Mining Frequent Patterns from Uncertain Data with MapReduce

by

Yaroslav Hayduk

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

Master of Science

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

March 2012

Thesis advisor
Dr. Carson K. Leung

Author
Yaroslav Hayduk

Mining Frequent Patterns from Uncertain Data with MapReduce

Abstract

Frequent pattern mining from uncertain data allows data analysts to mine frequent patterns from probabilistic databases, within which each item is associated with an existential probability representing the likelihood of the presence of the item in the transaction. When compared with precise data, the solution space for mining uncertain data is often much larger due to the probabilistic nature of uncertain databases. Thus, uncertain data mining algorithms usually take substantially more time to execute. Recent studies show that the MapReduce programming model yields significant performance gains for data mining algorithms, which can be mapped to the map and reduce execution phases of MapReduce. An attractive feature of MapReduce is fault-tolerance, which permits detecting and restarting failed jobs on working machines. In this M.Sc. thesis, I explore the feasibility of applying MapReduce to frequent pattern mining of uncertain data. Specifically, I propose two algorithms for mining frequent patterns from uncertain data with MapReduce.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This thesis would not have been possible without the love and support of my family. My mother Nina, my father Bohdan, my sister Annie, and my niece Dianka have given me their unconditional love and support throughout my M.Sc. program. I have also had the extraordinary fortune to have been living with the Quesnel family here in Winnipeg, which was always much too fun! This thesis would have also been impossible without their continued love and support that I'll forever hold close to my heart.

A special big thank you goes to my first English teacher Bohdana Semenivna, who dedicated her every effort to teach me how to spell properly. Now, some fifteen years later, each time I go about spelling the word *ubiquitous*, I remember you.

I also would like to thank the members of my thesis examination committee—Dr. Wai-Keung Fung and Dr. Peter C.J. Graham—for giving me valuable comments and suggestions on my thesis and thank Dr. Pourang P. Irani for chairing my thesis defense.

Last, but surely not the least, I want to thank my supervisor, Dr. Carson K. Leung, who has made sure that I most certainly carry my M.Sc. thesis project all the way through to the defense in this *uncertain* world.

Yaroslav Hayduk
B.Sc.(Honours), Lviv Polytechnic National University, Ukraine, 2009

The University of Manitoba
March 2012

To Tanich'ka!

Chapter 1

Introduction

As frequent pattern mining [AS94] discovers frequently occurring sets of items (hereafter referred to as *frequent patterns*) from data, it permits data analysts to determine interesting correlations in the dataset being examined. Previously, most of the research effort in the area of frequent pattern mining was concentrated on mining frequent patterns from traditional databases of *precise data* such as shopping market basket data. Here, every transaction item is guaranteed to exist.

In this M.Sc. thesis, I primarily focus on mining frequent patterns from databases having *uncertain data*. For uncertain data mining, we can suspect, but cannot guarantee, the presence or absence of an item. For example, a doctor is 90% certain (but cannot guarantee) that a patient suffers from a particular disease. The uncertainty associated with each item can be expressed in terms of existential probabilities, which indicate the likelihood of the presence or existence of an item. The problem of mining frequent patterns from uncertain datasets is nontrivial as the input datasets can be large and contain many unique items, having different existential probabilities. Here, the search space is significantly larger than

that when mining patterns from datasets having precise data.

Briefly, to find frequent patterns for a given dataset, a user specifies the **minimum support** threshold, which denotes the percentage of transactions in the dataset containing a pattern for that pattern to be frequent. In uncertain data mining, to account for the existential probabilities associated with items, a new measure called the **minimum expected support**, was introduced by Chui et al. [CKH07] to act as a threshold for mining uncertain datasets. The expected support of an itemset specifies the sum of the expected probabilities of an itemset in each of the transactions in the database.

Commonly used algorithms for discovering patterns from uncertain and precise data can be broadly divided into three categories, such as candidate generate-and-test algorithms (e.g., Apriori [AS94], U-Apriori [CKH07]), hyper-structure algorithms (e.g., H-Mine [PHL+01], UH-Mine [ALWW10]), and tree-based pattern growth algorithms [HPY00]. Among them, tree-based algorithms, which operate entirely in main memory and avoid expensive disk accesses, are usually faster. This is why I am focusing on them.

Examples of tree-based algorithms include FP-growth [HPY00], UF-growth [LMB08], and UFP-growth [CGG10]. The **FP-growth** algorithm, which mines precise data, (1) represents the compressed input database in the form of a tree, (2) traverses that tree in a depth-first manner to read-off frequent patterns and their corresponding supports, and (3) recursively builds sub-trees (commonly referred to as *conditional trees*) and reads frequent pattern from them.

The **UF-growth** algorithm retrofits the FP-growth algorithm, allowing it to directly work with items' existential probabilities (i.e., it directly works with uncertain data). To reduce the memory consumption, this approach, however, requires the probabilities of items in the initial database to be rounded to (commonly) two decimal places. By rounding

probabilities, we limit the possibly infinite number of expected support values to 10^2 values, in the range of $(0, 1]$.

The **UFP-growth** uncertain data mining algorithm adds a pre- and a post-processing step to the FP-growth precise data mining algorithm. For each transaction in the uncertain database, the pre-processing step samples transaction items according to their existential probabilities. In essence, the pre-processing step converts an *uncertain database* to a larger database, containing *precise data*, which can be attacked by algorithms mining precise data. At the end of the mining process, the post-processing step computes the average expected support of patterns over all samples. Patterns with an average expected support value meeting the user-specified threshold are considered frequent. This technique is straightforward because it employs existing algorithms used for mining precise data to mine uncertain data.

Both of the discussed approaches for mining uncertain data differ principally in terms of arriving at the final result. The UF-growth algorithm modifies the FP-growth algorithm such that, at each execution step, UF-growth performs numerous, potentially slow, floating point calculations. Moreover, the UF-tree, which represents an in-memory compressed version of the uncertain database, achieves lower compressions ratios than the FP-tree used in the FP-growth algorithm. On the contrary, the UFP-growth algorithm avoids modifying the FP-growth algorithm by using a sampling technique at the cost of converting the initial uncertain dataset to a larger dataset containing precise data. In addition, while the UF-growth algorithm always produces accurate results, the accuracy of the UFP-growth algorithm depends on the number of samples taken.

1.1 Problem Definition

In general, algorithms that mine patterns from uncertain data take more time to execute than their precise counterparts due to the probabilistic nature of the dataset. In addition, when the input dataset is large, the tree representing the given dataset often does not fit in the main memory. This calls for *distributed mining of uncertain datasets*.

When looking at the prominent parallel systems, which are currently used by researchers to process large datasets, a noteworthy example is the *MapReduce* programming model [DG08], which is currently enjoying a resurgence of popularity. The two biggest strengths of MapReduce are (i) the fault-tolerance feature as well as (ii) the ability to parallelize computations on large datasets. Recent research [LWZ+08] shows that MapReduce was successfully used for mining frequent patterns from precise data. Given this result, I chose to investigate the feasibility of using the MapReduce programming model to parallelize the execution of UF-growth and UFP-growth. Moreover, to further speed-up the mining process, I investigated the feasibility of combining the use of the MapReduce programming model and the ForkJoin framework, which allows us to distribute the mining tasks among the processing cores of a multi-core processor.

For the sake of clarity, let us briefly overview what ForkJoin and MapReduce are. **ForkJoin** [Lea00] is a framework for distributing work between cores of a multi-core processor on a single computing node. It is built around the producer-consumer principle, where the producers generate tasks and consumers work on them. In contrast to the parallel frameworks which use a single work queue for maintaining tasks, the ForkJoin framework maintains an intrinsic queue bound to each thread. The use of multiple queues avoids expensive locking operations required when accessing a single queue by either consumer or

producer threads. An additional benefit of using ForkJoin is that, when a thread exhausts its work queue, it can become a *thief*, meaning that it can randomly select a thread from a pool of available threads and tries to steal a task from the tail of the queue of the chosen thread.

MapReduce, on the other hand, is a programming model that allows us to process large amounts of data in parallel using relatively inexpensive commodity hardware. Currently, the most widely-used approaches for implementing parallel algorithms are through the use of MPI [MPI12] and OpenMP [Ope12] libraries. They provide message-passing capabilities and access to shared memory, respectively, by multiple threads. Both libraries require programmers to explicitly control the inter-process communication and synchronization. Moreover, algorithm implementers often have to carefully analyze the program code to avoid concurrency issues such as livelocks, deadlocks, and race conditions. These concurrency issues are challenging to analyze and debug because each possible execution of the concurrent program is difficult to reproduce. Conversely, in MapReduce, the programmer does not need to explicitly manage the inter-process communication. Instead, MapReduce provides the capacity to define computation tasks as user-defined jobs containing the "map" and "reduce" functions, which operate on portions of the dataset *independently*. In some cases, a combiner function is useful. This function is often defined in the same way as the reducer function.

One of the most attractive features of MapReduce is fault-tolerance, which automatically detects and reschedules failed jobs. For example, when a failure occurs in MPI and OpenMP environments, to recover from it, we normally need to restart the whole computation from the beginning. As the datasets used in frequent pattern mining can be quite large, it would be highly inefficient to restart the mining process.

As data mining algorithms commonly process large datasets, researchers started to apply the MapReduce programming model in the development of data mining libraries. For instance, the latest version of *Apache Mahout* [Apa12c], a MapReduce-based data mining and machine learning library, supports four use cases: recommendation mining, clustering, classification, and frequent itemset mining from *precise data*. Most importantly, Apache Mahout contains a parallel implementation of the FP-growth algorithm, called **Parallel FP-growth (PFP-growth)** [LWZ+08]. The current version of Apache Mahout, however, does not contain algorithms that support the mining of frequent patterns from *uncertain data*.

Although there exists a parallel implementation of the FP-growth algorithm (i.e., PFP-growth), it can only process precise data and cannot handle uncertain data. Further, although the UFP-growth algorithm works with uncertain data, no attempts have been made to execute its steps in parallel. To the best of my knowledge, no research has been done in the area of parallel frequent pattern mining of uncertain data. Given that this work has never been attempted, and given the need to develop distributed versions of uncertain data mining algorithms, the questions that I am aiming to answer in my research are: By using *MapReduce*, can we decompose our problem to multiple smaller problems, which can be attacked by multiple machines, such that each smaller-sized problem can fit in the main memory? Moreover, will such means of distribution speed-up the execution of uncertain data mining algorithms? Also, as MapReduce distributes the work between the computing nodes in the network, will we be able to further divide mining tasks between the processing cores on a multi-core machine?

To answer these questions, I propose (in Chapter 4) parallel versions of two tree-based uncertain data mining algorithms, namely UF-growth [LMB08] and UFP-growth [CGG10],

by adapting them to the MapReduce programming model. In addition, as MapReduce distributes portions of a bigger mining task between machines in the network to capitalize on computing nodes' multi-core facilities, I also employ the ForkJoin Framework [Lea00] to distribute the work between processing cores of a multi-core processor (in Chapter 3).

1.2 Thesis Statement

My **M.Sc. thesis statement** is as follows. To speed up the mining process of uncertain data mining algorithms, I propose to:

(a) adapt two uncertain data mining algorithms, namely UF-growth and UFP-growth, to MapReduce, which will facilitate the construction of smaller-sized trees on distributed machines; and

(b) combine the use of the MapReduce programming model with the use of the ForkJoin framework to further speed-up the mining process; and

(c) develop an optimized technique for building conditional trees.

I will further assess the effectiveness of using MapReduce and ForkJoin in the context of frequent pattern mining of uncertain data. The **key contributions** of my thesis include the following:

- the development of parallel implementations of the UF-growth and UFP-growth algorithms, which handle uncertain data using the MapReduce programming model;
- the application of the ForkJoin framework for frequent pattern mining of precise and uncertain data on multiple processing cores of a multi-core processor; and

- the proposal of a conditional tree building technique, which reduces the overall mining time as well as minimizes memory use.

Since not all algorithms can be mapped to MapReduce in a straightforward manner, one of the **key challenges** for my research was to map the execution steps of UF-growth and UFP-growth to MapReduce as well as to effectively combine the use of both MapReduce and ForkJoin.

I execute my algorithm implementations in the Apache Hadoop [Apa12b] environment using Amazon EC2 [Ama12a] clusters. Amazon EC2 allows us to set up a requested number of nodes in a relatively short time. To verify the correctness of my parallel algorithm implementations, I compared my mining results with the mining results obtained from running the U-Apriori [CKH07] algorithm. Both UF-growth and U-Apriori yielded the same results. On the other hand, because UFP-growth uses a sampling technique, it always arrived at a different high-quality approximation of the results.

1.3 Thesis Organization

Apart from selectively presenting a number of alternative frequent pattern algorithms, Chapter 2 uses a tree-based algorithm to introduce frequent pattern mining of precise and uncertain data. It covers earlier research efforts targeted at speeding up frequent pattern mining algorithms as well as provides an overview of the parallel frameworks and systems used during my research, specifically MapReduce and ForkJoin. To demonstrate the benefits of employing MapReduce in the process of mining frequent patterns from precise data, the chapter also covers the PFP-growth algorithm, which I used as a base for developing my parallel versions of the UF-growth and the UFP-growth algorithms.

Chapter 3 introduces my proposed optimizations for tree-based frequent pattern mining algorithms, which facilitate a faster construction of conditional trees as well as permit exploitation of the multi-core capacities of a processor. The chapter also discusses the benefits and the mechanism of caching FP-trees. Chapter 4 proposes two novel algorithms—the MapReduce versions of UF-growth and the UFP-growth algorithms—for mining frequent patterns from uncertain data on distributed machines using MapReduce and ForkJoin. Chapter 5 provides experimental results describing (a) the benefits of using the proposed optimizations to tree-based algorithms; (b) the benefits of the proposed MapReduce versions of the UF-growth and the UFP-growth algorithm and concludes by (c) comparing both developed approaches. Finally, Chapter 6 presents a summary of this thesis as well as suggests some possible future work.

Chapter 2

Related Work

In this chapter, I first introduce the main concepts behind frequent pattern mining and discuss commonly used algorithms for mining frequent patterns from precise and uncertain data. Then, I discuss two frameworks, namely MapReduce and ForkJoin (which facilitate parallel data processing on distributed machines and processor cores, respectively) and present a number of parallel computing techniques for speeding up frequent pattern mining. Moreover, I also provide a summary of recent work targeted at converting data mining algorithms to MapReduce.

2.1 Frequent Pattern Mining of Precise Data

Before discussing the related work done in the area of executing frequent pattern mining algorithms in parallel, let us introduce the fundamentals of frequent pattern mining of precise and uncertain data. The introduction to traditional frequent pattern mining algorithms in the following sections will help the interested reader to get a better understanding of the forthcoming sections, describing frequent pattern mining of uncertain data.

2.1.1 Key Concepts of Mining Frequent Patterns from Precise Data

As an introduction to frequent pattern mining, researchers commonly use the pattern generate-and-test **Apriori** [AS94] algorithm as part of their discussion. As (1) tree-based algorithms are superior [KKR10] to Apriori-based algorithms and (2) my thesis is focused on tree-based algorithms only, I chose to base my introduction primarily on tree-based algorithms.

Frequent pattern mining was initially introduced by Agrawal et al. [AIS93]. The task of frequent pattern mining is to detect items that commonly co-occur in the input dataset. Let $\mathcal{I}$ be a set of items. A set $X = \{i_1, \ldots, i_j\} \subseteq \mathcal{I}$ is called an *itemset*. An itemset having a single item in it is called a *singleton itemset* or just a *singleton*. A database *transaction* is defined by a tuple $T = (id, I)$, where id is the transaction id and I is an itemset. We consider a universe of m items occurring in transactions. Formally, a transaction database $\mathcal{D}$ over $\mathcal{I}$ is a set of transactions over $\mathcal{I}$. A transaction $T = (id, I)$ contributes to the support (or *supports*) an itemset $X \subseteq \mathcal{I}$, if $X \subseteq I$. The support of an itemset X in $\mathcal{D}$ is the number of transactions in the database $\mathcal{D}$ containing X. A *frequent itemset* or *pattern* is a subset of items, the frequency of which is no less than a user-specified threshold value called the *minimum support* σ, where $0 \leq \sigma \leq |\mathcal{D}|$. If an itemset appears in a large enough portion of the dataset (i.e., large number of transactions in the dataset), it is considered *frequent*. The collection of itemsets, which fall above the minimum support threshold value, is denoted by $\mathcal{F}(\mathcal{D}, \sigma) = \{X \subseteq \mathcal{I} \mid support(X, \mathcal{D}) \geq \sigma\}$.

