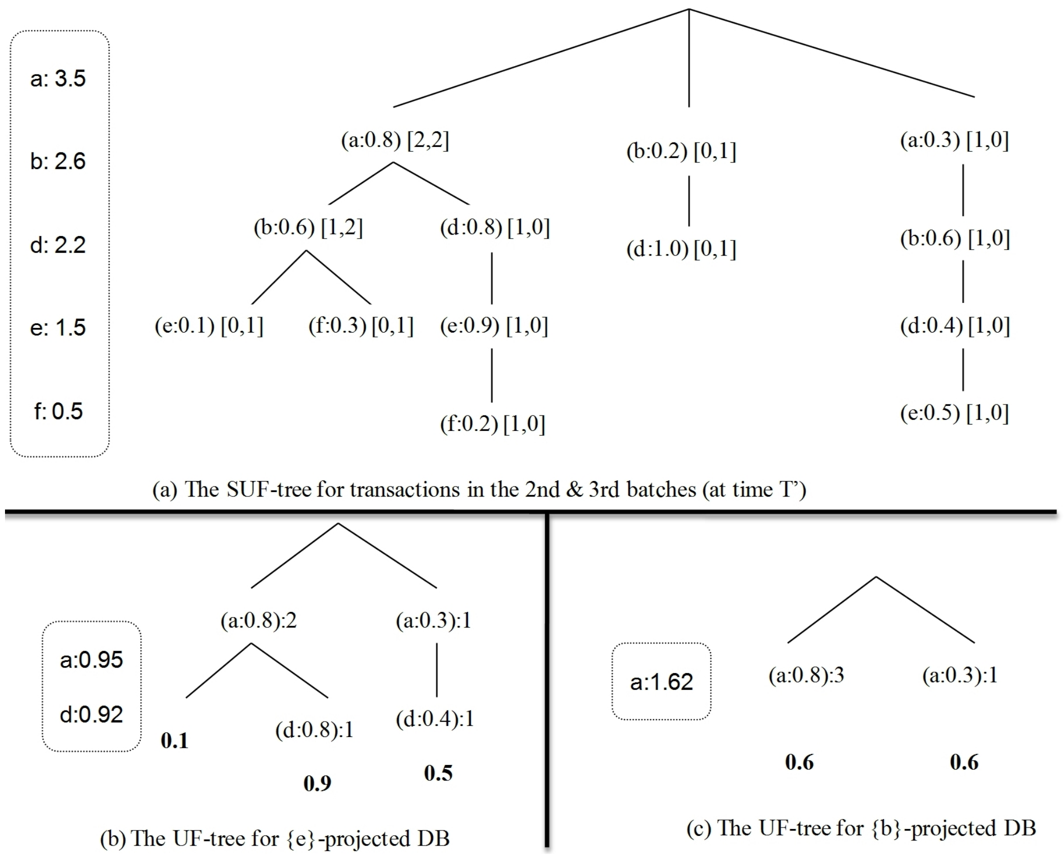


Figure 5.5: The global SUF-tree for the second and third batches and all its projected UF-trees for Example 5.2.

the first tree node ((a:0.8)[2,2]) can be computed as 2 + 2 = 4. Consequently, the expected support of item a can be computed as $expSup(\{a\}) = [(2+2) \times 0.8] + [(1+0) \times 0.3] = 3.5$. With such a way, the algorithm finds the expected supports for all other single itemsets. They are $expSup(\{b\}) = [(1+2) \times 0.6] + [(0+1) \times 0.2] + [(1+0) \times 0.6] = 2.6$, $expSup(\{d\}) = [(1+0) \times 0.8] + [(0+1) \times 1.0] + [(1+0) \times 0.4] = 2.2$, $expSup(\{e\}) = [(0+1) \times 0.1] + [(1+0) \times 0.9] + [(1+0) \times 0.5] = 1.5$ and $expSup(\{f\}) = [(0+1) \times 0.3] + [(1+0) \times 0.2] = 0.5$. Since the $minsup$ is equal to 0.9, we know that items a, b, c, d and e are frequent at time T'. Item f is not frequent at time T' because its expected support (0.5) is lower than the *minsup*.

Next, our proposed SUF-growth algorithm forms projected databases by traversing the paths upwards (from tree leaves to tree root). It first forms the $\{e\}$-projected database by traversing relevant tree paths which contain e node. Here, the $\{e\}$-projected database contains the items from the following three tree paths:

- $\langle(a{:}0.8), (b{:}0.6)\rangle$ occurring once with $(e{:}0.1)$,
- $\langle(a{:}0.8), (d{:}0.8)\rangle$ occurring once with $(e{:}0.9)$, and
- $\langle(a{:}0.3), (d{:}0.4)\rangle$ occurring once with $(e{:}0.5)$.

With the above information, our proposed SUF-growth algorithm constructs the $\{e\}$-projected UF-tree as shown in Figure 5.5(b). From this $\{e\}$-projected UF-tree, the algorithm calculates the expected supports for itemsets $\{e,\ a\}$ and $\{e,\ d\}$ as *expSup*($\{e,\ a\}$) $= [1 \times 0.8 \times 0.1] + [1 \times 0.8 \times 0.9] + [(1 \times 0.3 \times 0.5] = 0.95$ and *expSup*($\{e,\ d\}$) $= [1 \times 0.8 \times 0.9] + [(1 \times 0.4 \times 0.5] = 0.92$. Since the *minsup* is equal to 0.9, we know that both of them are frequent. Similarly, our SUF-growth algorithm extracts appropriate paths and forms the $\{b\}$-projected database which only contains one item a. The algorithm then builds the $\{b\}$-projected UF-tree (as shown in Figure 5.5(c)) and uses it to calculate the expected support for itemsets $\{b,\ a\}$ as *expSup*($\{b,\ a\}$) $= [3 \times 0.8 \times 0.6] + [(1 \times 0.3 \times 0.6] = 1.62$. As a result, our proposed SUF-growth algorithm reports all true frequent itemsets in the second and third batches of the stream of uncertain data at time T'. They are $\{a\}$, $\{b\}$, $\{b,\ a\}$, $\{d\}$, $\{e\}$, $\{e,\ a\}$ and $\{e,\ d\}$ (with their corresponding expected supports of 3.5, 2.6, 1.62, 2.2, 1.5, 0.95 and 0.92 respectively).■

5.3 Summary

In this chapter, we proposed two algorithms, named UF-streaming and SUF-growth, for mining frequent itemsets from streams of uncertain data. Among them, UF-streaming is an approximate algorithm which applies the UF-growth algorithm with *preMinsup* (which is lower than actual *minsup*) to find "frequent" itemsets and uses a tree-structure called UF-stream to store and maintain mined "frequent" itemsets. As an "immediate" mode based mining algorithm, UF-streaming completely performs the mining process since the stream of uncertain data flows in.

Unlike the UF-streaming algorithm, SUF-growth is an exact algorithm to mine true frequent itemsets from streams of uncertain data. SUF-growth algorithm constructs SUF-tree to store information (i.e., items, their expected supports and a list of occurrence counts) from data streams and applies divide-and-conquer approach (by building projected databases and their corresponding projected trees) to find frequent itemsets. As a "delayed" mode based mining algorithm, SUF-growth only performs the mining process when users request the mining results. With such a way, the algorithm saves a lot of unnecessary computation. Table 5.1 summarizes the major differences between our UF-streaming algorithm and SUF-growth algorithm.