The overall problem of **frequent pattern mining** is that, given a set of items $\mathcal{I}$, a transaction database $\mathcal{D}$, and a minimum support threshold σ, find $\mathcal{F}(\mathcal{D}, \sigma)$ as well as the

Table 2.1: A sample market basket transaction database

id	Transaction X
10	{beer, chips, wine}
20	{beer, chips}
30	{pizza, wine}
40	{chips, pizza}

Table 2.2: Corresponding frequent patterns

Itemset	Support
{beer}	2
{chips}	3
{pizza}	2
{wine}	2
{beer, chips}	2
{beer, wine}	1
{chips, pizza}	1
{chips, wine}	1
{pizza, wine}	1
{beer, chips, wine}	1

support count of each frequent patter $\mathcal{F}(\mathcal{D}, \sigma)$.

Example 2.1 *Consider the example database in Table 2.1. where* $\mathcal{I} = \{$*beer, chips, pizza, wine*$\}$*. For a minimal support threshold of 1, Table 2.2 shows the mining result.*

Given a set of domain items $\mathcal{I}$, represented in the database, perhaps the simplest approach for finding frequent patterns is to generate all possible $2^{|\mathcal{I}|}$ combinations, or candidate itemsets and count the occurrence of each one, outputting itemsets, which pass the minimum support threshold. For a low value of $|\mathcal{I}|$ (e.g., $0 \leq |\mathcal{I}| \leq 20$), the task of generating all possible combinations is not computationally challenging. But, for larger values of $|\mathcal{I}|$ it can be impractical to compute all the combinations in a reasonable amount of time.

The algorithms in the Apriori-family make use of several optimizations of the aforementioned naïve technique, as well as take advantage of several properties of frequent itemsets.

One of the key properties of frequent itemsets is the *support monotonicity property*. This property ensures, that if an itemset is *infrequent* it *cannot* serve as a prefix for different frequent itemset. More formally, given a transaction database $\mathcal{D}$, let $X, Y \subseteq \mathcal{I}$ be two itemsets. Then,

$$X \subseteq Y \Rightarrow support(Y) \leq support(X),$$

which means, that if an itemset is infrequent, all its proper supersets or extensions are guaranteed to be infrequent. Conversely, if an itemset is frequent, all of its subsets must also be frequent. Recall Table 2.2. If the itemset $\{beer, chips, wine\}$ is frequent, then all its proper subsets, namely $\{beer, chips\}$, $\{beer, wine\}$, $\{chips, wine\}$, $\{chips\}$, $\{wine\}$, $\{beer\}$, must also be frequent.

Despite all the research efforts devoted to improving the performance of the algorithms in the Apriori family, they are always required to generate and test candidate patterns. This scheme results in a significantly large number of database scans, which is very I/O inefficient and degrades the overall performance.

2.1.2 A Tree-Based Approach

Let us consider the **FP-growth** [HPY00] algorithm, which uses a more efficient approach for finding frequent patterns. FP-growth is a depth-first algorithm, which was developed to address the limitations of the first generation Apriori algorithm. FP-growth scans the database in two passes, building a tree-based, compressed, in-memory representation of the input database. The algorithm then traverses the resulting tree, reading off itemsets and constructing new smaller-sized sub-trees recursively. This process effectively eliminates the subsequent database scans, required by Apriori-based algorithms to test candidate itemsets.

Table 2.3: A sample transaction database

ID	Items	(Ordered) Frequent Items
10	a, b, c, d, u	a, b, c, d
20	a, c, q, t, u, g, w, d	a, c, q, t, u, w, d
300	a, j, b, c, i, d	a, b, c, d
400	g, a, n, b	a, b
50	l, a, b, m, c	a, b, c
60	a, b, p	a, b
70	h, a, q, t, u, w	a, q, t, u, w
80	b, c, q, u, t, w, k	b, c, q, t, u, w

FP-growth scans the database the first time to count the support of each unique item, retaining only frequent singleton items, along with their occurrence (or support) counts. These items are then arranged in frequency descending order and inserted into a header table. The next scan compresses the database into an extended prefix tree—a frequent pattern tree called the FP-tree. Apart from storing the (1) *item name*, each tree node contains information about (2) the number of times this sorted prefix of items (up to and including that node item) was encountered (referred to as *the occurrence count*), as well as (3) node *links*, which facilitate efficient tree traversal and provide special means of navigation between nodes, having the same item. For each encountered transaction, the algorithm prunes the infrequent singleton items from it and sorts the leftover items according to the order in the header table. Then, starting at the root node, the algorithm tries to merge the transaction with the nodes in the existing tree, where the same item exists in the child node and the transaction. Otherwise, a new node is created with a count of 1. In other words, if a path in the tree and the sorted transaction contain the same prefix (i.e., the same series of items, without any intermediate nodes), the transaction items can be merged with existing nodes in the tree, increasing their counts by one.

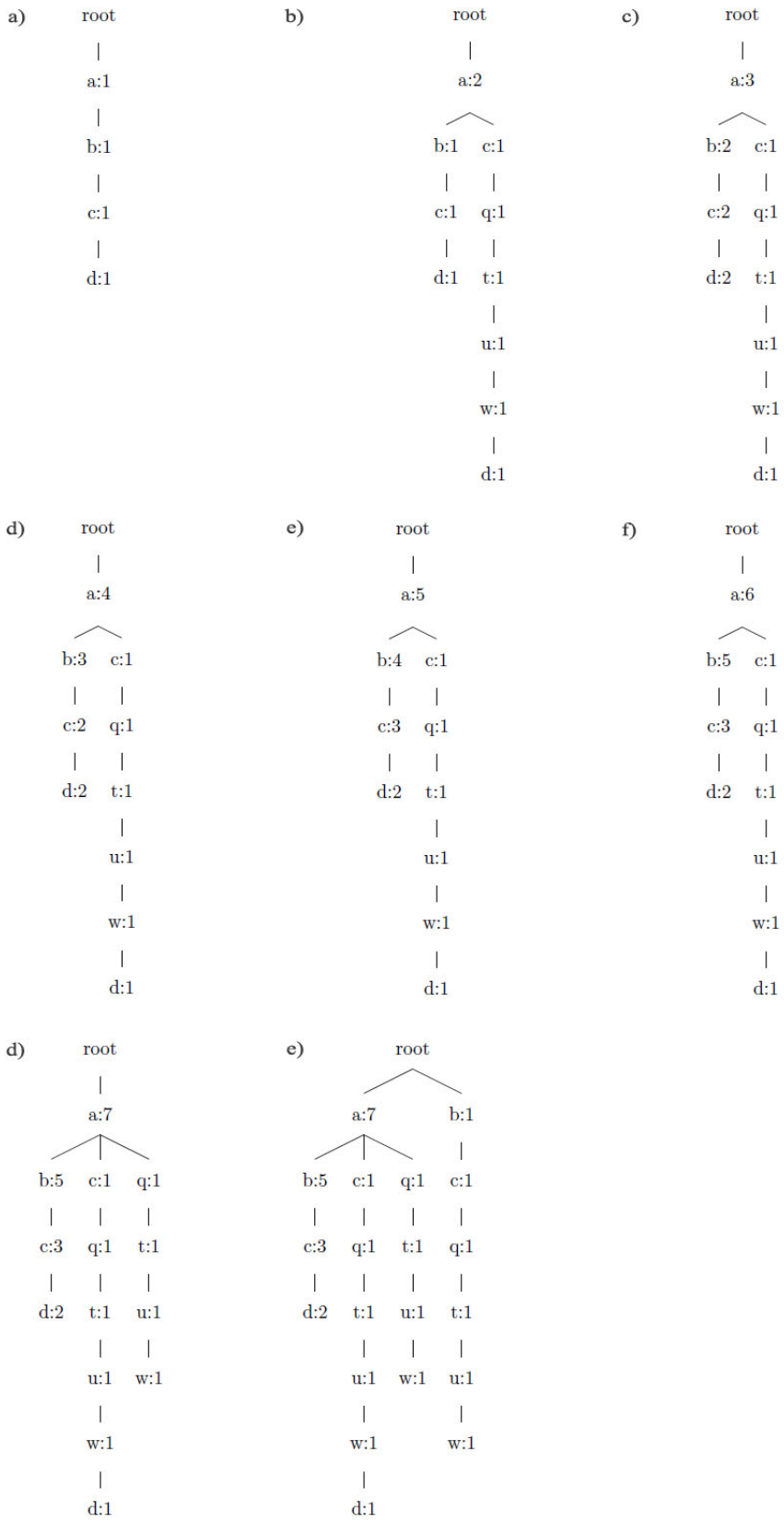

Figure 2.1: The FP-tree construction process

Example 2.2 *Given the transaction database in Table 2.3 and a minimum support of 3, the algorithm builds the FP-tree depicted in Figure 2.1. Here, each step represents the state of the tree after the insertion of each transaction.*

The header table stores all the unique tree items, their corresponding occurrence counts in the tree, as well as maintains the links, connecting all nodes containing the same item with each other. The in-memory representation of the final tree is shown in Figure 2.2. For the sake of clarity, Figure 2.2 demonstrates node links for item d only. In general, to mine patterns from the FP-tree, the FP-growth algorithm recursively builds conditional trees and mines patterns from them, until it can no longer create any new conditional trees.

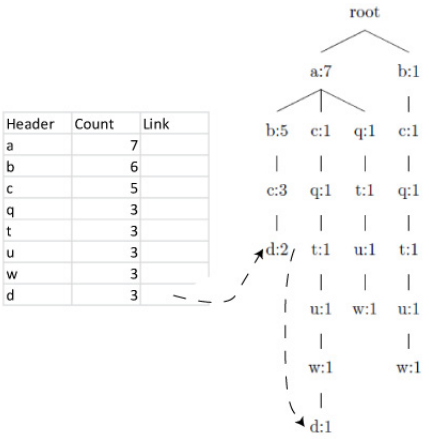

Figure 2.2: Final FP-tree with header links

Let us elaborate on the mining process in more detail. For each item $\{i\}$ in the header table, the algorithm (1) follows its links, (2) traverses the tree bottom-up, (3) extracts all the tree paths, and (4) builds a new conditional tree using the information encountered during the traversal process. At the end of traversing all the links for a given item, the algorithm arrives at a new sub-tree, called an $\{i\}$-*projected tree*, which represents a database $\mathcal{D}^i$ containing only the transactions having item i in them.

During the creation of a projected tree, the algorithm records all unique items encountered, and their accumulated occurrence counts in a new header table. Then, the algorithm prunes the $\{i\}$-*projected tree* to an $\{i\}$-*conditional tree* by traversing the projected tree once again and removing items that become locally infrequent (i.e., whose accumulated occurrence counts fall below the minimum support threshold in the newly-constructed tree). In summary, the first traversal from each node links items i in the main tree and builds a new projected sub-tree. The second traversal visits each node of the projected tree, removes infrequent nodes having infrequent items, and creates a conditional tree. To avoid confusion, in the rest of this thesis, we refer to **projected trees** as trees comprising both (1) the locally frequent and (2) locally infrequent items, and **conditional trees** as trees having **only** the locally frequent items.

For every item in the conditional tree, the algorithm recursively repeats the described procedure (i.e., for each item in the conditional header table, it first creates a projected tree and then prunes it to create a conditional tree). The recursion terminates when there are no more frequent local singletons in the next conditional tree. In essence, the conditional trees created by the FP-growth algorithm represent many smaller-sized non-overlapping databases, which do not have any computational dependencies on each other and thus can be mined independently.

Example 2.3 *Let us continue with Example 2.2. Consider item d in the header table in Figure 2.2. The algorithm follows the d-*links *extracting two paths, namely* $\langle$*c:2, b:2, a:2*$\rangle$ *and* $\langle$*w:1, u:1, t:1, q:1, c:1, a:1*$\rangle$*. The extracted tree paths indicate, that the transactional database contains (1) two patterns containing items a, b, c and (2) one pattern containing items a, c, q, t, u, w. Using these two paths, the algorithm builds the* $\{d\}$*-projected sub-tree. Specifically, the new tree is created using the following two paths:* $\langle$*a:2, b:2, c:2*$\rangle$ *and* $\langle$*a:1,*

c:1, q:1, t:1, u:1, w:1⟩. Notice that the initial bottom-up traversal discovered a node having item a with a count of seven. *However, a appears only* three *times together with d. Hence, all the items in the first and the second extracted paths can contribute a maximum of* two *and* one *item occurrence counts to a new sub-tree, respectively. Figure 2.3 depicts all projected trees extracted from the tree in Figure 2.2.*

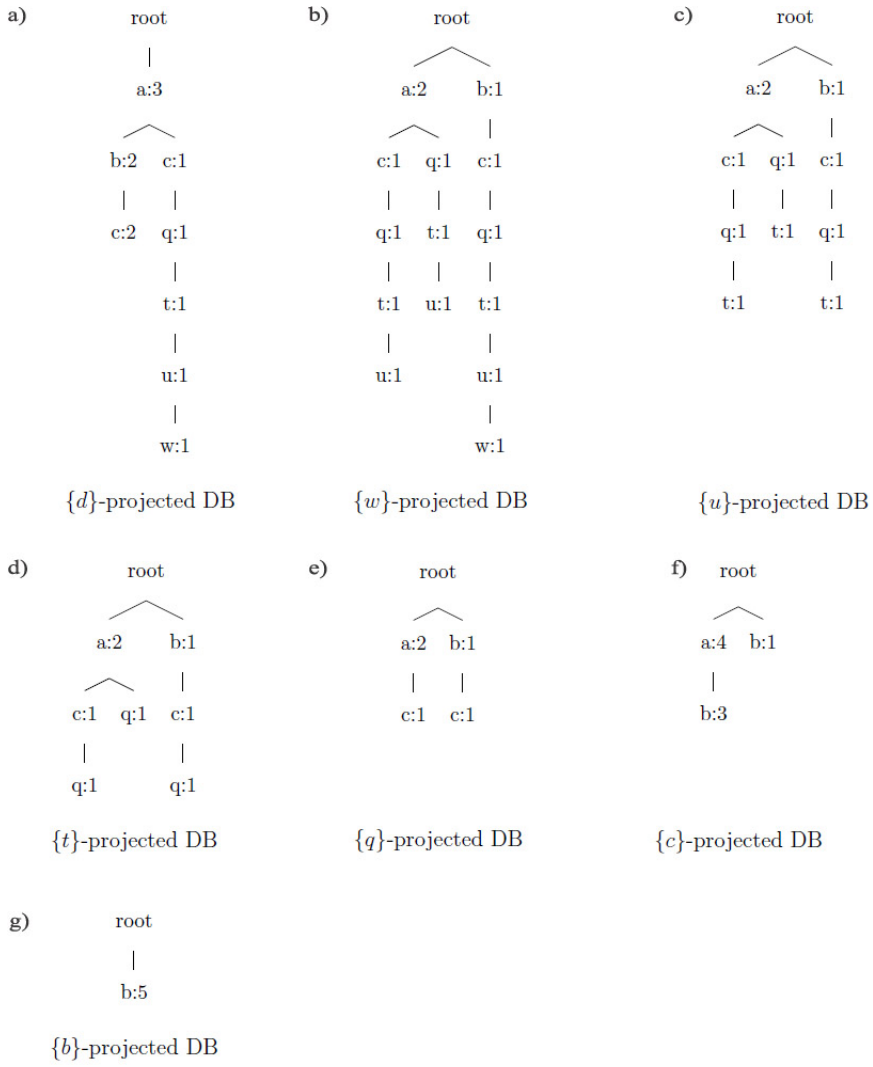

Figure 2.3: Projected FP-trees

Following the previous example discussing the construction of projected trees, *the algo-*

rithm progresses to create conditional trees. *For our* $\{d\}$*-projected tree, the previous traversal recorded* 3 *as the overall count for items* a, c; *2 as an overall count for item* b; *and 1 as the overall count for items* q, t, u, w. *As the minimum support is equal to 3, the algorithm prunes nodes, having items* b, q, t, u, w. *Figure 2.4 contains all conditional trees, constructed from the main tree in Figure 2.2.*