Table 5.1: UF-streaming algorithm vs. SUF-growth algorithm

	UF-streaming algorithm	**SUF-growth algorithm**
Algorithm property	approximate algorithm	exact algorithm
Tree structure	UF-tree and UF-stream	SUF-tree
Mining threshold	*preMinsup*	*minsup*
Mining mode	"immediate" mode	"delayed" mode

Chapter 6

Experimental Evaluation

To evaluate our proposed algorithms that perform uncertain frequent itemset mining (i.e., UF-growth algorithm), constrained frequent itemset mining from uncertain data (i.e., ACUF-growth algorithm) and frequent itemset mining from streams of uncertain data (i.e., UF-streaming algorithm and SUF-growth algorithm), we performed several experiments. The experimental results cited in this chapter are based on the data generated by the program developed at IBM Almaden Research Center [AS94]. The data contain 10k to 1M records with an average transaction length of 10 items, and a domain of 1,000 items. We assigned an existential probability from the range (0,1] to each item in each transaction. In addition to this database, we also conducted the following experiments using some other databases such as UCI real-life databases [FA10] and FIMI databases [GZ03, BGZ04]. The observations or trends were consistent.

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

CPU and I/Os; it includes the time for both tree construction and frequent itemset mining steps. In the experiments, we mainly evaluated the efficiency of the proposed algorithms.

6.1 The Mining of Frequent Itemsets from Uncertain Data

First six experiments are designed to evaluate the algorithms which mines frequent itemsets from uncertain data. We focus on our proposed UF-growth algorithm and its Apriori-based counterpart, namely the U-Apriori algorithm.

Experiment 6.1 In the first experiment, we tested the effect of *minsup*. We fixed the database size as 10k records with an average transaction length of 10 items and a domain of 1,000 items and varied the *minsup*. We compared the runtimes of two

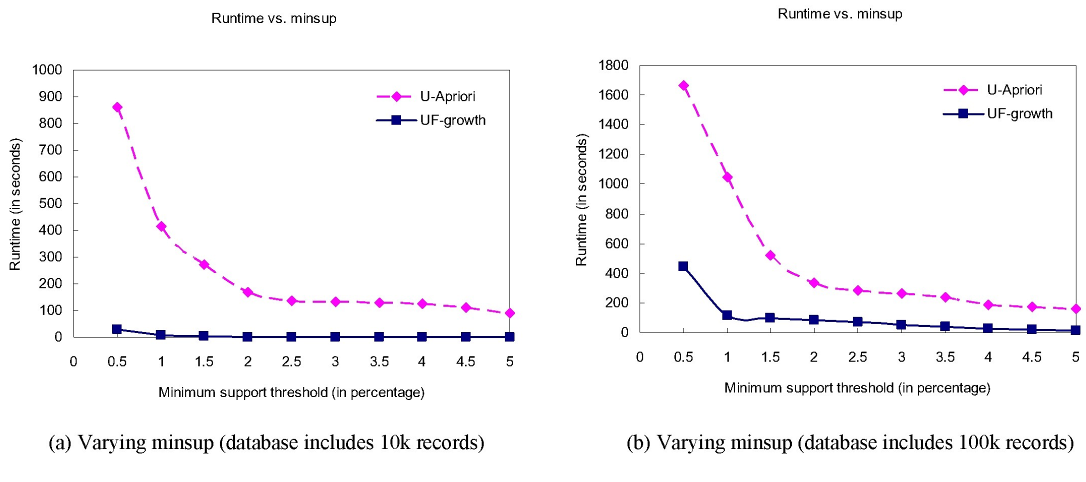
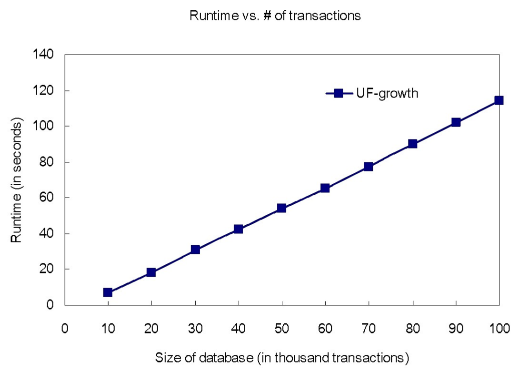


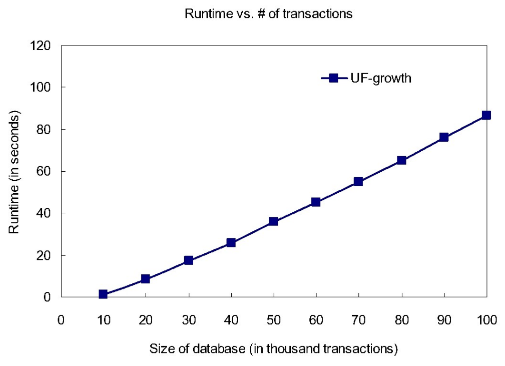
Figure 6.1: Runtimes of the UF-growth and U-Apriori algorithms with varying *minsup* for Experiment 6.1.

algorithms: (1) the U-Apriori algorithm [CKH07] and (2) our UF-growth algorithm. As shown in Figure 6.1(a), with *minsup* increasing, runtimes for both U-Apriori and UF-growth algorithms decreased. From this figure, we observed that the runtime for our UF-growth algorithm is lower than the runtime for the U-Apriori algorithm. The U-Apriori algorithm relies on the costly candidate generation process. It explains why the experimental result showed that our UF-growth algorithm required much less runtime. We also repeated the same experiment with a database which contains 100k records with an average transaction length of 10 items and a domain of 1,000 items. As shown in Figure 6.1(b), the experimental result presented exactly the same trends as the previous experiment. ■

Experiment 6.2 In the second experiment, we tested scalability with the number of transactions for our UF-growth algorithm. We fixed the *minsup* and the distribution of existential probability. The variant in this experiment is the size of transaction



(a) Varying transaction database size (minsup = 0.01)



(b) Varying transaction database size (minsup = 0.02)

Figure 6.2: Runtime of the UF-growth algorithm with varying database size for Experiment 6.2.

database. Specifically, we varied the size of transaction database from 10k transactions to 100k transactions. The experimental result showed that when the number of transactions increased, the runtime for our UF-growth algorithms increased. Moreover, the mining with our proposed algorithm had linear scalability. See Figure 6.2(a) with *minsup* equal to 0.01 and Figure 6.2(b) with *minsup* equal to 0.02. ■

Experiment 6.3 In the third experiment, we tested the effect of the distribution of item existential probabilities by varying the average value of existential probabilities. Here, we fixed the database size as 100k records with an average transaction length of 10 items and a domain of 1,000 items and the *minsup*. The experimental result showed that with the average value of existential probabilities increased, the runtime of our UF-growth algorithm increased. See Figure 6.3(a) with *minsup* equal to 0.01 and Figure 6.3(b) with *minsup* equal to 0.02. An explanation for such an experimental result is that, when the average value of existential probabilities was small, the

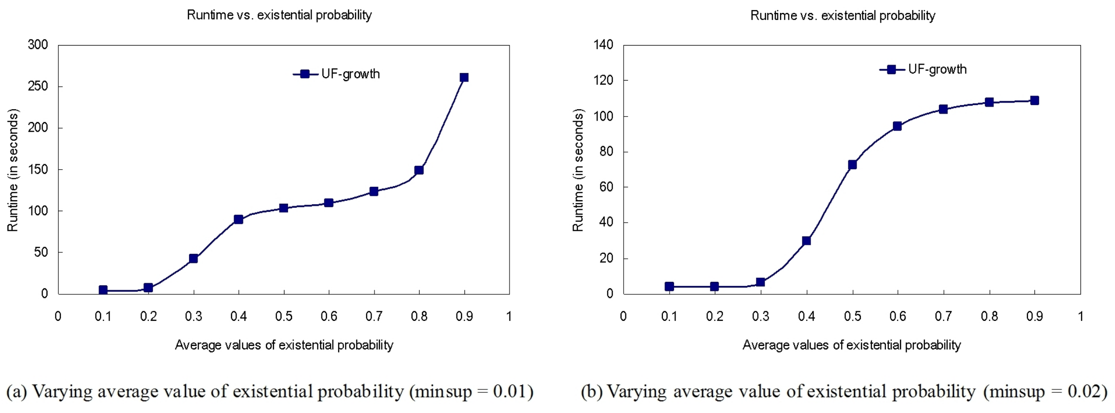
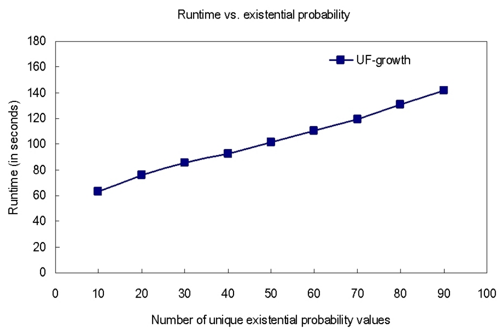


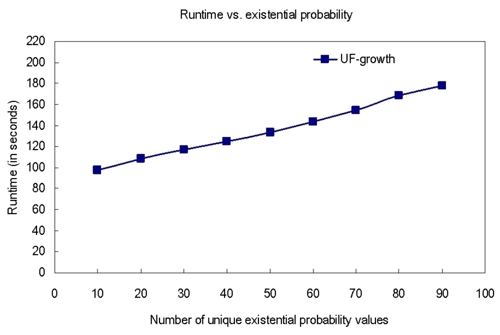
Figure 6.3: Runtime of the UF-growth algorithm with varying average value of existential probability for Experiment 6.3.

itemsets had less chance to be frequent. This led to low computation cost and small number of frequent itemsets (i.e., the number of frequent itemsets is only 1,742 when the average value of existential probabilities is equal to 0.2 and the *minsup* is equal to 0.01). As a result, the runtime for our UF-growth algorithm was short. When the average value of existential probabilities became higher, the itemsets had more chance to be frequent. The computation cost and the number of frequent itemsets increased (i.e., the number of frequent itemsets reaches 735,672 when the average value of existential probability is equal to 0.8 and the *minsup* is equal to 0.01), which required longer runtime. ■

Experiment 6.4 The fourth experiment also tested the effect of the distribution of item existential probabilities. This time, we varied the number of unique existential probability values. In this experiment, We fixed the database size as 100k records with an average transaction length of 10 items and a domain of 1,000 items and the *minsup*. We also fixed the average value of existential probabilities as 0.5. The experimental



(c) Varying number of unique existential probability (minsup = 0.01)

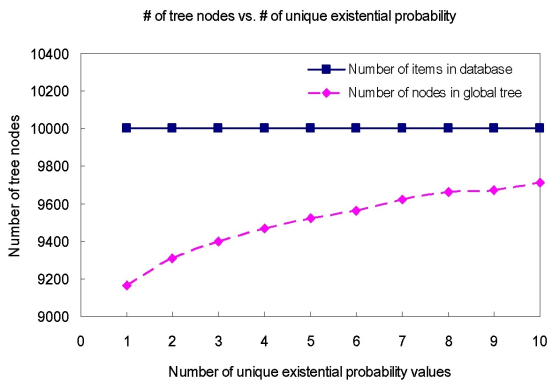


(d) Varying number of unique existential probability (minsup = 0.02)

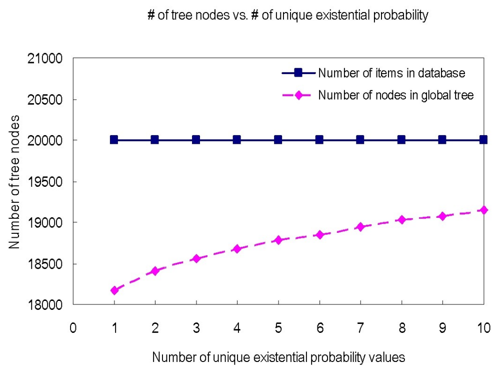
Figure 6.4: Runtime of the UF-growth algorithm with varying number of unique existential probability for Experiment 6.4.

result showed that when the number of unique existential probability values increased, the runtime of our UF-growth algorithm increased. See Figure 6.4(a) with *minsup* equal to 0.01 and Figure 6.4(b) with *minsup* equal to 0.02. An explanation for such an experimental result is that, when the distribution of existential probability was sparse (i.e., items took on a few unique existential probability values), the UF-tree was small due to a high chance for path sharing. A small UF-tree resulted in a short runtime. On the other hand, when the distribution of existential probability was denser (i.e., items took on a large number of unique existential probability values), the UF-tree was larger and it required longer runtime to mine. ■

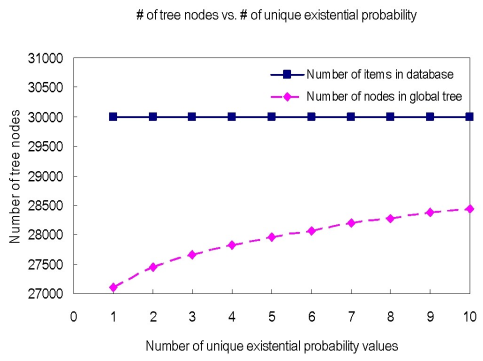
Experiment 6.5 In the fifth experiment, we evaluated the relation between the distribution of item existential probabilities and the number of nodes in the UF-tree. Here, we fixed *minsup* as 0.01 and the average value of existential probabilities as 0.5. We varied the number of unique existential probability values from 1 to 10. The experimental result showed that when the number of unique existential probability values increased, the number of nodes in global UF-tree increased. See Figure 6.5(a) with a transaction database containing exactly 10k items, Figure 6.5(b) with a transaction database containing exactly 20k items and Figure 6.5(c) with a transaction database containing exactly 30k items. An explanation for such an experimental result is that, with the number of unique existential probability values increased, the chance for path sharing in an UF-tree decreased. As a result, the number of nodes in global UF-tree increased. ■



(a) Database contains 10k items



(b) Database contains 20k items



(c) Database contains 30k items

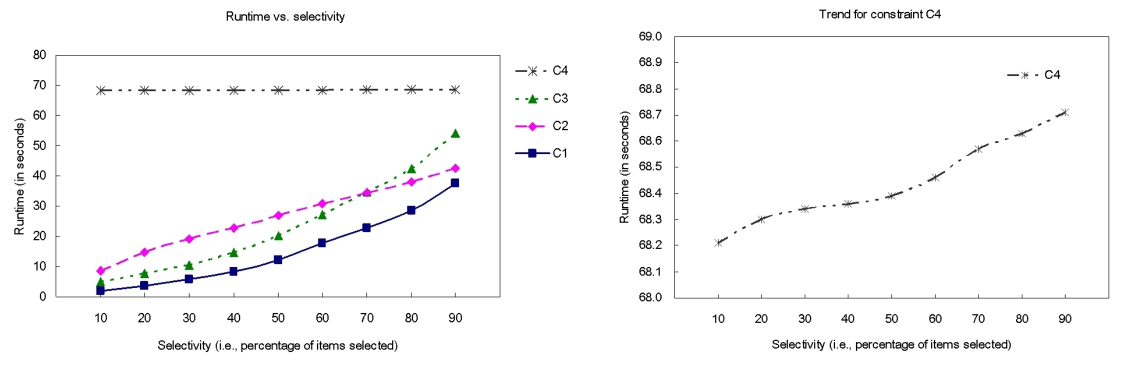
Figure 6.5: Number of nodes in global UF-tree with varying existential probability for Experiment 6.5 and Experiment 6.6.