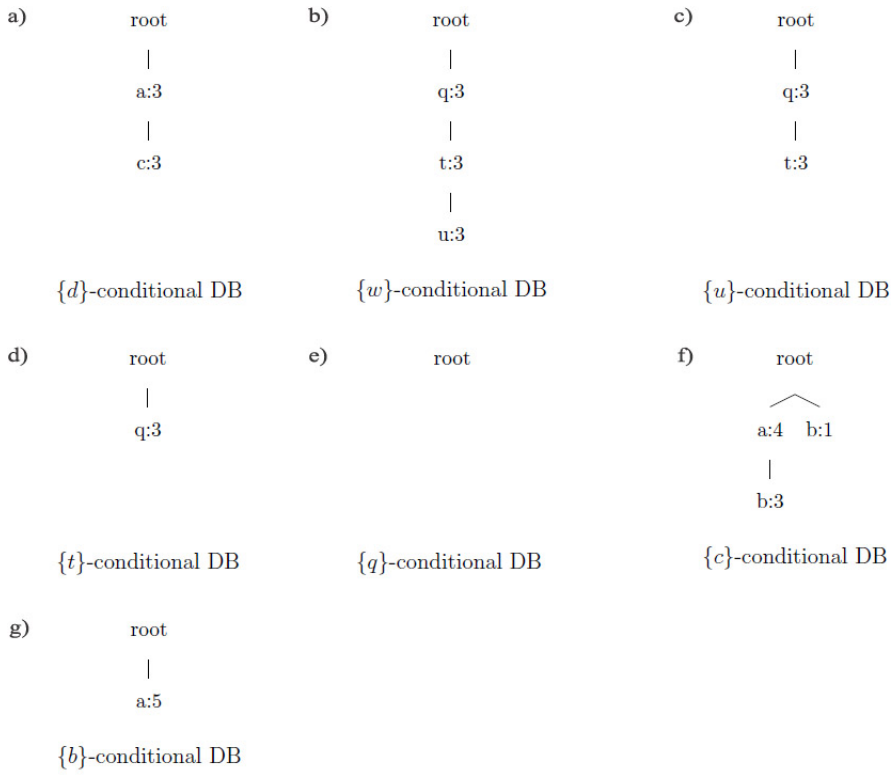

Figure 2.4: Conditional FP-trees

Using the information in the $\{d\}$*-conditional tree, the algorithm can read off two frequent patterns. This is done by concatenating each locally frequent item with its conditional (i.e., in our example with* $\{d\}$*). The support (i.e., the number of times the pattern occurred in the database) of the newly-created pattern is inherited from the count of the local item in the conditional tree. In our example, items* c *and* d *have an overall count of* 3 *in the* $\{d\}$-

conditional tree, hence the algorithm reads off (ad:3) and (cd:3) as frequent patterns in this conditional tree. Recall that the process of creating projected trees, then conditional trees and the reading off of frequent patterns is performed recursively, until the search space is exhausted. In the $\{d\}$-conditional sub-tree example, the algorithm recursively constructs the $\{d, c\}$-conditional tree and outputs (dca:3) as the only pattern. At this point, the algorithm exhausts the search space of item d, and moves on to create and mine a $\{w\}$-conditional tree.

The tree representation of the input dataset takes less space than the original dataset. When the input dataset is dense, the algorithm achieves the best compression rates. For sparse datasets, the constructed tree becomes bushy. Hence, due to little node sharing, the space-saving benefits are less apparent.

2.1.3 Existing Approaches to Mine Frequent Patterns from Precise Data

The survey by Tiwari et al. [TGA08] gives an overview of the current state and the challenging issues related to frequent pattern mining. Besides the commonly used FP-growth [HPY00] algorithm (which I am focusing on in my research), the survey also discussed other frequent pattern mining algorithms, including Apriori [AS94], H-Mine [PHL+01], and Eclat [Zak00].

Kumar et al. [KKR10] provided a thorough comparison of tree-based and Apriori-based algorithms. The Apriori-based algorithms suffer from the level-wise candidate generation-and-test problem, where, at each step, these algorithms generate a huge number of candidate patterns and need several database scans to check whether the candidate patterns are frequent. The number of scans is directly dependent on the maximum length of the candidate

patterns. In contrast, the tree-based algorithms (e.g., FP-growth) perform at most two database scans to build a compressed representation of the database. The major drawback in using tree-based frequent pattern mining algorithms is in that they extensively use memory, which limits their wide application on commodity hardware, where the memory size may be limited.

2.2 Frequent Pattern Mining of Uncertain Data

This section introduces the key concepts of frequent pattern mining of *uncertain data* as well as discusses the most popular algorithms, which mine uncertain data. Also, because this thesis is targeted at executing the UF-growth algorithm in parallel, this chapter discusses the specifics of the UF-growth algorithm.

2.2.1 Key Concepts of Mining Frequent Patterns from Uncertain Data

Frequent pattern mining of uncertain data considers the case, in which each item in the database is associated with an existential probability. Let us consider some possible use cases, of how and where uncertain data can be observed or/and collected.

1. *Privacy.* Prior to the analysis of medical data, artificial noise can be added to it to protect patients' privacy.

2. *Inherent noise in data or data corruption.* Data collected by sensors (e.g., satellite sensors) is never precise, and is represented with a probability of its existence.

3. *Medical data.* When a doctor is uncertain about the type of illness that the patient has, he can express his observations with existential probabilities (e.g., cold:70%; flu:30%).

4. *Data aggregation by customer.* Instead of analyzing certain items per transaction, analysts can analyze the probability of a customer buying a product, creating estimated purchase probabilities, which are then represented with datasets having uncertain data.

In contrast to frequent pattern mining of uncertain data, frequent pattern mining of precise data algorithms work exclusively with binary data, where each item is either present in, or absent from a transaction. Hence, for precise data mining algorithms to work with uncertain data, the data needs to be converted/instantiated to a binary data model.

This conversion is commonly carried out by extra data pre-processing steps, which are executed before running the mining algorithm. Given a predefined threshold, (1) the *quantization step* converts the items' probabilities into 1 (present) or 0 (absent), and removes the absent items from further processing. Then, (2) the *conversion step* removes itemsets having marginally low probabilities. When the threshold value is incorrectly set too high, the conversion step may create a problem for datasets having many items with low probabilities, because it can potentially remove many items, distorting the final mining result.

It is also worth noting that the quantization step instantiates only one possible world from the total of 2^m possible worlds, where m is the number of distinct items in the dataset. Briefly, the possible world interpretation of uncertain data [CKH07] means that, for each item x in the dataset, there exist two possible worlds: One where x exists, and another where x does not exist. For example, given two items a and b in a transaction t_i, there are four possible worlds: (1) both a and b in t_i, (2) a in t_i but b not in t_i, (3) a not in t_i but b in t_i, and (4) neither a nor b in t_i.

Algorithms for frequent pattern mining of uncertain data address the aforementioned problems by working directly with probabilistic data. In uncertain data mining, the occurrence frequency of an itemset is defined differently from traditional frequent pattern mining and is captured by *the expected support* threshold. The expected support of an itemset in a database DB is obtained from the following equation [DYM+10]:

$$expSup(X) = \sum_{i=1}^{|DB|} \left(\prod_{x \in X} P(x, t_i) \right), \tag{2.1}$$

where $P(x, t_i)$ denotes the existential probability of item x, occurring within the itemset X in transaction t_i. Hence, the algorithms, which mine uncertain data directly, account for items with both marginally low and high probabilities.

2.2.2 The UF-growth Algorithm

Leung et al. [LMB08] proposed a tree-based equivalent of FP-growth [HPY00] for mining uncertain data, called **UF-growth**, which modifies the FP-growth algorithm in terms of how the transaction tree is built. Recall that FP-growth uses a tree-based data structure, called the FP-tree, to store a compact representation of the transaction database, which contains frequency information for all frequent items. The FP-tree, however, does not store existential probabilities associated with items. During the first scan, UF-growth finds all the frequent singleton items by recording the accumulated support values for each item in the transaction database and pruning the items below the minimum support threshold. To construct the header table, the algorithm sorts the frequent singletons in descending order of expected support. During the second scan, the algorithm builds the main UF-tree (a variant of the FP-tree) which facilitates the storage of probabilistic information.

Table 2.4: A sample uncertain transaction database

ID	Items
10	f:1.00, a:0.75, c:0.90, d:0.12, g:0.08, i:0.13, m:0.62, p:0.60
20	a:0.75, b:0.60, c:0.90, f:1.00, l:0.21, m:0.10, o:0.18
300	b:0.15, f:0.95, h:0.08, j:0.06, o:0.02
40	b:0.50, c:0.50, k:0.25, s:0.15, p:0.05
50	a:0.70, f:0.95, c:0.70, e:0.25, l:0.01, p:0.45, m:0.50, n:0.11

Table 2.5: A sample uncertain transaction database: sorted and pruned

ID	(Ordered) Frequent Items
10	f:1.00, c:0.90, a:0.75, m:0.62, p:0.60
20	f:1.00, c:0.90, a:0.75, b:0.60, m:0.10
30	f:0.95, b:0.15
40	c:0.50, b:0.50, p:0.05
50	f:0.95, c:0.70, a:0.70, m:0.50, p:0.45

Each node in the UF-tree stores (1) the item name, (2) *expected support of the item* as well as (3) the number of times such an item appeared together with such expected support (i.e., node count). When the algorithm creates the main tree, it merges the transaction items with the nodes in the tree when the item-probability pairs of a node match with the item-probability pairs in the database transaction.

Example 2.4 *Table 2.4 presents an example of an uncertain database. Given a minimum support threshold of 1.00, the algorithm scans the database for the first time, and, for items* f, c, a, b, m, p, e, k, l, o, s, i, d, n, g, h *and* j *accumulates* 3.90, 2.30, 2.20, 1.25, 1.22, 1.10, 0.25, 0.25, 0.22, 0.20, 0.15, 0.13, 0.12, 0.11, 0.08, 0.08 *and* 0.06 *as the corresponding commutative support counts. Table 2.5 contains a filtered representation of the uncertain database, where each transaction contains frequent items only, which are sorted according to the non-ascending frequency order. Note, that to increase the chance of path sharing, each node contains the rounded value (i.e., up to two decimal places) of the expected support.*

After the algorithm inserts the first transaction into the tree, the tree contains one tree

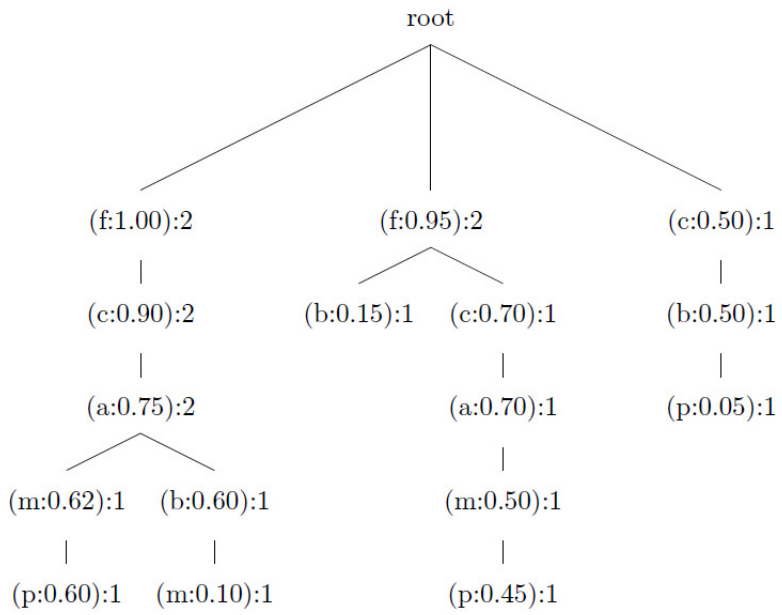

Figure 2.5: A UF-tree

branch, being $\langle$*(f:1.00):1, (c:0.90):1, (a:0.75):1, (m:0.62):1, (p:0.60):1*$\rangle$*. Moving on to the second transaction, the UF-growth algorithm merges items* f, c, a *from the current transaction and the first three nodes in the first branch of the existing UF-tree, as their expected supports are the same, resulting in* $\langle$*(f:1.00):2, (c:0.90):2, (a:0.75):2*$\rangle$*. Items* $\langle$*b:0.60,m:0:0.10*$\rangle$ *cannot be merged with any existing items in the tree; hence, new nodes are created for them and inserted after the* $\langle$*f(1.00):2, c(0.90):2, a(0.75):2*$\rangle$ *prefix. The tree construction process continues in the same fashion for the subsequent transactions. In the end of this process, the algorithm builds a tree, presented in Figure 2.5.*

Although the structure of the UF-tree as well as its construction is similar to that of the FP-tree, there are subtle differences in mining both trees. Recall that, to build conditional and projected sub-trees, the FP-growth algorithm had to maintain the occurrence of X only; in UF-growth we need to maintain the expected support value as well as the occurrence count

of X. Further, to compute the expected support of $X \cup y$ (i.e., the extension to pattern X, where y is an item), we need to multiply the expected support of X by the expected support of each y encountered while traversing the tree path.

Example 2.5 *Let us continue with Example 2.4. To create the* $\{p\}$*-projected tree, UF-growth extracts three tree paths, namely* $\langle$*(f:1.00):1, (c:0.90):1, (a:0.75):1, (m:0.62):1*$\rangle$ *occurring with the conditional probability of* 0.60, $\langle$*(f:0.95):1, (c:0.70):1, (a:0.70):1, (m:0.50):1*$\rangle$ *occurring with the conditional probability of* 0.45 *and* $\langle$*(c:0.72):2, (b:0.90):2*$\rangle$ *occurring with the conditional probability of* 0.05*. The existential support of* $\{p, m\} = 0.60 \times 1 \times 0.62 + 0.45 \times 1 \times 0.50 < 1$, $\{p, a\} = 0.60 \times 1 \times 0.75 + 0.45 \times 1 \times 0.70 < 1$, $\{p, c\} = 0.60 \times 1 \times 0.90 + 0.45 \times 1 \times 0.70 + 0.05 \times 1 \times 0.50 < 1$, $\{p, b\} = 0.05 \times 1 \times 0.50 < 1$ *and* $\{p, f\} = 0.60 \times 1 \times 1.00 + 0.45 \times 1 \times 0.95 = 1.0275$*; as the existential support of* $\{p, m\}$, $\{p, a\}$, $\{p, c\}$ *and* $\{p, b\}$ *are below the minimum support threshold, they are pruned from the tree, creating a* $\{p\}$*-conditional tree. Figure 2.6(a) and Figure 2.6(b) represent the* $\{p\}$*-projected and* $\{p\}$*-conditional trees respectively.*

A detailed analysis of the FP-growth and UF-growth algorithms shows, that UF-growth requires both the item name and its corresponding existential probability to match to merge nodes. In fact, the UF-growth algorithm arrives with a UF-tree, having a much lower compression ratio than in the FP-tree, constructed by the FP-growth algorithm. While the complexity analysis of the FP-growth and the UF-growth algorithms is out of scope of this thesis, interested readers may refer to the paper by Kosters et al. [KPP03] for more information on this topic.

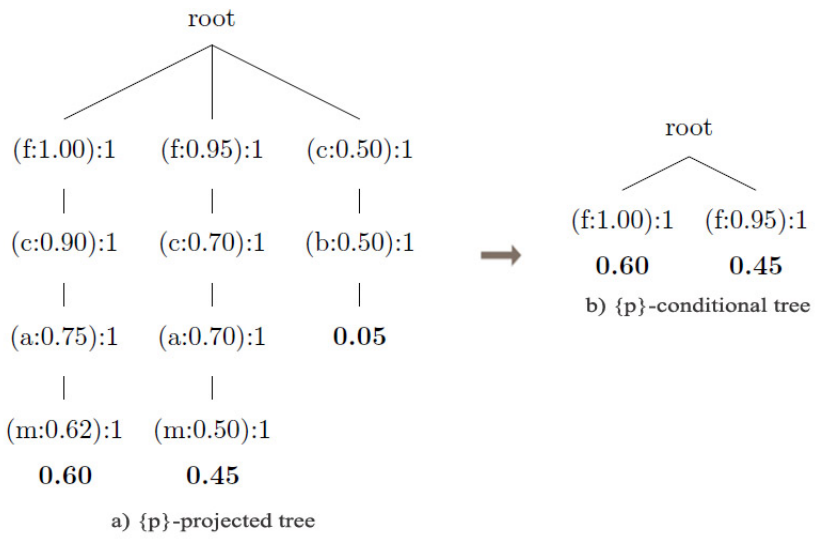

Figure 2.6: UF-trees

2.2.3 Existing Approaches to Mine Frequent Patterns from Uncertain Data

To mine frequent patterns from uncertain data, researchers developed a wide variety of algorithms, which are, in their great majority, based on the frequent pattern mining algorithms of *precise* data. A survey by Leung [Leu11] contains a comprehensive review of uncertain data mining algorithms, which include U-Apriori [CKH07], UF-growth [LMB08], UFP-growth [CGG10], UH-Mine [ALWW10], U-Eclat [CGG10], and UV-Eclat [LS11].

The **U-Apriori** algorithm inherits the same performance bottlenecks as Apriori-based algorithms. I therefore did not work with that algorithm in my research. Instead, I developed parallel versions of the tree-based UF-growth and UFP-growth algorithms, which do not incur the performance overhead associated with generating candidate itemsets.

Calders et al. [CGG10] proposed a sampling approach to mining frequent patterns from uncertain data. They did so by adding a pre-processing step (taking samples) and a post-

processing step (obtaining the real supports of frequent patterns) to the regular pattern mining algorithms and considering them as black boxes. They retrofitted the FP-growth algorithm and created the **UFP-growth** algorithm. To obtain an answer where the support of itemsets is approximated with a 99% probability having less than 1% error, UFP-growth needs to take one sample per transaction for 100,000 frequent items. Note that, given a transaction with m domain items (say, five items), the sample only captures one instance of 2^m (i.e., $2^5 = 32$) possible worlds. To guarantee the approximation quality for 10 times more frequent itemsets (i.e., 1,000,000 itemsets), the algorithm needs to take four samples per transaction. To deal with such data growth, I explored methods targeted at improving the speed of frequent pattern mining algorithms.

2.3 Overview of MapReduce, Hadoop and ForkJoin

Before moving on to discussing the existing approaches for executing frequent pattern mining algorithms in parallel, let me introduce the main concepts behind the MapReduce programming model and the ForkJoin framework.

2.3.1 Overview of MapReduce and Hadoop

This section provides an introduction to MapReduce and ForkJoin, which were the main frameworks that I used for developing parallel versions of UF-growth and UFP-growth.

Dean and Ghemawat [DG08] introduced the fundamentals of the MapReduce programming model, and demonstrated, that it is capable of processing vast amounts of raw data solving a number of distributable problems using relatively cheap commodity hardware. It is best suited for algorithms, which do not require shared state between computing nodes

and are not communication-intensive. Dean and Ghemawat showed how the model could be applied for log processing and distributed grep tasks, as well as data mining and machine learning tasks. Apache Hadoop [Apa12b] is a widely used open-source system for implementing and running MapReduce algorithms.

MapReduce functions work exclusively with $\langle key, value \rangle$ pairs, that is, the input and the output of these functions must be a list of $\langle key, value \rangle$ pairs. The input and output $\langle key, value \rangle$ pairs, however, are not required to be of the same type. The input dataset can be structured, semi-structured, or unstructured. It can also be stored in a database or in a file system. To access the data, Apache Hadoop uses a distributed file system—Hadoop Distributed File System (HDSF), which automatically replicates information on many computing nodes.

The ideas behind MapReduce can be described as follows. The dataset is first divided into chunks (or splits), and each chunk is then processed by a separate mapper. The *mapper functions* convert the $\langle key, value \rangle$ pairs from the initial data domain to $\langle key, value \rangle$ pairs in a different data domain. Afterwards, the system advances to the shuffle-and-sort phases. During these phases, the system retrieves all the data produced by instances of the mapper function and groups values by keys. Different instances of the mapper function can output different values for the same key. Then, the *reducer function* is run for each key as well as the list of values that are mapped to that key. Sometimes, it is useful to define a *reducer function*. The key difference between the reducer and the combiner functions is that the *reducer function* is run against the list of values associated with a specific key from all the mapping instances, while the *combiner* function is run against the values produced by one mapping instance only. Note that the mapper and the reducer functions process the $\langle key, value \rangle$ pairs in parallel. These processes are best illustrated using a MapReduce word count

example, as depicted in Figure 2.7, taken from van Groningen [vG12]. The initial text file is divided into pieces and is distributed among mapper functions, which emit a $\langle word, 1 \rangle$ for each word encountered. Then MapReduce groups values by keys, and executes a reducer functions for each key and its corresponding values. This effectively permits to counts all the 1's (or values, associated with each word) emitted by the mapper functions.

When explaining MapReduce in SQL programming terms, the map function specifies how the initial data grouping is performed (which corresponds to the SQL GROUP-BY clause), and the reduce function reassembles the aggregation function (such as SUM) that is applied to each group. In Apache Pig [Apa12d] terminology, the structure of the MapReduce program can be described as follows:

```
map_result = FOREACH input GENERATE FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
output = FOREACH key_groups GENERATE reduce(*);
```

The first statement parses the input data and applies the user-defined map function. Then, the data is grouped by the key (`$0` denotes the first column of the key). Finally, the system applies the user-defined reduce function for each list of values associated with the key.

The key benefits of the Google MapReduce system are automatic load balancing, fault-tolerance, and the ease of scalability. Briefly, the load balancing feature of MapReduce tries to distribute the work between each computing node evenly, such that each node stays idle as little time as possible. MapReduce achieves fault-tolerance by monitoring nodes' "heartbeat" to check is they are still "alive". Once a node has not responded for a predefined period of time, it is considered "dead"; and the jobs, which were running on that node, are rescheduled

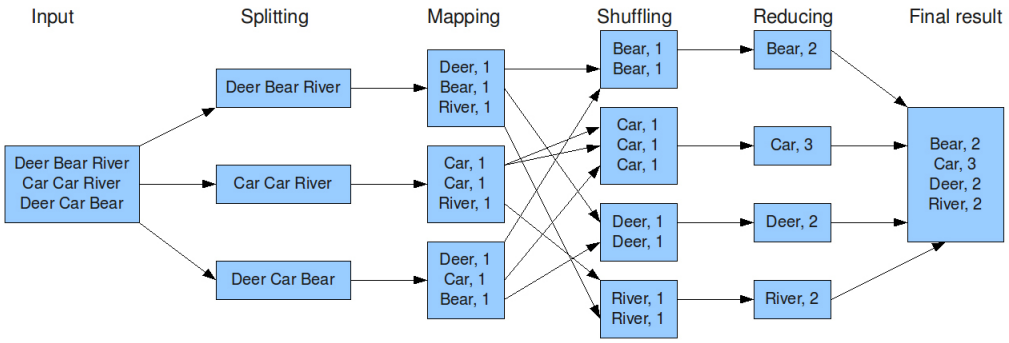

Figure 2.7: MapReduce word count example, from van Groningen [vG12]. ©2009, Search-Workings. Used with permission from Searchworkings Team.

on other working nodes. Lastly, MapReduce contains mechanisms, which allow employing thousands of nodes at the same time for working on a particular task.

2.3.2 Overview of the ForkJoin Framework

The main goal of the **ForkJoin** framework is to split computationally intensive tasks into multiple pieces, which can be performed in parallel independently, to minimize the execution time of the algorithm. ForkJoin is different from MapReduce in that the developer has to explicitly control the work distribution process. The ideas of ForkJoin are not new, and are heavily used in popular parallel processing systems such as OpenMP. In the Java programming language, the framework also uses the concept of *work stealing*, where the thread (which completes the work assigned to it) can *steal* tasks from other threads, assigning ready tasks to idle threads as efficiently as possible.

Each thread maintains a task queue in the form of a double-ended queue, which supports LIFO (Last-In-First-Out) *push* and *pop* as well as FIFO (First-In-First-Out) *take* operations. When the thread completes executing the current task, it fetches the next task from the *head* of the queue. When the queue becomes empty, the thread becomes a *thief*. It selects a

```
Result solve(Problem problem){
       if(problem is small)
         directly solve problem
       else{
         split problem into independent parts
         fork new subtasks to solve each part
         join all subtasks
         compose results from subresults
       }
 }
```

Figure 2.8: Problem decomposition in ForkJoin

different thread at random, and tries to *steal* a task from the *tail* of the queue of the chosen thread.

Most of the ForkJoin algorithms are recursive in nature, and work in a divide-and-conquer manner by splitting subtasks until they can be efficiently solved sequentially. Figure 2.8 demonstrates how a computational problem can be solved with the help of the ForkJoin framework.

Here, the $fork$ function creates new subtasks and the $join$ function pauses the current threads until the forked subtasks complete.

2.4 Parallel Data Mining

To the best of my knowledge, there do not exist any parallel data mining algorithms that mine uncertain data. Hence, this chapter overviews the noteworthy attempts to execute frequent pattern mining of *precise data* algorithms in parallel. Most of the research in this area concentrated on using MPI and OpenMP as the primary parallel processing tools.

2.4.1 Existing Parallel Frequent Pattern Mining Algorithms

The primary disadvantage of tree-based algorithms is that they use memory excessively. Even for relatively smaller-sized datasets, when the minimum threshold is set to a small value, the main tree can grow to billions of nodes, leading to substantial memory consumption. Hence, significant research efforts have been devoted to developing parallel implementations of frequent pattern mining algorithms of *precise data*, which make it feasible to construct and mine smaller-sized trees on multiple machines in parallel. Recall that each conditional tree does not have any computational dependencies on other conditional trees. This property was exploited by a number of parallel implementations of FP-growth.

A noteworthy algorithm developed by Liu et al. [LLZT07] demonstrated a multi-core version of the FP-growth algorithm. Liu et al. enhanced the sequential version of the algorithm, improving data locality. They independently decomposed both, the conditional tree construction routine as well as the mining routine. To eliminate cache misses, associated with the traversal of the FP-tree, a cache-friendly FP-array structure was proposed.

Recent work by Tlili and Slimani [TS11] proposed a method for mining frequent patterns with the use of grid computing. The authors proposed a load balancing technique, which dynamically determines the effectiveness of the potential migration of a given portion of a data mining task. The main limitation of this approach is that it is partially based on the Apriori algorithm. Other research works by Lin and Luo [LL10], for example, proposed a mining algorithm that works in a cloud environment. They constructed the main FP-tree on a single machine, and did not support fault tolerance.

FP-growth using MPI

Javed and Khokhar [JK04] created a parallel version of the FP-growth algorithm in the MPI [MPI12] environment. For each item in the header table of the main tree, the algorithm creates projected trees and sends them to different computing nodes (i.e., worker nodes), which perform the actual mining of those conditional trees. When all the computing nodes complete the mining process, they send the mined patterns back to the main computing node (i.e., master node), which agglomerates the mined patterns and returns the final result to the user. Linear speed-up by their algorithm was reported.

There are a number of *limitations to this algorithm*. The **first limitation** is that the MPI environment does not have inherent fault-tolerance facilities. Hence, when any one node fails during the execution of the algorithm, the whole computation terminates, as MPI cannot recover gracefully from failures. Such faults can be costly (e.g., in an industrial setting), where the input dataset is large and the minimum support threshold is low. Here, even when we use the discussed parallel technique, the mining time can be substantial. The **second limitation** is that, when the number of computing nodes is much smaller than the number of projected trees, the algorithm sends more than one projected tree to each of the computing nodes. For big databases, where each projected tree is large, it is costly to send them over the network to different computing nodes (especially in networks with low bandwidth). Moreover, only one computing node is responsible for building the main tree and sending it to the other worker nodes. Thus, the worker nodes have to stall until they receive the assigned projected tree(s). Such behavior deteriorates the runtime of the algorithm and does not promise linear speedup.

FP-growth using OpenMP

Zheng [Zhe02] proposed an OpenMP version of the FP-growth algorithm. Zheng parallelized the tree construction phase and the mining phase of the algorithm as well as showed that it was scalable. The scalability tests were, however, performed on a limited number of datasets.

2.4.2 Data Mining with MapReduce

Many systems have been developed for running MapReduce jobs. The most widely used of these is the Java-based Apache Hadoop [Apa12b]. Hadoop permits running parallel jobs on inexpensive commodity hardware. It has a built-in fault-tolerance system that automatically reschedules failed jobs. The fault-tolerance feature is especially beneficial in large clusters, and is one of the biggest assets of Apache Hadoop.

Apache Hadoop is used by Apache Mahout [Apa12c], a scalable machine-learning and data mining library, which contains numerous MapReduce algorithm implementations. Unlike WEKA [HFH+09], a different data mining library that contains algorithm implementations capable of executing on only one machine, most of the algorithms implemented as part of Apache Mahout leverage the power of MapReduce for executing algorithms in parallel. The algorithms implemented in the Apache Mahout library include collaborative filtering, user-and item-based recommenders, k-means, fuzzy k-means clustering, mean shift clustering, Dirichlet process clustering, latent dirichlet allocation, singular value decomposition, complementary naïve bayes classifier and a random forest decision-tree-based classifier. Most importantly, the library contains the implementation of the aforementioned PFP-growth algorithm, which I used as a base for developing the MapReduce versions of UF-growth and

UFP-growth.

There are a number of other data mining problems, for which techniques that employ the MapReduce programming model have been used. For example, Corderio et al. [CJT+11] proposed an approach for processing very large datasets to cluster their points. For datasets having billions of points, they could successfully use up to 1,024 cores in parallel. In addition, MapReduce was successfully applied by Zhu and Wang [ZW10] for community mining using genetic algorithms. Here, a variation of MapReduce, called the Chain MapReduce model, was used to do the mining. Zhu and Wang reported significant speedup of their developed parallel algorithm, as opposed to using its sequential counterparts.

Most of the current algorithms in the literature that use MapReduce are, however, not concentrated on frequent pattern mining. Nevertheless, researchers started to develop general guidelines for converting sequential algorithms to MapReduce for various types of algorithms. For example, Chu et al. [CKL+06] developed a set of guidelines for adapting a wide variety of machine-learning algorithms to the MapReduce programming model. Specifically, they authors covered locally-weighted linear regression, k-means, logistic regression, naïve Bayes, support vector machine, independent component analysis, principal components analysis, Gaussian discriminant analysis, expectation maximization, and back-propagation algorithms. This guideline does not concentrate on any particular algorithm, and covers algorithms that fit the Statistical Query Model [Kea93]. The research conducted by Chu et al., however, does not contain guidelines for converting frequent pattern mining of uncertain data algorithms to MapReduce.

FP-growth on MapReduce

Parallel FP-growth (PFP-growth) addresses the limitations of the MPI-based imple-

mentation of FP-growth by employing MapReduce for mining frequent patterns. The key to the runtime efficiency of this approach is that it never builds the main tree on one machine, but partitions the main database such that each machine, which receives a database chunk, can build projected trees locally. PFP-growth performs such partitioning by constructing smaller, disjoint representations of the database on many nodes, which are more likely to fit in the main memory of each node. Hence, in this scenario, no one machine is responsible for (1) building the main tree, (2) building projecting trees and (3) sending these projected trees to different machines. This is beneficial for use-cases, where the main tree cannot fit in main memory. Moreover, the inherent fault-tolerant nature of MapReduce allows recovery from node failures by transparently restarting the computation, which was running on the failed node, on some other working node in the MapReduce network.

Below is a description of the PFP-growth algorithm, specifying the input and the output values of mappers and reducers at each stage:

1. The **Parallel Counting stage** counts the support of singleton itemsets (i.e., frequent itemsets of size 1). Specifically, each time the algorithm locates an item in the transaction, it increments the support count of that item by 1. This stage further sorts frequent singleton itemsets in decreasing frequency order so as to maximize the chance for path sharing in the FP-tree.

   At this stage, during the **mapping phase**, mapper functions receive $\langle key, value = T_i \rangle$ as input, where T_i denotes a unique transaction (note, that the value of key represents the file offset of T_i in the database, and is ignored in the calculations). For each item x in transaction T_i mappers output $\langle key' = x, value' = 1 \rangle$.

   Then, during the **reduction phase**, for each key representing a singleton, MapReduce

automatically collects $S(key)$, representing all values, associated with a key, and feeds it to the reducer in the form of $\langle key = x, value = S(key)\rangle$ as input. Generally, $S(key)$ is represented as a list; as such, we simply calculate the *sum* of all the items in the list to calculate the support of a singleton itemset. In the end, each reducer outputs $\langle key' = x, value' = sum(S(key))\rangle$.