Experiment 6.6 In the sixth experiment, we measured the number of nodes in a UF-tree. The result showed that the number of nodes in the global UF-tree representing a database is no more than the number of items in all transactions of such a database. From Figure 6.5, we observed that the number of tree nodes is less than the number of items for all transaction databases with 10k items (Figure 6.5(a)), 20k items (Figure 6.5(b)) and 30k items (Figure 6.5(c)). ■

6.2 The Mining of Constrained Frequent Itemsets from Uncertain Data

Recall that aggregate constraints mainly possess four types of properties. They are anti-monotonicity property, monotonicity property, convertible anti-monotonicity property and convertible monotonicity property. To clearly show the results of our experiments, we name the aggregate constraint possessing anti-monotonicity property as C_1 (i.e., $max(X.Price) < \$60$), the aggregate constraint possessing monotonicity property as C_2 (i.e., $min(X.Weight) \leq 60$kg), the aggregate constraint possessing convertible anti-monotonicity property C_3 (i.e., $avg(X.Temperature) \geq -10°$C) and the aggregate constraint possessing convertible monotonicity property as C_4 (i.e., $sum(X.Rainfall) \geq 100$mm). All five experiments showed in this section are based on these four types of aggregate constraints.

Experiment 6.7 The seventh experiment was designed for our ACUF-growth algorithm. We tested the effect of selectivity in this experiment. We fixed the database size as 100k records with an average transaction length of 10 items and a domain of 1,000 items. We first set the *minsup* as 0.01 and varied the percentage of items selected (selectivity). The result showed that when the selectivity increased, the runtimes for all C_1, C_2, C_3 and C_4 increased. For verifying our observation, we also set the *minsup* to 0.02 and 0.03. All experiments indicated that the trend was consistent. Figure 6.6(a) showed the result of our experiment with $minsup = 0.02$. Since this figure cannot clearly show the increasing trend for constraint C_4, we zoomed y-axis and obtained Figure 6.6(b) which clearly presented the relationship between selectivity and runtime for C_4. ■



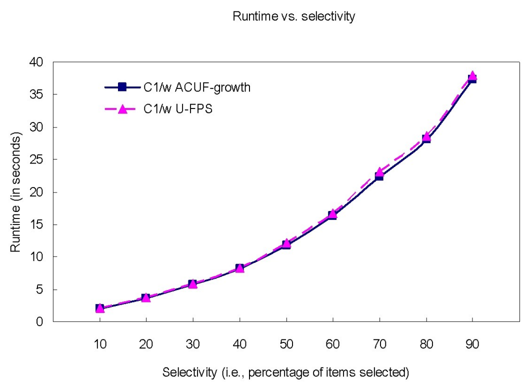
(a) Varying selectivity for all constraint types (b) Zoomed-in view for constraint C4 with varying selectivity

Figure 6.6: Runtime of the ACUF-growth algorithm with varying selectivity for Experiment 6.7 and Experiment 6.8.

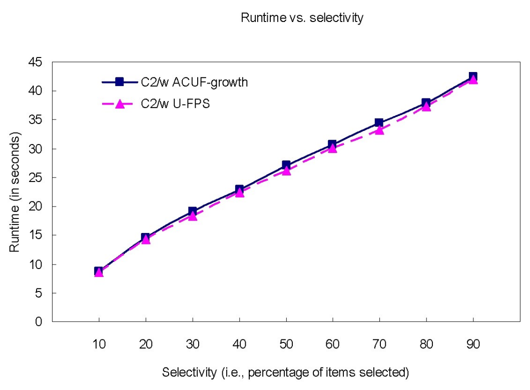
Experiment 6.8 In the eighth experiment, we evaluated the performance of our proposed ACUF-growth algorithm on four types of aggregate constraints. From Figure 6.6(a), we observed that C_4 incurred the highest runtime because the algorithm "extended" both valid and invalid items when handling convertible monotone constraints. The runtimes for C_2 and C_3 were in the middle because for both of them, only valid items were "extended". Among them, C_2 incurred more runtime than C_3 when the selectivity was low and C_3 incurred more runtime than C_2 when the selectivity was high. When selectivity was low (i.e., the set of valid items was small and the set of invalid items was large), monotone constraint C_2 "extended" valid items with all these large amount of invalid items to form the valid frequent itemsets. On the other side, convertible monotone constraint C_3 only "extended" valid items with part of invalid items to form the potential valid frequent itemsets and then checked constraint against them to find those frequent itemsets satisfying the constraint. At this moments, the runtime required for additional frequent itemset forming for C_2

was more than the runtime for the additional constraint checking for C_3. However, when selectivity was high (i.e., the set of valid items was large and the set of invalid items was small), monotone constraint C_2 "extended" valid items with small amount of invalid items to form the valid frequent itemsets. On the other side, the set of potential valid frequent itemsets for further constraint checking by convertible monotone constraint C_3 was large. At this moments, the runtime required for additional constraint checking for C_3 was more than the runtime for additional frequent itemset forming for C_2. As a result, the runtime for C_2 and C_3 had a cross point at around 70% selectivity. Anti-monotone constraint C_1 incurred the lowest runtime because the algorithm only "extended" valid items and the valid frequent itemsets did not include any invalid items. ■

Experiment 6.9 In the ninth experiment, we compared the performance of our proposed ACUF-growth algorithm with the U-FPS algorithm on anti-monotone constraint C_1 and on monotone constraint C_2 separately. When mining frequent itemsets from uncertain data with anti-monotone constraint C_1, both our ACUF-growth algorithm and the U-FPS algorithm "extended" only valid items. Moreover, all valid frequent itemsets were formed only with valid items. As shown in Figure 6.7(a), the runtimes for two algorithms were very close. Our ACUF-growth algorithm incurred slightly lower runtime than the U-FPS algorithm because U-FPS checked all items to distinguish invalid items from valid ones. However, due to the item ordering, our ACUF-growth algorithm stopped checking constraints whenever it detected the first valid item. The following table illustrates the slight difference of the runtimes between



(a) Comparing ACUF-growth with U-FPS on C1



(b) Comparing ACUF-growth with U-FPS on C2

Figure 6.7: The performance comparison of the ACUF-growth and U-FPS algorithms for Experiment 6.9.

our ACUF-growth algorithm and U-FPS algorithm when mining with constraint C_1 (as shown in Figure 6.7(a)).

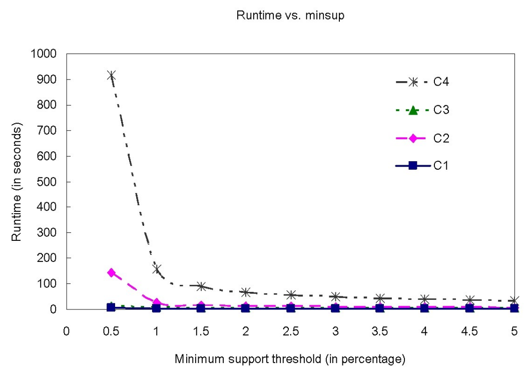
Selectivity	**Runtime for ACUF growth**	**Runtime for U-FPS**
10%	2.04s	2.19s
20%	3.56s	3.72s
30%	5.73s	5.85s
40%	8.19s	8.29s
50%	11.71s	12.12s
60%	16.27s	16.72s
70%	22.31s	23.08s
80%	28.06s	28.70s
90%	37.26s	37.94s

When mining frequent itemsets from uncertain data with monotone constraint C_2, both our ACUF-growth algorithm and the U-FPS algorithm "extended" only valid items. Moreover, all valid frequent itemsets were formed by valid items together with invalid items. As shown in Figure 6.7(b), our ACUF-growth algorithm required slightly higher runtime than the U-FPS algorithm because U-FPS mined constrained frequent itemsets from a more compact UF-tree (as U-FPS arranged items in non-ascending frequency order from leaves to root) than the modified UF-tree built by our ACUF-growth algorithm. The following table illustrates the slight difference of the runtimes between our ACUF-growth algorithm and the U-FPS algorithm when mining with constraint C_2 (as shown in Figure 6.7(b)).