2. The **Grouping stage** splits the list containing frequent singletons into distinct groups and assigns a unique id to each group. The new list, containing group-to-singleton mappings, is called a group list (G-list). The Grouping stage identifies, which conditional trees should be mined together on one computational node. This stage is not computationally intensive, and can run on one machine.

3. The **Parallel FP-growth stage** identifies group-dependent transactions and mines patterns from these transactions. The **mapping phase** identifies all group-dependent transactions, which are then fed to reducer functions. This is done by performing the following steps:

   First, on each machine executing the mapper functions, the algorithm loads the G-list into main memory, and creates a reverse map, mapping singletons to their corresponding group id. Then, each mapper receives $\langle key = gid, value = DB(gid)$ as input. For every transaction T_i in the received list $DB(gid)$, the algorithm a) substitutes all transaction items with their corresponding group ids from the reverse map and creates a new list T (of the same size) as the transaction size. b) For each group-id gid in T, the algorithm locates the right-most appearance L of gid in T, and outputs a new (truncated) transaction, in the form of $\langle key' = gid, value' = \{T_i[1] \dots T_i[L]\}\rangle$.

   Recall that Javed and Khokhar [JK04] used one computational node to create the main

FP-tree, extract conditional trees and send each conditional tree to worker nodes. In MapReduce, however, at any point of time, no one node receives a complete database representation. At the current mapping stage, each node receives a chunk of the database only; hence, for each transaction, mappers identify transaction items, which contribute to the construction of conditional trees for singletons, belonging to a particular group.

Next, at the **reduction phase**, the MapReduce framework agglomerates all the group-dependent transactions and, for each group, assigns a different reducer to work with them. Here, each reducer constructs a sub-tree, and then executes the FP-growth algorithm to perform the mining of frequent patterns for group items.

4. Finally, the **Aggregating stage** aggregates the resulting frequent patterns, produced by the previous stage.

One of the important aspects of the PFP-growth algorithm to note, is that it does not build the main tree on any one given node. As UF-trees are normally larger than FP-trees, the benefits of employing the aforementioned partitioning technique, which facilitates distributed construction and mining of conditional trees, would be even more evident for my proposed parallel versions of UF-growth and UFP-growth discussed in Chapter 4.

Although Li et al. [LWZ+08] reported linear speedup of their proposed PFP-growth algorithm, it may not perform as well for some types of datasets. For example, the maximum number of computing nodes, which can theoretically participate in the mining process, is limited to the number of frequent singleton items. The described behavior is not a problem, however, under the assumption that we have more frequent singletons than machines in the MapReduce network.

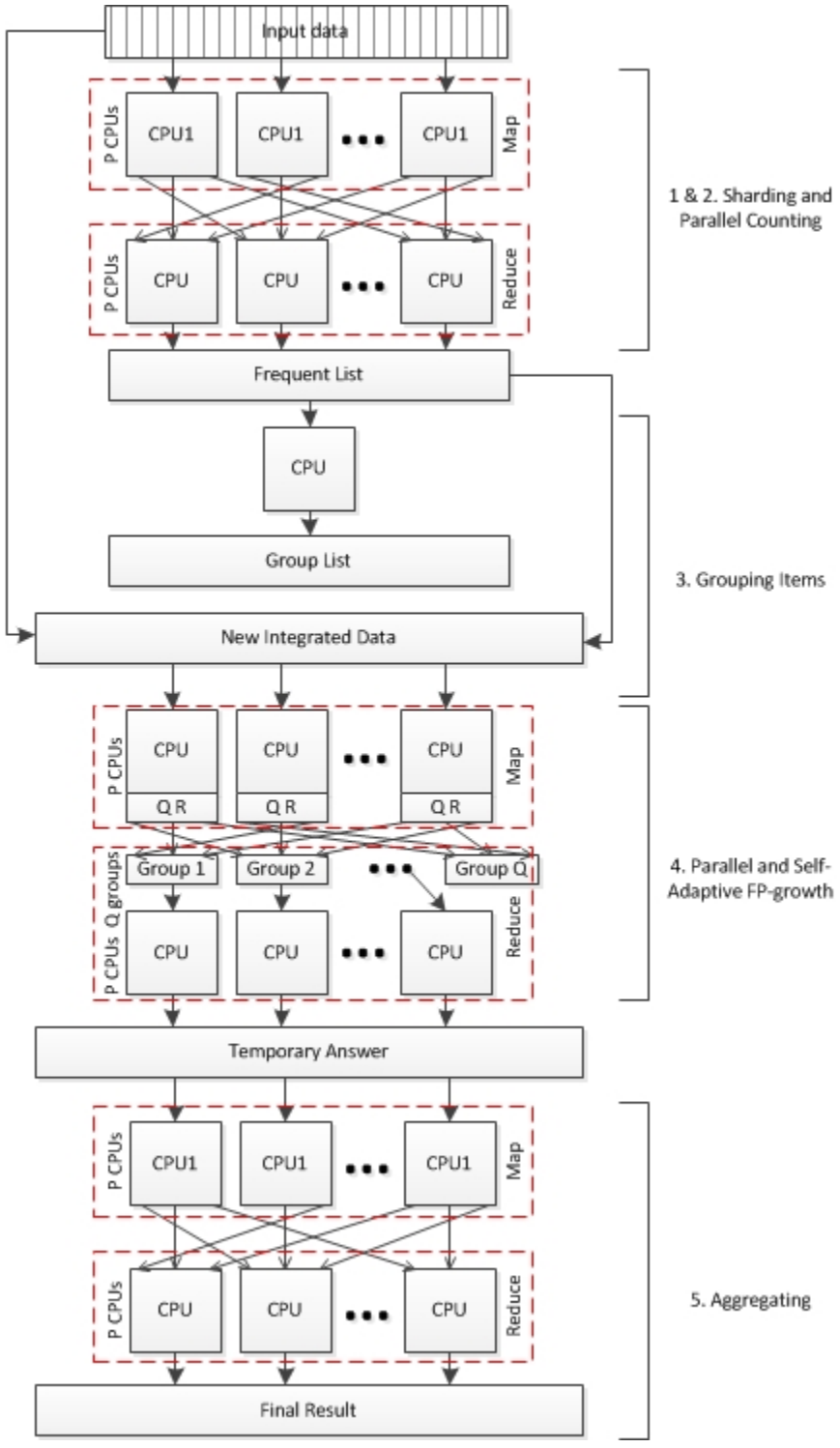

Figure 2.9: Parallel FP-growth execution stages, from Li et al. [LWZ+08]. ©2008, ACM, Inc. Reprinted by a license to reuse in a thesis.

2.5 Summary

This chapter (1) covered the fundamentals of frequent pattern mining of precise and uncertain data and (2) outlined the key algorithms, which are commonly used to target frequent pattern mining tasks. Also, it (3) outlined the current approaches for executing frequent pattern mining algorithms of precise data in parallel.

The approaches, based on the algorithms from the Apriori-family, struggle from the candidate generate-and-test problem, which significantly degrades the performance of the mining algorithms. On the contrary, tree-based approaches avoid generating candidate patterns by constructing a compressed representation of the database, which is stored in a tree-based structure. Tree-based approaches are beneficial in that they do not incur disk access overheads, requiring only two scans of the input database. Nevertheless, when the input dataset is large, the tree cannot always fit into main memory.

Given this limitation, I explored methods, targeted at distributed frequent pattern mining. One of the available approaches was to distribute the mining process between the computing nodes using MPI-based clusters. The limitations of this approach, however, are in that it (1) builds the main tree representation of the database on one computing node, which is not always possible to do, given a large input dataset, and (2) does not provide fault-tolerance compatibilities. In networks that contain a large number of commodity computing nodes, the nodes have a higher probability of failing sporadically. If any given node in the MPI-network fails, the process of recovery is not straightforward, and often involves restarting the whole computation from the beginning.

In response to these challenges, a MapReduce version of the algorithm can be used. MapReduce is a programming model, which was specifically developed to target the pro-

cessing of large datasets on commodity hardware. It provides transparent fault tolerance facilities, which detect node failures and restart computations on working nodes in the network. In addition, I explored the feasibility of using the ForkJoin framework to further divide the mining process of different conditional trees across processor cores in each node.

Overall, the PFP-growth algorithm will serve as a blueprint for developing my proposed uncertain data mining algorithms. The following chapter will discuss a conditional tree construction technique as well as reveal the specifics of using the ForkJoin framework for mining FP-trees in parallel.

Chapter 3

Improvements to Tree-Based Frequent Pattern Mining Algorithms

Recall that one of the issues with the PFP-growth algorithm is that it does not exploit any available multi-core facilities of the MapReduce computation nodes. This chapter discusses my application of the ForkJoin programming model, which takes advantage of processing cores of a multi-core machine, as well as discussing a tree caching technique. Further, I proposes in this chapter an optimization for constructing conditional trees, which reduces memory use as well as the number of node visits. All my optimizations are equally applicable for tree-based algorithms of precise and uncertain data.

3.1 Optimization 1: Exploiting Multi-core Processors in the ForkJoin Framework

Recall that, during the Parallel FP-growth stage of the PFP-growth algorithm, each mapper receives group-dependent transactions in the form $\langle key = gid, value = DB(gid)\rangle$. Using these transactions, the algorithm constructs the group-dependent main tree, which is then used for building and mining conditional sub-trees for each group-dependent item. If a given group is mapped to more than one item, then the algorithm *sequentially* builds and mines each conditional tree, which can be slow for large datasets.

To increase the mining speed, I exploit machines having multi-core processors by using the ForkJoin framework [Lea00]. My improvement performs the following steps:

1. detects the number of processing cores N_{cores} on a multi-core processor;

2. divides the list of group-dependent items G_i into approximately equal parts, such that each thread (running on a different processing core) is responsible for mining $\frac{|G_i|}{N_{cores}}$ items, and inserts them into the queue of each thread;

3. when any given thread finishes constructing and mining conditional trees for all its assigned singletons, it attempts to steal a singleton from the queue of a random thread.

This approach is particularly beneficial in MapReduce infrastructures having limited number of computing nodes. Here, each group id is mapped to many singleton items; hence, each computational node in the cluster is responsible for the construction and the mining of conditional trees for a number of singleton items.

This approach, however, requires more main memory to execute, as multiple conditional trees are built and stored in main memory concurrently. In contrast, when the sequential

FP-growth algorithm finishes the mining process for a given item, it destroys all the sub-trees, which were used for that item, and then starts constructing new conditional trees for a different item. Moreover, recall that the MapReduce infrastructure is commonly built around commodity hardware. As such, not all systems have the processing capacities (i.e., many cores, large main memory), required for the ForkJoin framework to run effectively.

As these two issues in question are hardware-dependent, when newer hardware is used for building a MapReduce cluster, the proposed approach would make use of the available computing power in the most effective way. In summary, MapReduce divides the processing of all conditional trees among the processing nodes in the network, where each node is responsible for mining conditional trees for items that are mapped to a given group id. Then, on a single computational node, the ForkJoin framework employs multiple threads to mine conditional trees for a given items (i.e., distributes conditional trees between threads' task queues). Figure 3.1 shows an example of these processes, where $P1$ and $P2$ represent two different processing cores of a multi-core processor.

Here, the choice to use the ForkJoin framework (which uses Java threads under the hood) for distributing work between different cores of a multi-core processor stems from the fact that the Parallel FP-growth algorithm, which I extended, was implemented in Java as part of the Apache Mahout project.

3.2 Optimization 2: Tree Caching

Recall from Section 2.1.2 that, after mining frequent patterns from any given conditional tree, we dispose of that tree object and then move on to creating an object of the same type again. Another approach for speeding up the mining process of tree-based frequent pattern

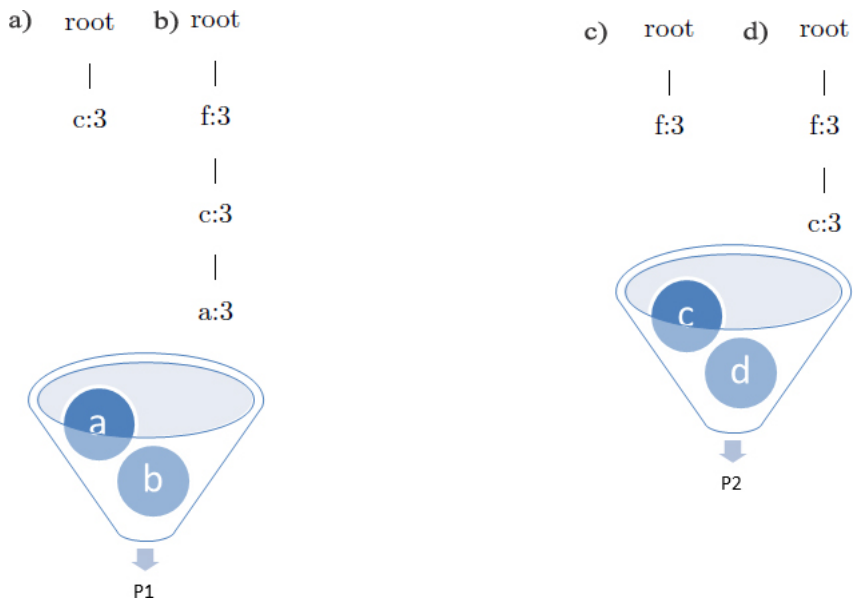

Figure 3.1: Mining conditional trees on multiple processing cores concurrently

mining algorithms is to reduce memory allocation/deallocation.

Before proposing a solution for reducing memory allocations, let us briefly look at the mechanics of Java memory management. In Java, we cannot directly deallocate memory used by any object and have to rely on the garbage collector. Garbage collection is a mechanism provided by the Java Virtual Machine to reclaim the heap space from objects that are eligible for garbage collection. To make an object eligible for garbage collection, we need to set all references to that object to *null* explicitly (e.g., $treeObject = null$). The garbage collector will defer the garbage collection process until we are running low on free memory; the process of garbage collection cannot be started manually.

As we are frequently instantiating and disposing of objects (read-memory), we would get frequent pauses during algorithm execution. When the garbage collector has enough unused tree objects scheduled for disposal, it will execute the garbage collection process, pausing the execution of the FP-growth algorithm. To avoid frequent garbage collections, I propose

to not dispose of trees, which are no longer needed, and store them in main memory.

My implementation of the FP-growth algorithm (and later, the MapReduce versions of the UF-growth and the UFP-growth algorithm) uses a thread-local cache maintained as a stack. The stack maintains a predefined number of cached trees; each thread maintains a different cache stack. When a given thread exhausts its stack, it will try to peek at the bottom of other cache stacks, maintained by different threads, to avoid instantiating an additional tree object. When this process fails, the thread has no choice but to create a new instance of a tree.

Caching, however, uses memory extensively. Hence, to avoid OutOfMemoryExceptions, I used the `java.lang.ref.SoftReference` class for wrapping my tree instances. SoftReferences are commonly used to implement memory-sensitive caches. Specifically, my implementation maintains a cache stack, which contains trees, wrapped in SoftReferences. When the program is running low on memory, the garbage collector will evict objects, wrapped in a SoftReference. Hence, this class provides us with a fail-safe mechanism, which automatically controls the size of our cache. This behavior can be useful in cases, when the cache size was initially set too big.

Although the steps presented here are valid for the Java programming language only, they are equally applicable to any programming language, which supports automatic garbage collection and has an equivalent to SoftReferences.

3.3 Optimization 3: Efficient Conditional Tree Construction

This section discusses the third optimization, which allows constructing conditional trees without constructing projected trees first. Recall that to construct a conditional tree, the FP-growth algorithm first constructed a projected tree and then pruned the locally-infrequent nodes from it to create a conditional tree. As a result, the process required two tree traversals.