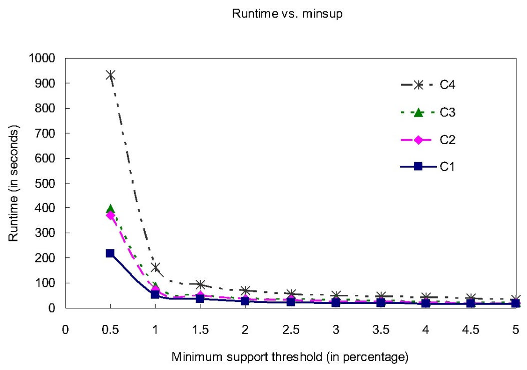
Selectivity	**Runtime for ACUF growth**	**Runtime for U-FPS**
10%	8.67s	8.56s
20%	14.60s	14.33s
30%	19.06s	18.41s
40%	22.81s	22.47s
50%	27.02s	26.24s
60%	30.72s	30.04s
70%	34.39s	33.30s
80%	37.93s	37.28s
90%	42.45s	41.93s

■

Experiment 6.10 In the tenth experiment, we tested the effect of *minsup* for our ACUF-growth algorithm. We fixed the database size as 100k records with an average transaction length of 10 items and a domain of 1,000 items. We first set the selectivity as 20% and varied the *minsup*. The result showed that when the *minsup* increased, the runtimes for all C_1, C_2, C_3 and C_4 decreased. For verifying our observation, we also set the selectivity as 40%, 60% and 80%. All experiments indicated that the trend was consistent. Figure 6.8(a) showed the result of our experiment with selectivity = 20% and Figure 6.8(b) showed the result of our experiment with selectivity = 80%. From Figure 6.8(a), we also observed that when selectivity was low (i.e., selectivity = 20%), C_4 incurred the highest runtime, C_2 incurred the second highest runtime, C_3 incurred the third runtime and C_4 incurred the lowest runtime. From Figure 6.8(b), we found that when selectivity was high (i.e., selectivity = 80%), C_4 incurred the



(a) Varying minsup for all constraint types with selectivity=20%



(b) Varying minsup for all constraint types with selectivity=80%

Figure 6.8: Runtime of the ACUF-growth algorithm with varying *minsup* for Experiment 6.10.

highest runtime, C_3 incurred the second highest runtime, the runtime for C_2 was slightly lower than C_3, and C_4 incurred the lowest runtime. This observation proved that the result for our performance evaluation (Experiment 6.8) was correct. ■

Experiment 6.11 In the last experiment for our ACUF-growth algorithm, we tested the scalability with the number of transactions. We fixed the *minsup* as 0.02 and selectivity as 50%. As shown in Figure 6.9, when the number of transactions increased, the runtimes for all C_1, C_2, C_3 and C_4 increased. ■

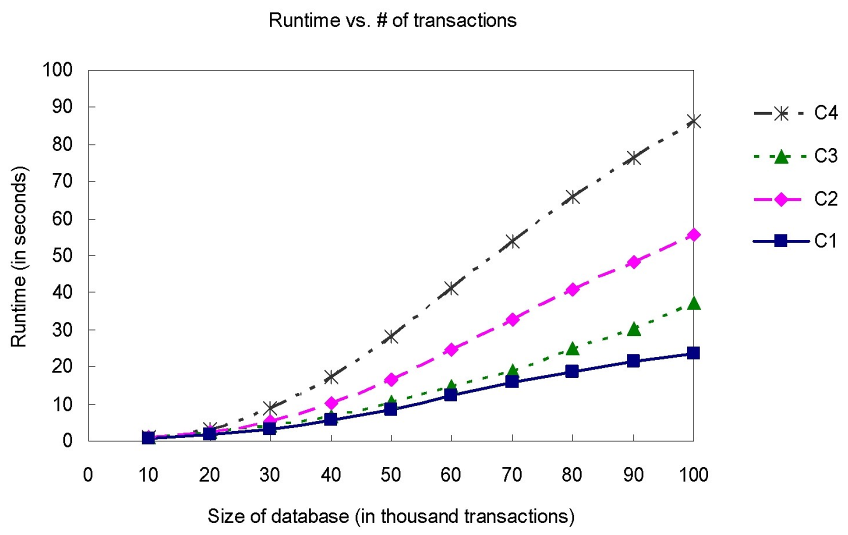


Figure 6.9: Runtime of the ACUF-growth algorithm with varying database size for Experiment 6.11.

6.3 The Mining of Frequent Itemsets from Streams of Uncertain Data

For evaluating the algorithms mining frequent itemsets from streams of uncertain data (i.e., our UF-streaming and SUF-growth algorithms), we used the dataset containing 1M records with an average transaction length of 10 items, and a domain of 1,000 items. We assigned an existential probability from the range (0,1] to each item in each transaction. We also set each batch to be 0.1M transactions and the window size to be $w = 5$ batches.

Experiment 6.12 We tested the effect of *minsup* in the twelfth experiment. We fixed the database size and the distribution of item existential probabilities. Theoretically, the runtimes for both UF-streaming and SUF-growth algorithms decrease when

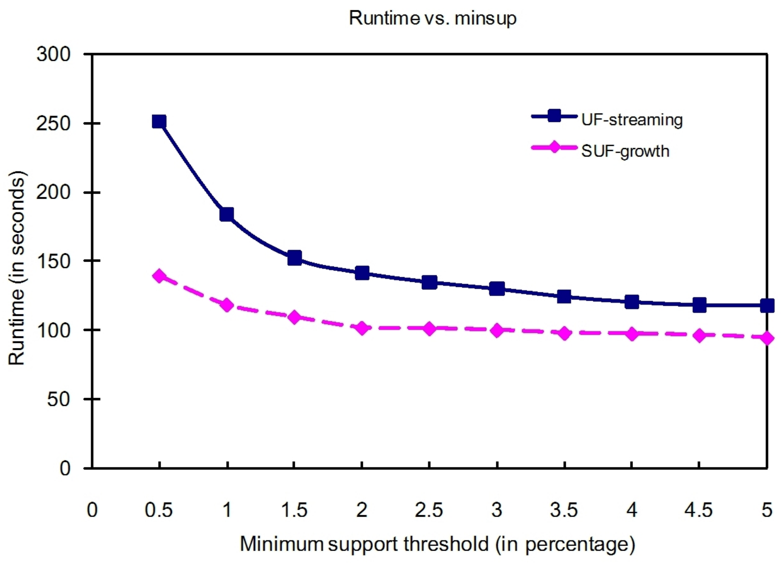


Figure 6.10: Runtimes of the UF-streaming and SUF-growth algorithms with varying *minsup* for Experiment 6.12 and Experiment 6.13.

minsup (or *preMinsup*) increases. Experimental result (as shown in Figure 6.10) confirmed that, when *minsup* increased, fewer itemsets had expected support $\geq$ *minsup*, and thus shorter runtimes were required. ■

Experiment 6.13 In the thirteenth experiment, we measured the efficiency of proposed UF-streaming and SUF-growth algorithms. The result showed that the UF-streaming algorithm responded to the user quicker than the SUF-growth algorithm because the former used the "immediate" mining mode so that it just needed to retrieve relevant paths from the UF-stream structure when the user asked for the mining result. However, the result also showed that the total execution costs for the SUF-growth algorithm was shorter than that for the UF-streaming algorithm (as shown in Figure 6.10), especially when the user requested for the mining results infrequently or near the end of a long data stream. It was because the SUF-growth algorithm used the "delayed" mining mode so that mining was done only when it was needed. ■