Using my optimized approach to constructing conditional trees, the conditional tree is constructed using two traversals of the main tree. The general idea of this process is as follows. The first bottom-up traversal accumulates the counts of all encountered items on the path, flagging all visited nodes. Then, by following the same path top-down again, which is accomplished by recursively visiting the flagged child nodes only, the second scan of the main tree traverses it in a depth-first manner building a conditional tree. Specifically, the algorithm performs the following steps:

1. for each item $\{i\}$ in the header table, traverse the tree bottom-up and count the occurrence of each encountered item on the tree path;

2. if the parent node of the current node has more than one child, flag the current node with item $\{i\}$ (i.e., `childNode.flag`=i);

3. determine which items are locally frequent, and insert them into the new header table, which will be associated with the conditional tree;

4. traverse the tree again in a top down fashion; when a node with multiple children is encountered, visit each child node, flagged with item $\{i\}$;

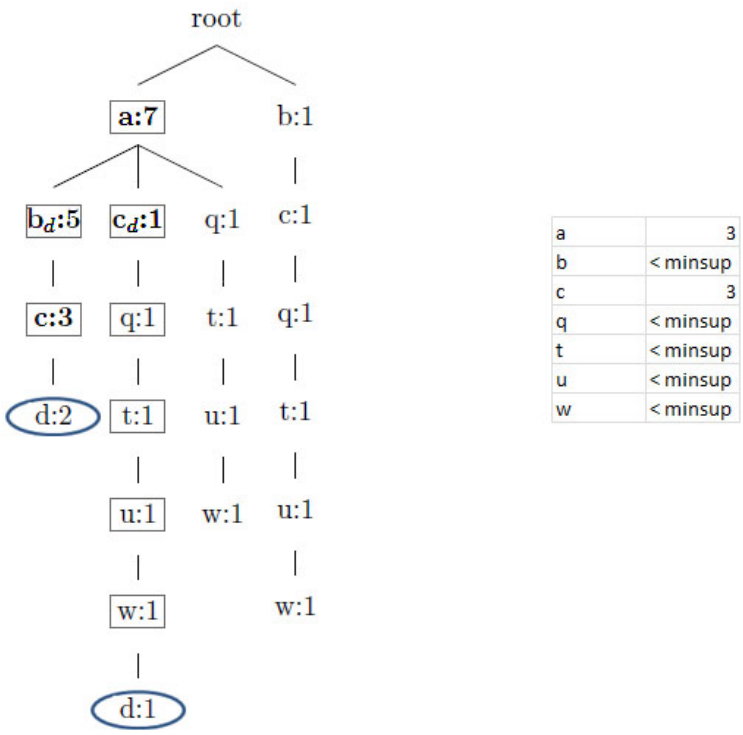

Figure 3.2: The construction of a $\{p\}$-conditional tree by the optimized algorithm

5. for each visited node, check if it is frequent, and if it is, add it to the new conditional tree;

6. stop traversing the current branch if all children of the current node are guaranteed to be infrequent.

Example 3.1 *Let us consider the same datasets as in Example 2.2. Without loss of generality, when building a $\{d\}$-conditional tree, the optimized algorithm first traverses each $\{d\}$-link (circled nodes denote $\{d\}$-link nodes) bottom-up, (1) accumulates encountered items' counts in the header table and (2) for nodes, having multiple children, flags each child node with $\{d\}$. Figure 3.2 demonstrates the process of building a $\{d\}$-conditional tree; nodes, visited during the first traversal appear in squares and nodes, visited during the second traversal, are*

bolded. In both branches during the second traversal, after visiting a node with element c, we can stop the traversal early, as we are guaranteed to not have any frequent items after item c in any *path, given the information collected during the first traversal of the tree.*

Using the tree in Figure 3.2, let us count the number of memory allocations and node visits that my optimization performed and compare them to the baseline FP-growth algorithm. During the first bottom-up traversal, my optimized algorithm visited 9 nodes in the main tree and did not allocate any new nodes. Then, my optimized algorithm traversed the tree once again in the top-down fashion, visiting 4 nodes in the main tree and allocating 2 nodes for the new conditional tree. The baseline FP-growth algorithm, on the other hand, visited 9 nodes (in the main tree) + 8 nodes (in the projected tree) = 17 nodes as well as allocated 9 new nodes for the projected tree, 7 of which were deallocated in the conditional tree. In total, my optimized algorithm traversed 13 nodes and allocated 2 new nodes. It did not perform any memory deallocations.

As observed from the above example, the benefits of employing my optimization include:

1. efficient tree node allocation, which removes the need to allocate memory for infrequent nodes in a projected tree and free it, when pruning a projected tree to a conditional tree;

2. efficient tree traversal, which visits all of the tree nodes only twice in the worst case.

The main limitation of this optimization reveals itself in cases, where the main tree is bushy, having many nodes with a large number of children. When a node having a large number of children is reached during the second tree traversal, the algorithm needs to check *all* the flags of each child node to determine, whether it needs to be visited. Although such datasets are not common, they could deteriorate the behavior of my proposed optimization.

3.4 Summary

In this chapter, I proposed three optimizations. The first optimization involves the application of the ForkJoin framework for harnessing the power of multi-core processors. The application of the ForkJoin framework allows us to further divide the process of mining different conditional trees on many cores of a multi-core processor by launching multiple Java threads, which allow the mining of different FP-trees concurrently. The second optimization involves the use of tree caching for reducing the number of garbage collections. The third optimization is the proposal of an efficient technique for building conditional trees. This technique allows us to omit constructing projected trees and thus eliminates unneeded memory allocations and deallocations. Although the examples in the previous sections covered frequent pattern mining algorithms of precise data, all three optimizations are equally applicable to uncertain tree-based data mining algorithms.

Chapter 4

Uncertain Frequent Pattern Mining Using MapReduce

Recall that the main problem with tree-based frequent pattern mining algorithms for uncertain data is that they result in a much larger tree, then the tree-based precise data mining algorithms. Hence, it takes more time to mine a larger UF-tree.

In Section 1.1, we asked whether it was feasible to deconstruct the steps of the UF-growth algorithm and the UFP-growth algorithm and execute some of these steps in parallel using the MapReduce programming model. By making a number of modifications to the PFP-growth algorithm, I discovered that this is indeed possible. Briefly, to implement the UFP-growth algorithm using MapReduce, this chapter retrofits the PFP-growth algorithm by introducing the mapper and reducer functions, responsible for taking samples and calculating final itemset supports. Conversely, to develop a MapReduce version of the UF-growth algorithm, I change the PFP-growth algorithm so that it executes the UF-growth algorithm instead of the FP-growth algorithm. Additionally, I propose the use of a new combiner

function, which reduces the communication overhead associated with the sending and the receiving of the input database chunks.

I also enhance the baseline FP-growth and UF-growth algorithms, which are executed during the stages of the MapReduce version of the UFP-growth algorithm and the MapReduce version of the UF-growth algorithm respectively, with the tree-based algorithm enhancements, presented in Chapter 3. Lastly, this chapter discusses the benefits along with the limitations of MapReduce in the context of frequent pattern mining of uncertain data.

4.1 UF-growth on MapReduce

Recall from Section 2.4.2 that FP-growth on MapReduce involves (1) partitioning the input database into non-overlapping sub-databases of smaller size (during the **mapping phase of the Parallel Counting stage**), (2) mining sub-databases in parallel (during the **reduction phase of the Parallel Counting stage**), and (3) consolidating the obtained patterns (during the **Aggregating stage**). When handling uncertain data, the main challenge is that we can no longer use the FP-growth algorithm for mining uncertain sub-databases. Consequently, the MapReduce version of the UF-growth algorithm can be described as follows.

1. As each item under the uncertainty data model is associated with an existential probability, instead of calculating the actual support of singleton itemsets (which is done in the FPF-growth algorithm), the **Parallel Counting stage** now calculates the expected support of singleton itemsets. For a singleton itemset $\{x\}$, Equation (2.1) can be simplified as follows:

$$expSup(\{x\}) = \sum_{i=1}^{|DB|} P(x, t_i). \tag{4.1}$$

Recall that $P(x, t_i)$ represents the probability, associated with a given transaction item.

Specifically, at the **mapping phase**, instead of outputting $\langle key' = x,\ value' = 1\rangle$, mapper functions output $\langle key' = x,\ value' = expSup(X)\rangle$.

2. The steps at the **Grouping stage** do not require any changes. This stage splits the list containing frequent singletons into distinct groups and assigns a unique id to each group.

3. The **Parallel FP-growth stage** now comprises one additional combiner phase. As before, at the **mapping phase**, mapper functions identify all group-dependent transactions.

   Then, during the new **combiner phase**, combiner functions receive group-dependent transactions in the form of $\langle key = gid,\ value = \{T_1 \dots T_n\}\rangle$, where T_i represents the i^{th} transaction, and insert them into the UF-tree, creating a compressed tree-based representation of these group-dependent transactions.

   At the **reduction phase**, the algorithm collects all the group-dependent UF-trees (as output by the previous combiner phase) and merges them into one single UF-tree. Then, this phase executes an optimized implementation of the UF-growth algorithm, which (a) mines a number of conditional trees concurrently using the ForkJoin framework, (b) uses tree caching and (c) uses the conditional tree optimization technique, all discussed in Chapter 3.

4. The **Aggregating stage** does not require any changes. This stage continues to collect

the mined patterns and store them to disk.

Let us consider some of the implementation specifics of implementing the discussed algorithm steps. In the UF-growth algorithm used at the Parallel FP-growth stage, to reduce the size of a node, I used byte values (0..99) for storing probabilities, as we are limited to 100 distinct values. Leung et al. [LMB08] reported, that such quantization does not drastically skew the final result. Note, that when multiplying probabilities, we still need to convert byte values to float values and then perform floating point arithmetic.

During the implementation of this algorithm, the problem that I encountered was that, during the Parallel FP-growth stage, the reducer functions, arranged the group-dependent transaction items in an order, different to the one defined in the F-list. The baseline UF-growth algorithm, counted the expected frequency of singleton items (the second time), arranging singleton items in a non-descending order and constructing the main tree using a different order than in the F-list. This behavior resulted in generating just a subset of patterns, as was fixed by removing the redundant singleton counting step.

The newly-created combiner phase of the Parallel FP-growth stage not only reduces the communication cost of sending trees (and not group-dependent transaction items) between computation nodes, but also permits us to preserve the frequency-descending order of the items in the input database. Recall that, before generating group-dependent transactions, the Grouping stage first sorts the original transactions according to the frequency non-descending order of frequent singletons. Hence, the local instances of the UF-growth algorithm do not need to build the group-dependent main trees on their local machines—they receive them as input from previous MapReduce iterations.

4.2 UFP-growth on MapReduce

Recall that the UFP-growth algorithm adds a pre- and a post-processing step to the FP-growth algorithm to take samples from transactions and calculate final itemset supports. Although Calders et al. [CGG10] demonstrated the benefits of using such a sampling technique by applying it to different precise data mining algorithms, they did not apply it to precise data mining algorithms developed to run in the MapReduce environment. As the sampling technique can be used together with any frequent pattern mining algorithm working with precise data, my MapReduce version of the UFP-growth algorithm adds a pre- and a post-processing step to the PFP-growth algorithm. As the PFP-growth algorithm executes the FP-growth algorithm at one of its execution stages, I enhanced the FP-growth algorithm with my proposed optimizations to tree-based frequent pattern mining algorithms, as discussed in Chapter 3. The following describes the changes to the PFP-growth algorithm, which enabled the construction of the Parallel UFP-growth algorithm.

1. At the **Parallel Counting stage**, mapper functions receive $\langle key, value = T_i \rangle$, where T_i represents the i^{th} transaction, as input, and, for each item x in transaction T_i, output $\langle key' = x, value' = expSup(x) \rangle$.

2. The steps at the **Grouping stage** do not require any changes. This stage continues to split the list containing frequent singletons into distinct groups and assign a unique id to each group.

3. The **Parallel FP-growth stage** performs the most important part of the algorithm. Specifically, besides identifying group-dependent transactions, the **mapping phase** also performs sampling. As before, the algorithm loads the G-list into main memory,

and creates a reverse map, in which a singleton is mapped to its corresponding group *id*. Each mapper receives $\langle key, value = T_i \rangle$, where T_i represents the i^{th} transaction, as input. Then, the received transaction is sampled using the Concatenating the Samples method [CGG10]. For every transaction item, the method generates a random real number r ranging from 0 to 1. If r is bigger than the existential probability of the current item, then that item is included in the transaction sample Γ, otherwise it is not. For the current transaction sample, the algorithm (a) substitutes all its items with their corresponding group ids from the reverse map, creating a new list T of the same size, as the transaction size. (b) For each group-id gid in T, the algorithm locates the rightmost appearance L of gid in T, and outputs $\langle key' = gid, value' = \{\Gamma_i[1] \dots \Gamma_i[L]\} \rangle$.

For each received transaction by the mapper functions, this process is repeated s times, where s is the number of samples taken.

Then, at the **combiner phase**, combiner functions receive the group-dependent transactions in the form of $\langle key = gid, value = \{T_1 \dots T_n\} \rangle$ and insert them into the FP-tree, creating a compressed tree-based representation of these transactions.

At the **reduction phase**, the reducer functions collect all the group-dependent FP-trees (as output by the combiner stage) and merge them into one FP-tree. After, the algorithm executes an optimized implementation of the FP-growth algorithm, which (a) mines a number of conditional trees concurrently using the ForkJoin framework, (b) uses tree caching, and (c) uses the conditional tree optimization technique, discussed in Chapter 3. For each mined pattern v, the reducer functions output $\langle key' = null, value' = v + support(v) \rangle$.

4. Finally, to calculate the estimated support of an itemset, the **Aggregating stage**

divides the itemset support, obtained from the output of the previous reduction phase, by n, where n is the number of samples.

At the **mapping phase** mapper functions receive frequent patterns in the form of $\langle key = null, value = v + support(v)\rangle$ and output the real support values of a pattern in the form of $\langle key' = null, value = v + \frac{support(v)}{n}\rangle$ (given that $\frac{support(v)}{n} \geq minsup$). At this stage, the **reduction phase** does not perform any work.

In this algorithm, some of the stages do not need to execute reducer functions. In Apache Hadoop, it is legal to set the number of reduce tasks to zero if no reduction is desired.

4.3 Algorithm Comparison

When contrasting the MapReduce version of the UF-growth algorithm and the MapReduce version of the UFP-growth algorithm we can observe, that the latter requires a post processing step to calculate final itemset supports. Hence, at the *Aggregation stage*, when the minimum support threshold is set to a low value, the MapReduce version of the UFP-growth algorithm can spend a significant amount of time calculating real itemset supports and pruning itemsets, the support of which lies below the minimum support threshold.

On the contrary, while the MapReduce version of the UF-growth algorithm does not perform any calculations at the Aggregation stage, it can spend a significant amount of time executing the Parallel UF-growth stage. As we will see in Chapter 5, the amount of time spent at the Parallel FP-growth stage is a couple of times bigger, then the execution time of all the steps of the MapReduce version of the UFP-growth algorithm.

4.4 Fine-Tuning

This section provides some additional implementation details for both algorithms. It also provides some recommendations in terms of tuning Apache Hadoop.

Both algorithms allow specifying the number of items in a group manually (as a command line parameter), which allows us to vary the number of conditional trees that get mined during one pass of the reducer function during the Parallel FP-growth stage. When a small number of items per group is set, the algorithm is very responsive in that it can output patterns dynamically, as when, during the *Parallel FP-growth stage*, the reducer functions finish their work, the resulting patterns are automatically flushed to the HDFS and are available for viewing.

To collect patterns, the PFP-growth algorithm uses a max-heap indexed by the support value of the patterns found, which stores the top-k most frequent items. In my algorithm implementations, I output the patterns directly to the HDFS, as the use of a max-heap permits storing only the k-max patterns, and does not give a complete picture of all the frequent patterns.

Also, we need to modify the default number of mapper/reducer functions, which can be spawned simultaneously on a given computation node. By default, that number is set to 2; hence, a maximum of 2 mapper and 2 reducer functions can run on one computing node. As we are using ForkJoin, we need to ensure that only one instance of a mapper or a reducer function is executed at any point of time on a given node. To make these changes, two parameters in the `hadoop-default.xml` configuration file, namely `mapred.tasktracker.map.tasks.maximum` and `mapred.tasktracker.reduce.tasks.maximum`, need to be set to 1. Please refer to the Apache Hadoop Official Wiki [Apa12a] for

more information.

One can observe that the effects of using ForkJoin to divide the mining of conditional trees between the processor cores can be equivalent to running two reducer functions, executing independent instances of the FP-growth algorithm on one computing node. Let us elaborate on the cons of using such an approach. First, the implementation of MapReduce versions of the UF-growth and the UFP-growth algorithms use a tree caching technique, which recycles old conditional trees. Two independent reducer functions do not have the benefit of sharing cached trees. Moreover, by using ForkJoin, we get the benefit of *work stealing*, which allows to load balance more effectively. When executing multiple reducers on one node, we do not have this load balancing benefit.