Experiment 6.14 In the fourteenth experiment, we evaluated the scalability of our UF-streaming and SUF-growth algorithms. We fixed the *minsup* and the distribution of item existential probabilities. From Figure 6.11, we knew that when the size of transaction database increased, the runtimes for both UF-streaming and SUF-growth algorithms increased. Moreover, the runtimes of our algorithms were linear with respect to the number of transactions. ■

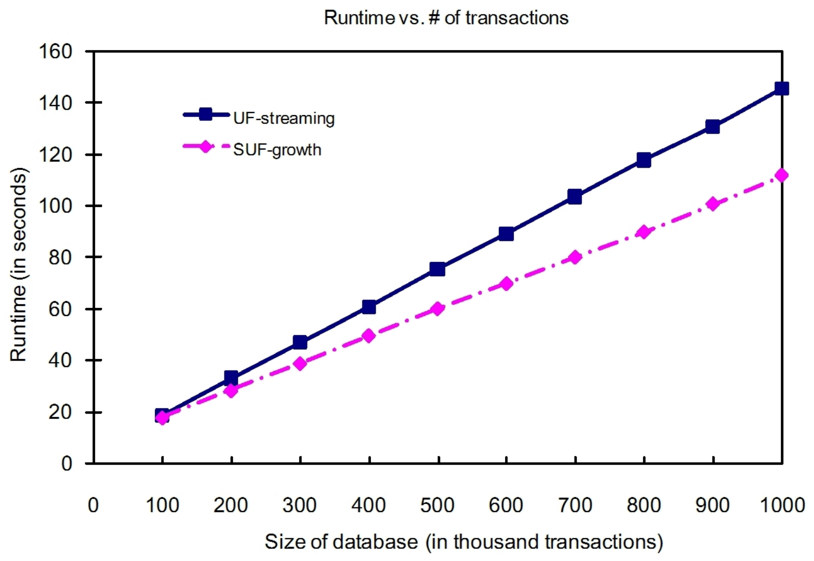


Figure 6.11: Runtimes of the UF-streaming and SUF-growth algorithms with varying database size for Experiment 6.14.

Similarly as what we did for our UF-growth algorithm, we also tested the effect of existential probability distribution by varying the number of unique existential probability values and the average value of the existential probabilities for the UF-streaming and SUF-growth algorithms. The experiment results showed: (1) when the number of unique existential probability values increased, the runtimes of both our UF-streaming and SUF-growth algorithms increased; (2) when the number of unique existential probability values increased, the number of nodes in the global UF-tree (or the SUF-tree) increased and (3) when the average values of existential probabilities increased, the runtimes of both our UF-streaming and SUF-growth algorithms increased.

6.4 Summary

In this chapter, we evaluated the algorithms which perform uncertain frequent itemset mining (i.e., UF-growth algorithm), constrained frequent itemset mining from uncertain data (i.e., ACUF-growth algorithm) and frequent itemset mining from streams of uncertain data (i.e., UF-streaming and SUF-growth algorithms) by sets of experiments. For our UF-growth algorithm, we compared it with the U-Apriori algorithm and showed that our algorithm is more efficient than U-Apriori by consuming less runtime. The experimental results also showed that when the size of transaction database, average value of existential probabilities or number of unique existential probability increased, the runtime for our UF-growth algorithm increased. However, when the value of *minsup* increased, the runtime for our algorithm decreased. Moreover, when the number of unique existential probability increased, the number of the tree nodes in global UF-tree increased. Nevertheless, the number of tree nodes in global UF-tree was no more than the number of items in the transaction database (which was used to build the global UF-tree) in all situations.

For our ACUF-growth, UF-streaming and SUF-growth algorithms, we did the similar experiments and received the similar results such as when the size of transaction database increased, the runtime for all these algorithms increased. Since our ACUF-growth algorithm was a constraint based algorithm, we also evaluated it by selectivity. The result showed that when constraint selectivity increased, the runtime for our ACUF-growth algorithm increased. The experiment results cited in this chapter showed the properties of our proposed algorithms and demonstrated the effectiveness of them.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Frequent itemset mining plays an essential role in the mining of various patterns and is in demand in many real life applications. However, there are many real-life situations in which the data in transaction databases are uncertain. This calls for uncertain frequent itemset mining. In uncertain frequent itemset mining, the user may be only interested in some small specific subsets of all frequent itemsets. This calls for constrained frequent itemset mining of uncertain data. Moreover, due to advances in technology, a flood of uncertain data can be produced in many situations. This leads to frequent itemset mining from streams of uncertain data. To deal with these situations, we proposed in this M.Sc. thesis the following four algorithms:

- UF-growth algorithm, which efficiently finds frequent itemsets from uncertain data;
- ACUF-growth algorithm, which mines from uncertain data for those frequent

itemsets which satisfy user-specified aggregate constraints;

- UF-streaming algorithm, which is an approximate algorithm designed to find frequent itemsets from streams of uncertain data; and
- SUF-growth algorithm, which is an exact algorithm designed to find frequent itemsets from streams of uncertain data.

Among them, the UF-growth algorithm constructs a UF-tree to capture the important contents of uncertain data where each item is associated with an existential probability. Each node in the UF-tree not only contains an item and its expected support, but also the number of occurrence of such expected support for such an item. After the UF-tree is constructed, the algorithm recursively mines frequent itemsets from this UF-tree by a divide-and-conquer (by building projected databases and their corresponding projected trees) approach.

The ACUF-growth algorithm builds a modified UF-tree in which the order of the tree nodes is based on a specific order R. Here, the order R is closely related to the forms of aggregate constraints handled by the algorithm to make sure the constraints possessing nice properties (e.g., anti-monotonicity property, monotonicity property, convertible anti-monotonicity property, and convertible monotonicity property). These constraint properties helps our ACUF-growth algorithm efficiently mine constrained frequent itemsets from uncertain data without checking every itemset to determine whether or not it satisfies the aggregate constraints.

UF-streaming is an approximate algorithm, which applies the UF-growth algorithm with *preMinsup* (which is lower than actual *minsup*) to find "frequent" itemsets and uses a tree-structure called UF-stream to store and maintain mined "frequent" itemsets. Moreover, UF-streaming is an "immediate" mode based mining algorithm which starts mining "frequent" itemsets as soon as streams of uncertain data flow in. Differing from the UF-streaming algorithm, SUF-growth is an exact algorithm which mines truly frequent itemsets with actual *minsup*. The SUF-growth algorithm first captures uncertain data stream in a SUF-tree where each tree node contains the item, its expected support and a list of occurrence count for the item with such an expected support. The algorithm then applies a divide-and-conquer approach (similar as the UF-growth algorithm) to find frequent itemsets from the SUF-tree. As an "delayed" mode based mining algorithm, SUF-growth only performs the mining when the user requests. With such a way, the algorithm saves a lot of unnecessary computation comparing to the "immediate" mode based mining algorithm (i.e., UF-streaming).

We evaluated the algorithms which perform uncertain frequent itemset mining (i.e., UF-growth algorithm), constrained frequent itemset mining from uncertain data (i.e., ACUF-growth algorithm) and frequent itemset mining from streams of uncertain data (i.e., UF-streaming and SUF-growth algorithms) by sets of experiments. The experiment results showed the behaviors of our proposed algorithms and illustrated the effectiveness of them.

7.2 Future Work

In this thesis, we have proposed, developed and evaluated four algorithms, namely UF-growth, ACUF-growth, UF-streaming and SUF-growth algorithms. Among them, UF-growth is designed for mining frequent itemsets from uncertain data; ACUF-growth is proposed to mine from uncertain data for those frequent itemsets which satisfy user-specific aggregate constraints (i.e., min, max, average and sum); UF-streaming and SUF-growth can efficiently find frequent itemsets from streams of uncertain data. When mining frequent itemsets from streams of uncertain data, the user may be only interested in a tiny portion of mined data. In other words, mining constrained frequent itemsets from streams of uncertain data is in demand. As future work, we plan to design a tree-based system which integrates uncertain mining, stream mining and constrained mining. The resulting system would fulfill the task of efficiently mining from streams of uncertain data for those frequent itemsets which satisfy user-specific constraints.

To elaborate a bit, based on principles of our UF-streaming and ACUF-growth algorithms, we could first do constraint checking on the domain items in the transactions of current batch and order these domain items in a specific order (i.e., non-descending order R^+ or non-ascending order R^- of attribute values). We could then construct a modified UF-tree for the current batch of streaming uncertain data. The structure of this modified UF-tree would be exactly same as that of the modified UF-tree we built for our ACUF-growth algorithm. Next, we could recursively grow the valid “frequent” itemsets from this modified UF-tree with a *preMinsup* which is lower than the actual *minsup*. Finally, we would store all valid “frequent” itemsets in a UF-stream structure.

As discussed in this thesis, our SUF-growth algorithm improved our UF-streaming algorithm by (1) returning to the user all and only those truly frequent itemsets, (2) avoiding the additional UF-stream structure to store mined itemsets and (3) reducing unnecessary computation by using a “delayed” mode for mining. For our planed system, we could also take these advantages by integrating our SUF-growth algorithm with our ACUF-growth algorithm. Specifically, we could first do constraint checking on the domain items in the transactions of current batch and order these domain items in a specific order (i.e., non-descending order R^+ or non-ascending order R^- of attribute values). Then, we could construct a modified SUF-tree in which the items would be arranged in non-descending order R^+ or non-ascending order R^- of attribute values from tree leaves to tree root based on the forms of constraints. In the last step, we would recursively mine from this modified SUF-tree for those truly frequent itemsets that satisfy the user-specific constraints by using the same approach as our SUF-growth algorithm.

Like many other frequent itemset mining algorithms, all algorithms we proposed in this thesis assume that the UF-trees (or the modified UF-trees) used in the mining process can all fit in memory. While the available memory space is sufficient in many situations, there are still some other situations where the available memory space is quite limited. As future work, we plan to upgrade our system to handle these situations. Specifically, we plan to to find from streams of uncertain data for those constrained frequent itemsets in a limited memory environment. Inspired by the ACoCo algorithm (which is an approximate algorithm for mining from streams of precise data those constrained frequent itemsets in a limited memory environment) and the ECoCo algorithm [LBY08] (which is an exact algorithm for mining from streams of precise data those constrained frequent itemsets in a limited memory environment), one possible solution is proposing a new tree structure called UCoCo tree (indicates uncertain CoCo tree) which would be a modification of CoCo tree.

Similar to our UF-tree, each node in an UCoco tree would store (1) an item, (2) its expected support, and (3) the number of occurrence of such expected support for such an item. Except for this modification, the UCoCo tree would keep all other characters of the original CoCo tree [LBY08]. By building a global UF-tree (modified UF-tree) and using UCoCo trees, our system would fulfill the task of finding from streams of uncertain data those constrained frequent itemsets in a limited memory environment.

Note that all algorithms we proposed in this thesis—UF-growth, ACUF-growth, UF-streaming, and SUF-growth algorithms—returned resulting frequent itemsets as textual lists. It is well known that "a pictures is worth a thousand words". With a visual representation of frequent itemsets, the users could easily find the information (i.e., the relations among mining results) in which they would be interested. As another future work, we plan to build a visualization system which could be used to visualize and analyze the mining results of our algorithms.

Bibliography

[Agg07] C.C. Aggarwal. On density based transforms for uncertain data mining. In *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE)*, pages 866–875, Istanbul, Turkey, 2007.

[AIS93] R. Agrawal, T. Imielinski, and A. Swanmi. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 207–216, Washington, DC, USA, 1993.

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

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, Santiago de Chile, Chile, 1994.

[AY08a] C.C. Aggarwal and P.S. Yu. A framework for clustering uncertain data streams. In *Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE)*, pages 150–159, Cancun, Mexico, 2008.

[AY08b] C.C. Aggarwal and P.S. Yu. Outlier detection with uncertain data. In *Proceedings of the 8th SIAM International Conference on Data Mining (SDM)*, pages 483–493, Atlanta, GA, USA, 2008.

[AY09] C.C. Aggarwal and P.S. Yu. A survey of uncertain data algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 21(5):609–623, 2009.

[BAG99] R.J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the 15th IEEE International Conference on Data Engineering (ICDE)*, pages 188–197, Sydney, Australia, 1999.

[BGZ04] R. Bayardo, B. Goethals, and M.J. Zaki, editors. *Proceedings of the Second IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, Brighton, UK, 2004.

[BKR+09] T. Bernecker, H.P. Kriegel, M. Renz, F. Verhein, and A. Zuefle. Probabilistic frequent itemset mining in uncertain databases. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 119–128, Paris, France, 2009.

[BMS97] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 265–276, Tucson, AZ, USA, 1997.

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

[CCKN06] M. Chau, R. Cheng, B. Kao, and J. Ng. Uncertain data mining: an example in clustering location data. In *Proceedings of the 10th Pacific-Asia Conference on Knowledge Discovery and Data mining (PAKDD)*, pages 199–204, Singapore, 2006.

[CH08] G. Cormode and M. Hadiieleftheriou. Finding frequent items in data streams. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, pages 1530–1541, Auckland, New Zealand, 2008.

[CK08] C.K. Chui and B. Kao. A decremental approach for mining frequent itemsets from uncertain data. In *Proceedings of the 12th Pacific-Asia Conference on Knowledge Discovery and Data mining (PAKDD)*, pages 64–75, Osaka, Japan, 2008.

[CKH07] C.K. Chui, B. Kao, and E. Hung. Mining frequent itemsets from uncertain data. In *Proceedings of the 11th Pacific-Asia Conference on Knowledge Discovery and Data mining (PAKDD)*, pages 47–58, Nanjing, China, 2007.

[CM08] G. Cormode and A. McGregor. Approximation algorithms for clustering uncertain data. In *Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 191–200, Vancouver, BC, Canada, 2008.

[CWC09] D.Y. Chiu, Y.H. Wu, and A.L. Chen. Efficient frequent sequence mining by a dynamic strategy switching algorithm. *The International Journal on Very Large Data Bases (VLDB Journal)*, 18(1):303–327, 2009.

[CXP$^+$04] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 876–887, Toronto, ON, Canada, 2004.

[CZ03] W. Cheung and O.R. Zaïane. Incremental mining of frequent patterns without candidate generation or support constraint. In *Proceedings of the Seventh International Database Engineering and Applications Symposium (IDEAS)*, pages 111–116, Hong Kong, China, 2003.

[DYM$^+$05] X. Dai, M.L. Yiu, N. Maamouilis, Y. Tao, and M. Vaitis. Probabilistic spatial queries on existentially uncertain data. In *Proceedings of the Ninth International Symposium on Spatial and Temporal Databases (SSTD)*, pages 400–417, Angra dos Reis, Brazil, 2005.

[DYMV05] X. Dai, M.L. Yiu, N. Mamoulis, and M. Vaitis. Probabilistic spatial queries on existentially uncertain data. In *Proceedings of the 9th International Symposium on Spatial and Temporal Databases (SSTD)*, pages 400–417, Angra dos Reis, Brazil, 2005.