Overall, when looking at Apache Hadoop configuration options, a number of tuning parameters need be set for running a MapReduce job on Hadoop. By changing these parameters, we can modify the number of mappers and reducers, memory allocation settings as well as controls for I/O and network use. When executing the algorithms on a local MapReduce cluster, to get an optimal configuration automatically, a system that adjusts these parameters, called Starfish [HLL+11], would be recommended. Starfish is, however, not available on the baseline Amazon EC2 MapReduce nodes, where I conducted my experiments.

4.5 Discussion

Recall that UF-growth has a lower tree compression ratio then FP-growth, and to merge two nodes in the tree, node items as well as their *probabilities* have to match. Nevertheless, the header table maintains links to items only, connecting nodes having the same item but a different probability. As such, the UF-growth algorithm is required to mine the conditional

tree for each item on the same machine (in the case of MapReduce), or using the same thread (in the case of ForkJoin).

Moreover, to yield the final mining result, every computing node has to perform the mining of the assigned conditional trees. Hence, the mining speed is limited by the slowest node to complete the mining process. As MapReduce uses commodity hardware, we could have scenarios, where slow nodes are performing the most computationally challenging work. Yet, by design, when MapReduce has many nodes available in the network, it assigns the same mining task to different machines, removing this problem altogether. When one machine completes the mining process, MapReduce notifies other nodes working on the same task to cancel their redundant jobs.

Real-life databases, however, contain at least one hundred domain items, making MapReduce an attractive choice. In the case of mining such datasets, the cost of sending conditional trees to the nodes in the network constitutes only a negligible fraction of the overall runtime of both the Parallel UF-growth and the Parallel UFP-growth algorithms. In Section 1.1, we also asked whether it would be possible to effectively mix the use of ForkJoin and MapReduce. As previously discussed, the use of multiple threads, facilitated by the ForkJoin framework, is beneficial in cases where we do not have many nodes in the MapReduce environment; here, each thread is responsible for mining a number of conditional trees sequentially. The ForkJoin framework offloads conditional trees, as well as the responsibility for mining them, to available threads. When we have as many computing nodes as frequent singleton items (i.e., where each node is responsible for mining only one conditional tree), the ForkJoin framework would not be useful, as we cannot divide any computations to multiple threads on one computational node. Such cases, where we have as many computing nodes as frequent singleton items are, nevertheless, extremely rare.

4.6 Summary

This chapter proposed MapReduce versions of the UF-growth and the UFP-growth algorithms. It also overviewed the steps, required to construct such algorithms. The algorithms were developed by employing the previously discussed tree-based optimizations and by using the PFP-growth algorithm as a blueprint. Also, in the process of transmitting sub-databases between computing nodes, a combiner stage was introduced, which compressed database transactions into an FP-tree, to reduce the communication footprint of the algorithm. Lastly, this chapter provided recommendations on how to fine-tune the developed algorithms as well as how to configure some of the key parameters in Apache Hadoop for optimized performance. The next chapter provides experimental results, which compare the execution time of the developed parallel algorithms.

Chapter 5

Experimental Results

In this chapter, I measure the performance benefits of applying the FP-tree traversal optimization, used for building conditional trees. I also evaluate the benefits of executing the FP-growth algorithm in the ForkJoin framework. Lastly, I compare the execution times of the Parallel UF-growth algorithm and the Parallel UFP-growth algorithm.

5.1 Experimental Setup

The first two sets of tests were done on a machine with an Intel Core i7 4-core processor (1.73 GHz), 8 GB of main memory, and the 64-bit Windows 7 operating system. All the algorithms were implemented in the Java programming language. The stock version of Apache Hadoop 0.20.0 was used as an implementation of the MapReduce programming model. The benchmarks used datasets from the UCI Machine Learning Repository [AN12] as well as datasets from the FIMI repository [Goe11]—namely, accidents, connect4 and mushroom, which are commonly used to benchmark frequent pattern algorithms. The number of transactions in each dataset varies from 8000 to 34000 transactions.

For the last batch of experiments, targeted at testing the MapReduce implementations of UF-growth and UFP-growth, I used the Amazon EC2 cluster [Ama12a] with 11 m2.xlarge computing nodes [Ama12b], with each node having 17.1 GB of main memory, 6.5 EC2 Compute Units (2 virtual cores with 3.25 EC2 Compute Units each) and 420 GB of instance storage. Since Amazon EC2 is mainly built on commodity hardware, there may be several different types of physical hardware underlying EC2 instances. As per the official Amazon documentation, one EC2 Compute Unit provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor, which is also the equivalent to an early (2006) 1.7 GHz Xeon processor. Lastly, I did not perform all experiments using the Amazon cloud due to limited financial resources.

For the UFP-growth algorithm, Calders et al. [CGG10] reported that 2 samples per transaction were sufficient to accurately approximate (i.e., with less than 0.02% error) the expected supports of the mined itemsets for most datasets. Following these observations, the Parallel UFP-growth algorithm also takes 2 samples from each transaction.

I completed a variety of tests of the Parallel UF-growth algorithm as well as compared the execution times of the Parallel UF-growth algorithm and the Parallel UFP-growth algorithm. Due to the financial cost associated with running the experiments in the Amazon cloud, I conducted every test only once; also, I tested the UFP-growth algorithm on two datasets only.

As the datasets from the FIMI repository are generally less than 1 million transactions in size, I generated three new synthetic datasets using the IBM Quest Dataset Generator [AS12]. The generated data ranges from 2 million to 5 million transactions with an average transaction length of 10 items, and a domain of 150 items. As these datasets contain precise data, I assigned an existential probability from the range (0,1] to each item in

every transaction according to the normal distribution. Here, I chose to use normal distribution, as datasets with a normal distribution usually take less time to mine than datasets with a uniform distribution. Normal distribution assigns higher probabilities to the values near the central value of an interval; hence, the number of frequent base intervals decreases.

5.2 Efficient Conditional Tree Construction

Experiment 5.1 *Recall from Section 3.3, I proposed an optimization, which enhances the conditional tree construction process. Hence, in the first experiment, I examined the benefits, associated with using this optimization. Specifically, I compared the execution time of the baseline FP-growth algorithm with my optimized version of the FP-growth algorithm, which used the top-down conditional tree construction technique, discussed in Section 3.3.*

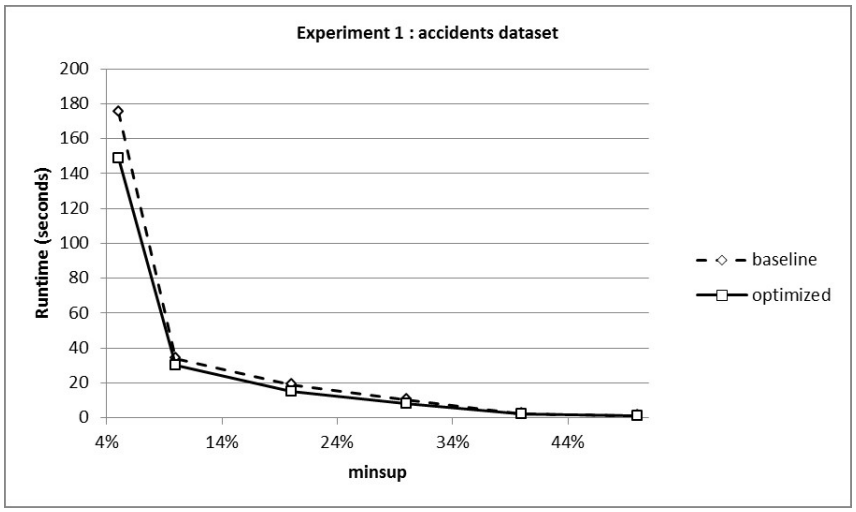

Figure 5.1: Accidents dataset—optimized tree construction (Experiment 5.1)

Figure 5.1 demonstrates, that for small minimum support values both algorithms yield the final result in less than 200 seconds for the accidents dataset. The performance differences

become apparent only when minsup is lowered to 30%. When the algorithm executes for less than 20 seconds, the optimization benefits are less apparent.

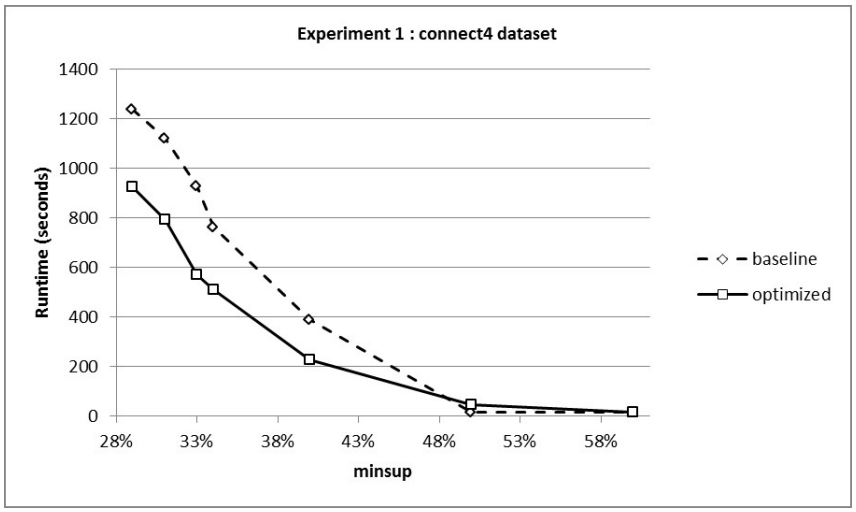

Figure 5.2: Connect4 dataset—optimized tree construction (Experiment 5.1)

As for the connect4 and mushroom datasets, the optimized version of the FP-growth algorithm outperforms the baseline FP-growth algorithm. Figure 5.2 and Figure 5.3 highlight that the optimized FP-growth algorithm performs better, when executed for more than 300 seconds. As noted in previous chapters, the runtime benefits are heavily dependent on data characteristics; as such, we get different running time improvements for mushroom, connect4 and accidents datasets.

Note the difference scale on the x-axis—the minimum support values need to be manually adjusted for each dataset.

5.3 ForkJoin with FP-growth

Experiment 5.2 *Recall from Section 2.1.2 that the original FP-growth algorithm does not exploit multi-core processors. Hence, in Section 3.1, I proposed an optimization, which used*

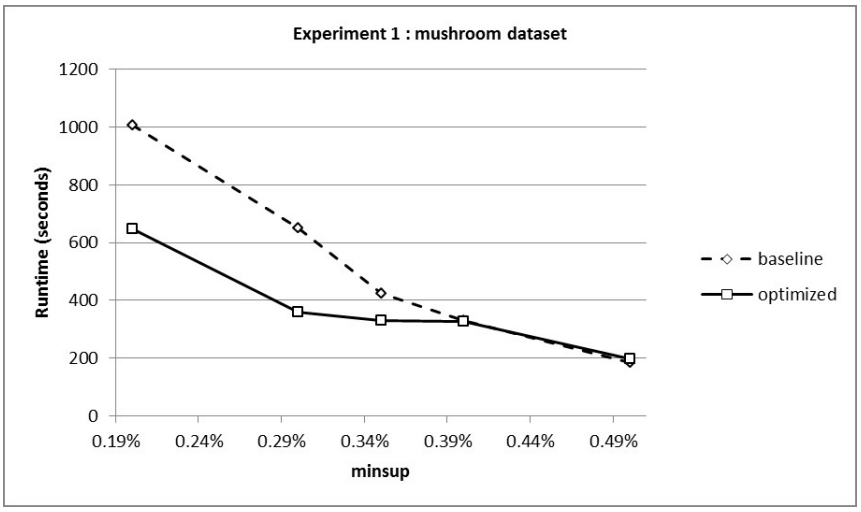

Figure 5.3: Mushroom dataset—optimized tree construction (Experiment 5.1)

the ForkJoin framework to leverage the multi-core facilities of processors. As a result, this experiment demonstrates the effect of employing multiple threads for mining frequent patterns using the FP-growth algorithm. As the tests were executed on a 4-core machine, I varied the number of threads from 1 to 4. To get some intuition about the effectiveness of mining multiple trees concurrently, the executed tests reflected the performance of the algorithm for datasets with both, short and long execution times.

One of the most commonly used performance metrics for parallel processing is **speed-up**, *which reflects the performance gain of parallel execution versus sequential execution. Hence, I calculated the speed-up obtained from running the FP-growth algorithm in the ForkJoin framework.*

For the accidents dataset, when the execution time is small, the algorithm executes slower on multiple threads than on one thread. Here, the algorithm cannot take advantage of multiple cores, as there is not enough work to do in parallel to amortize the overhead of thread creation, dispatch, join and teardown. The overhead of thread creation, however, is not magnified

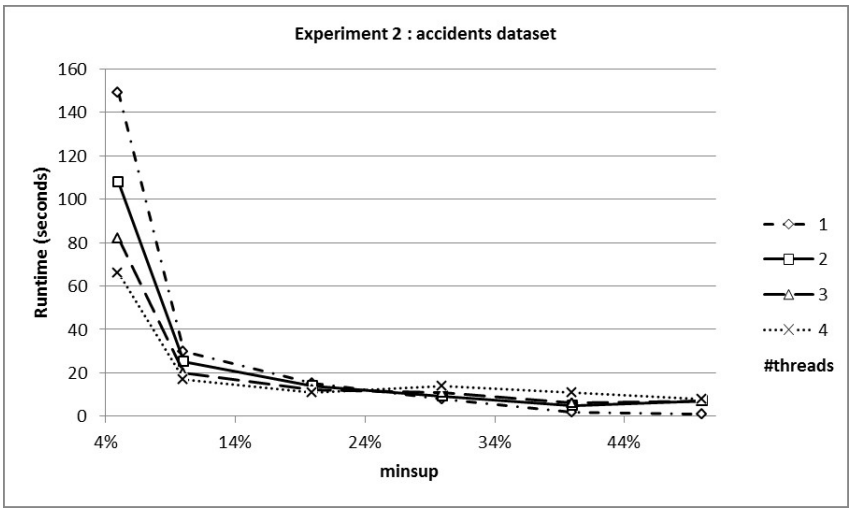

Figure 5.4: Accidents dataset—execution times on multiple threads (Experiment 5.2)

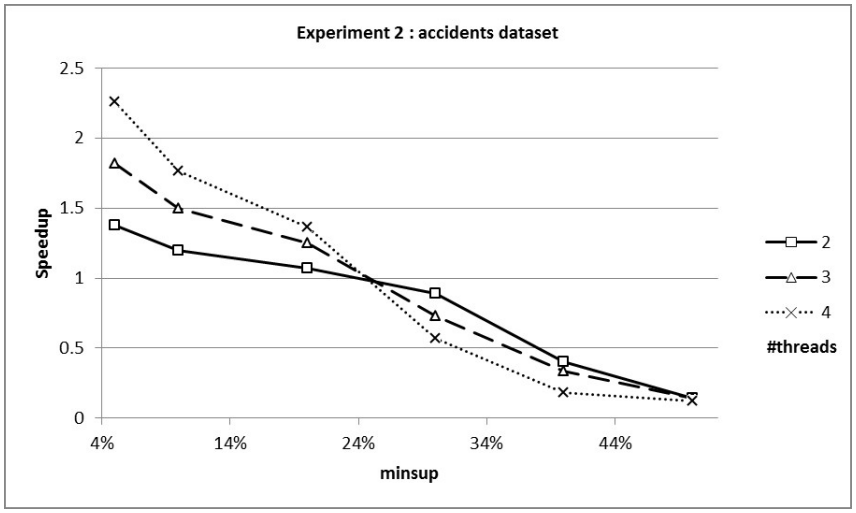

Figure 5.5: Accidents dataset—speedup obtained (Experiment 5.2)

because threads have to be started and stopped only once. In Figure 5.4, the run times for all threads level off when the algorithm executes for 10 seconds. With a minimum support set to less than 10%, the additional costs of starting and stopping threads are largely offset by the overall execution time reductions. By reducing the value of minsup, we generate more patterns, which increases the running time. Here, Gustafson's Law [Gus88] comes into