[EZ03] M. El-Hajj and O.R. Zaïane. COFI-tree mining: a new approach to pattern growth with reduced candidacy generation. In *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, Melbourne, FL, USA, 2003.

[FA10] A. Frank and A. Asuncion. UCI machine learning repository. School of Information and Computer Science, University of California, Irvine, CA, USA, 2010.

[GHP$^+$04] C. Giannella, J. Han, J. Pei, X. Yan, and P.S. Yu. Mining frequent patterns in data streams at multiple time granularities. In *Data Mining: Next Generation Challenges and Future Directions*. AAAI/MIT Press, 2004.

[GLW00] G. Grahne, L.V.S. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE)*, pages 512–521, San Diego, CA, USA, 2000.

[GZ03] B. Goethals and M.J. Zaki, editors. *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI)*, Melbourne, FL, USA, 2003.

[GZK05] M.M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: a review. *ACM SIGMOD Record*, 34(2):18–22, 2005.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–12, Dallas, TX, USA, 2000.

[JG06a] N. Jiang and L. Gruenwald. CFI-stream: mining closed frequent itemsets in data streams. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 592–597, Philadelphia, PA, USA, 2006.

[JG06b] N. Jiang and L. Gruenwald. Research issues in data stream association rule mining. *ACM SIGMOD Record*, 35(1):14–19, 2006.

[KP05] H.P. Kriegel and M. Pfeifle. Density-based clustering of uncertain data. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 672–677, Chicago, IL, USA, 2005.

[LB08] C.K. Leung and D.A. Brajczuk. Efficient mining of frequent itemsets from data streams. In *Proceedings of the 25th British National Conference on Databases (BNCOD)*, pages 2–14, Cardiff, UK, 2008.

[LB09a] C.K. Leung and D.A. Brajczuk. Efficient algorithms for mining constrained frequent patterns from uncertain data. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 9–18, Paris, France, 2009.

[LB09b] C.K. Leung and D.A. Brajczuk. Mining uncertain data for constrained frequent sets. In *Proceedings of the 13th International Database Engineering and Applications Symposium (IDEAS)*, pages 109–120, Cetraro, Italy, 2009.

[LBY08] C.K. Leung, D.A. Brajczuk, and J. Yu. Efficient algorithms for stream mining of constrained frequent patterns in a limited memory environment. In *Proceedings of the 13th International Database Engineering and Applications Symposium (IDEAS)*, pages 189–198, Coimbra, Portugal, 2008.

[LC10] X. Lian and L. Chen. Ranked query processing in uncertain databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(3):420–436, 2010.

[LCH07] C.K. Leung, C.L. Carmichael, and B. Hao. Efficient mining of frequent patterns from uncertain data. In *Proceedings of the Seventh IEEE International Conference on Data Mining Workshops(ICDMW)*, pages 489–494, Omaha, NE, USA, 2007.

[LCZ05] Z. Li, Z. Chen, and Y. Zhou. Mining block correlations to improve storage performance. *ACM Transactions on Storage (TOS)*, 1(2):213–245, 2005.

[Leu04] C.K. Leung. Interactive constrained frequent-pattern mining system. In *Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS)*, pages 49–58, Coimbra, Portugal, 2004.

[Leu09a] C.K. Leung. Anti-monotone constraints. *Encyclopedia of Database Systems*, page 98, 2009.

[Leu09b] C.K. Leung. Monotone constraints. *Encyclopedia of Database Systems*, page 1769, 2009.

[LH09] C.K. Leung and B. Hao. Mining of frequent itemsets from streams of uncertain data. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, pages 1663–1670, Shanghai, China, 2009.

[LHB10] C.K. Leung, B. Hao, and D.A. Brajczuk. Mining uncertain data for frequent itemsets that satisfy aggregate constraints. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC)*, pages 1034–1038, Sierre, Switzerland, 2010.

[LIC08] C.K. Leung, P.P. Irani, and C.L. Carmichael. WiFIsViz: effective visualization of frequent itemsets. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM)*, pages 875–880, Pisa, Italy, 2008.

[LK06a] C.K. Leung and Q.I. Khan. DSTree: A tree structure for the mining of frequent sets from data stream. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM)*, pages 928–933, Hong Kong, China, 2006.

[LK06b] C.K. Leung and Q.I. Khan. Efficient mining of constrained frequent patterns from streamsv. In *Proceedings of the 10th International Database Engineering and Applications Symposium (IDEAS)*, pages 61–68, Delhi, India, 2006.

[LKH06] C.K. Leung, Q.I. Khan, and B. Hao. Distributed mining of constrained patterns from wireless sensor data. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT) Workshops*, pages 248–251, Hong Kong, China, 2006.

[LLN02] C.K. Leung, L.V.S. Lakshmanan, and R.T. Ng. Exploiting succinct constraints using FP-trees. *ACM SIGKDD Explorations*, 4(1):40–49, 2002.

[LLN03] L.V.S. Lakshmanan, C.K. Leung, and R.T. Ng. Efficient dynamic mining of constrained frequent sets. *ACM Transactions on Database Systems (TODS)*, 28(4):337–389, 2003.

[LMB08] C.K. Leung, M.A. Mateo, and D.A. Brajczuk. A tree-based approach for frequent pattern mining from uncertain data. In *Proceedings of the 12th Pacific-Asia Conference on Knowledge Discovery and Data mining (PAKDD)*, pages 653–661, Osaka, Japan, 2008.

[LNM02] C.K. Leung, R.T. Ng, and H. Mannila. OSSM: a segmentation approach to optimize frequency counting. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*, pages 583–592, San Jose, CA, USA, 2002.

[LWW$^+$10] X. Liu, X. Wu, H. Wang, R. Zhang, J. Bailey, and K. Ramamohanarao. Mining distribution change in stock order streams. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE)*, pages 105–108, Long Beach, CA, USA, 2010.

[NKC$^+$06] W.K. Ngai, B. Kao, C.K. Chui, R. Cheng, M. Chau, and K.Y. Yip. Efficient clustering of uncertain data. In *Proceedings of the Sixth IEEE International Conference on Data Mining (ICDM)*, pages 436–445, Hong Kong, China, 2006.

[NLHP98] R.T. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 13–24, Seattle, WA, USA, 1998.

[PG09] A.K. Poernomo and V. 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)*, pages 697–706, Paris, France, 2009.

[PHL01] J. Pei, J. Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE)*, pages 433–442, Heidelberg, Germany, 2001.

[PHM00] J. Pei, J. Han, and R. Mao. CLOSET: an efficient algorithm for mining frequent closed itemsets. In *Proceedings of the 2000 ACM SIGMOD*

Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD), pages 21–30, Dallas, TX, USA, 2000.

[PLL+10] M. Plantevit, A. Laurent, D. Laurent, M. Teisseire, and Y.W. Choong. Mining multidimensional and multilevel sequential patterns. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(1):article No. 4, 2010.

[VRH+09] P.B. Volk, F. Rosenthal, M. Hahmann, D. Habich, and W. Lehner. Clustering uncertain data with possible words. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, pages 1625–1632, Shanghai, China, 2009.

[WF06] R.C.W. Wong and A.W.C. Fu. Mining top-k frequent itemsets from data streams. *Data Mining and Knowledge Discovery*, 13(2):193–217, 2006.

[Yan04] G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 344–353, Seattle, WA, USA, 2004.

[YLL08] J.X. Yu, Z. Li, and G. Liu. A data mining proxy approach for efficient frequent itemset mining. *The International Journal on Very Large Data Bases (VLDB Journal)*, 17(4):947–970, 2008.

[ZLY08] Q. Zhang, F. Li, and K. Yi. Finding frequent items in probabilistic data. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 819–832, Vancouver, BC, Canada, 2008.