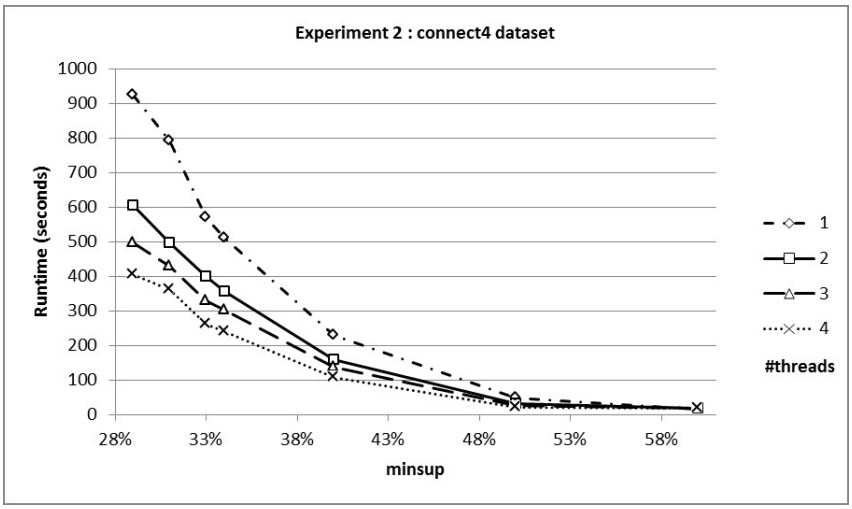

Figure 5.6: Connect4 dataset—execution times on multiple threads (Experiment 5.2)

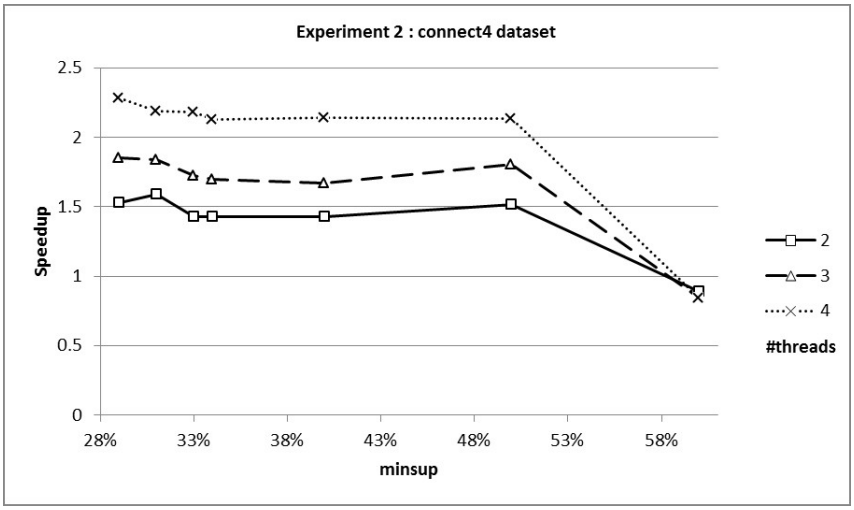

Figure 5.7: Connect4 dataset—speedup obtained (Experiment 5.2)

action, providing consistent speedup, as shown in Figure 5.5.

For the connect4 and mushroom datasets, Figure 5.6 and Figure 5.8 demonstrate, that we can see that for small runtimes the performance benefits are not evident. Conversely, when the algorithm executes for a longer period of time (e.g., more than 100 seconds), sub-linear speedup is achieved. As can be seen, when the algorithm has a sufficient amount of work to

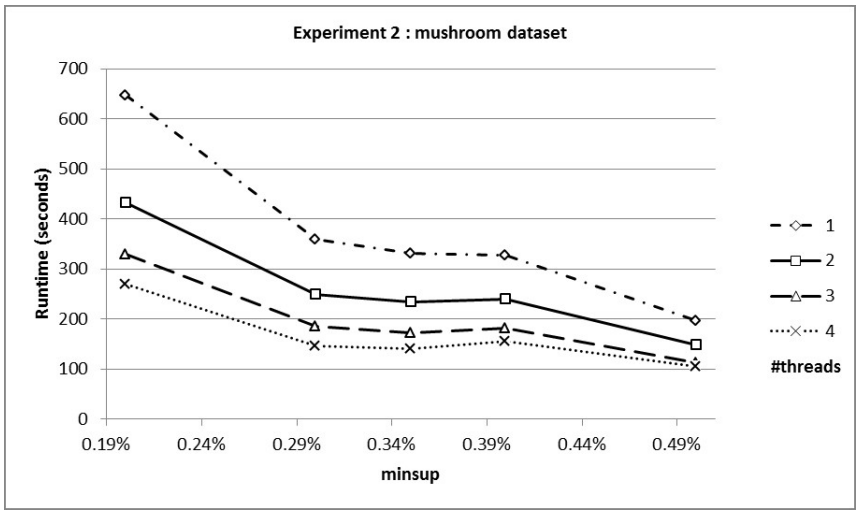

Figure 5.8: Mushroom dataset—execution times on multiple threads (Experiment 5.2)

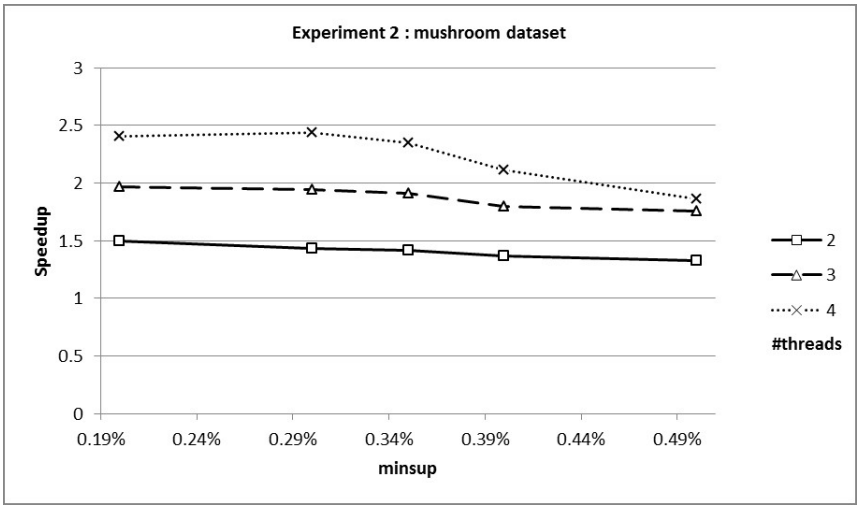

Figure 5.9: Mushroom dataset—speedup obtained (Experiment 5.2)

perform, it efficiently offloads this work to multiple threads, reducing the overall execution time.

The above results illustrate the effectiveness of the tree-based algorithm optimization described in Chapter 3. Specifically, my tests reflected some performance improvements for the accidents dataset, whereas for the connect4 and mushroom datasets, we saw drastic running

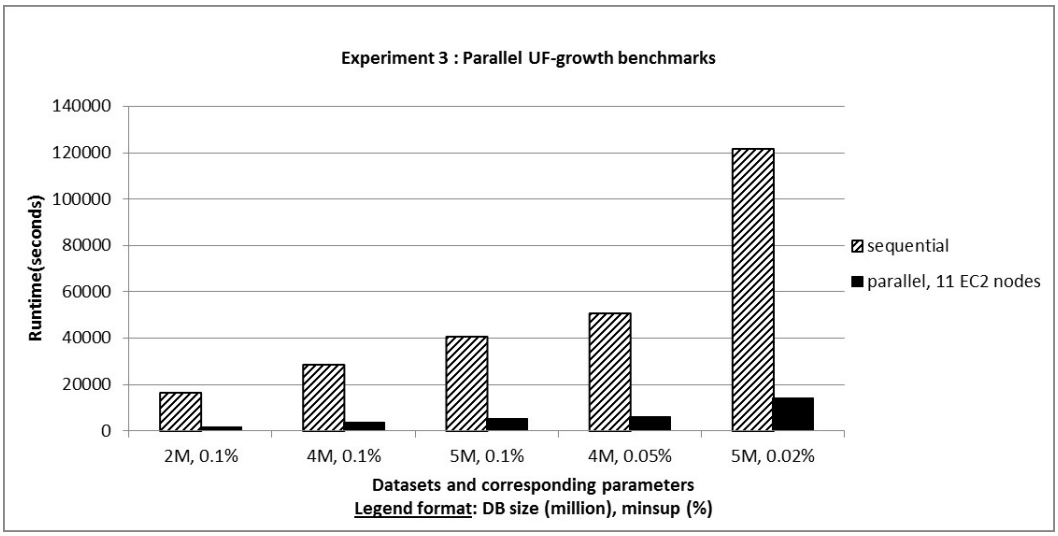

Figure 5.10: Parallel UF-growth benchmarks (Experiment 5.3)

time improvements.

5.4 Parallel UF-growth vs. Parallel UFP-growth

To verify the correctness of my parallel algorithm implementations, I compared the obtained mining results with the results produced by the U-Apriori algorithm [CKH07]. The Parallel UF-growth algorithm consistently arrived at the same result as the U-Apriori algorithm, while the Parallel UFP-growth algorithm always yielded different but high-quality approximate results.

Experiment 5.3 *In this experiment, I executed the UF-growth algorithm in the MapReduce environment with 11 nodes. Figure 5.10 suggests that even when the sequential version of the UF-growth algorithm executed for more than 120,000 seconds, it never reached the 20,000 second mark in the MapReduce environment.*

According to Figure 5.10 and Figure 5.11, when the total execution time of the Parallel

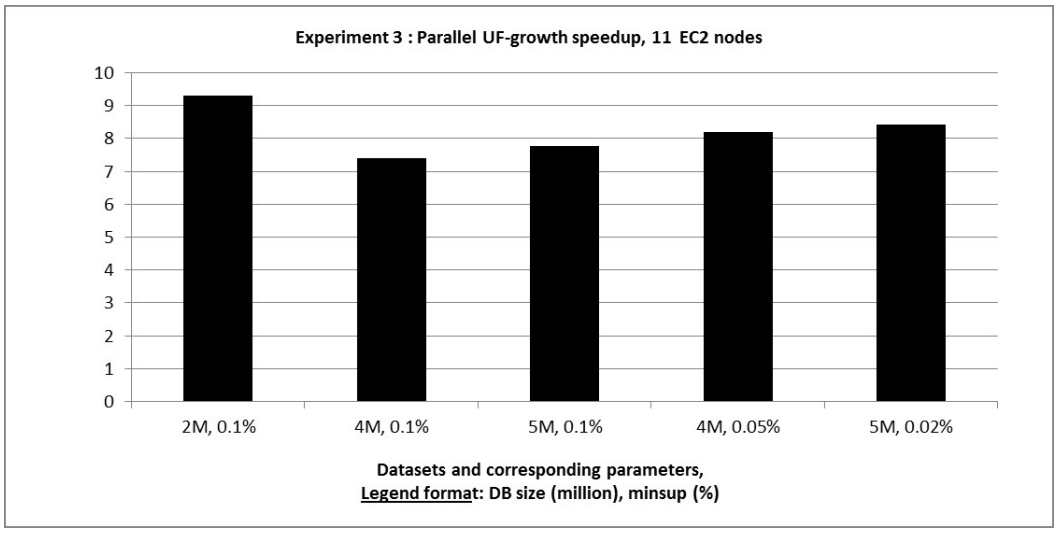

Figure 5.11: Parallel UF-growth speedup; as executed on 11 nodes (Experiment 5.3)

UF-growth algorithm is low, the achieved speedup of 7 times to 8 times over its sequential version was observed. As we increased the problem size, the algorithm achieved a speedup of around 8.5 times on 11 nodes. Interestingly, for 2 million transactions, the speedup rose to as high as 9 times.

The first source of overhead in the Parallel UF-growth algorithm was due to communication cost. In particular, the algorithm had to perform the partitioning of the input database before it could proceed with the mining process. For our algorithm to be beneficial, the data distribution costs need to be more than offset by the gain of parallelization.

To test the capacity of the Parallel UF-growth algorithm to further offload work to different processor cores by using the ForkJoin framework, I compared the execution times of the algorithm on one thread versus two threads. From Figure 5.12 it can be observed that while the execution time of the algorithm on two threads is lower, the overall benefits of distributing work to multiple threads is not particularly significant in general. This behavior is expected, as Amazon uses virtualization to expose virtual processing cores *on shared commodity hardware*

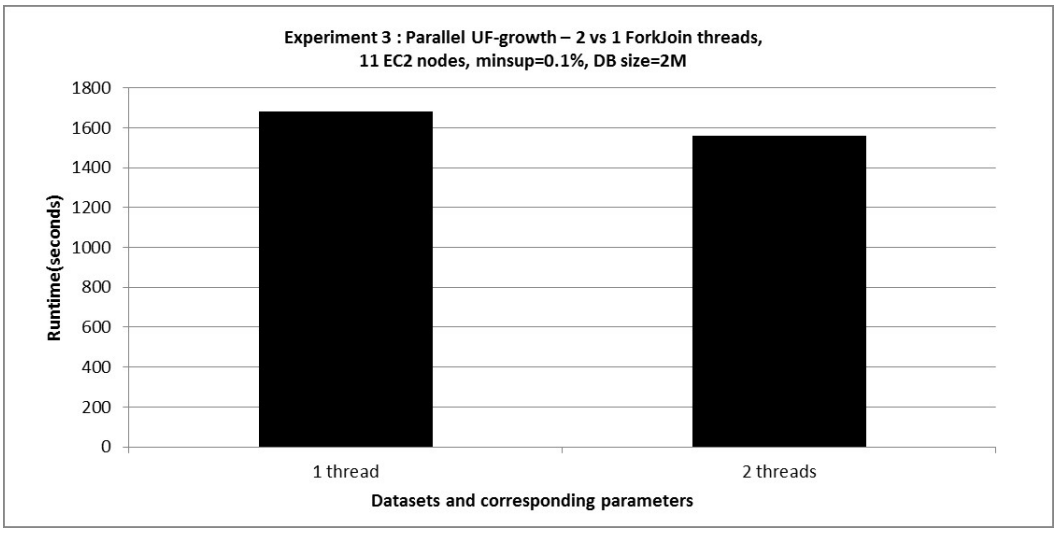

Figure 5.12: Parallel UF-growth with ForkJoin (Experiment 5.3)

resources, degrading potential speedup; significant performance improvements can be observed in MapReduce environments built from high-performance machines [LPD+11].

Experiment 5.4 *The last experiment compared the execution times of the Parallel UF-growth algorithm with the Parallel UFP-growth algorithm. Figure 5.13 shows that the Parallel UFP-growth algorithm consistently yielded the final result faster, when compared to the Parallel UF-growth algorithm.*

5.5 Summary

This chapter tested the proposed optimizations to tree-based frequent pattern mining algorithms as well as benchmarked the Parallel UF-growth algorithm and the Parallel UFP-growth algorithm in the MapReduce environment. Judging by the obtained results, we see that, by distributing the construction and mining of conditional trees to different processor cores, we effectively increased the memory use, associated with the mining, because we have

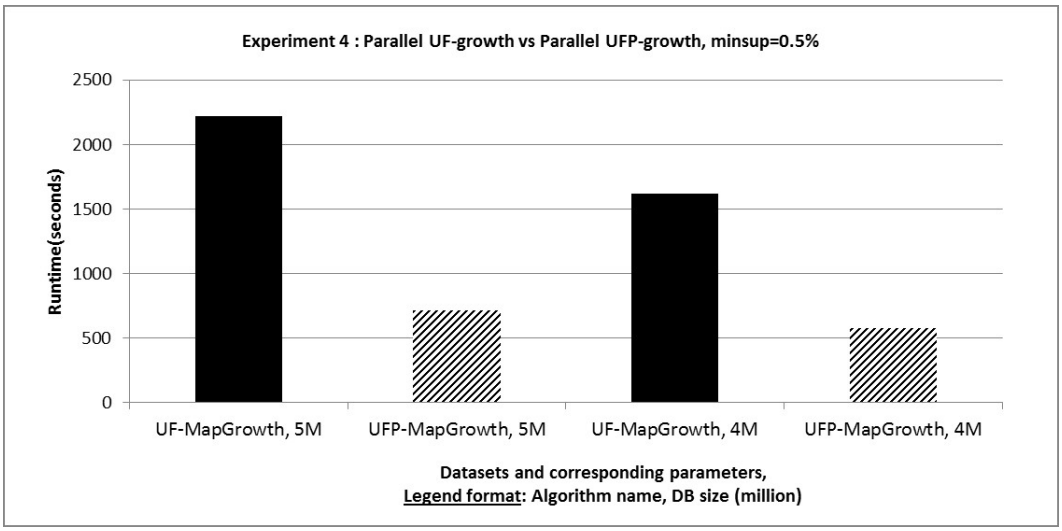

Figure 5.13: Parallel UF-growth vs. Parallel UFP-growth (Experiment 5.4)

to keep many more trees in memory when executing the algorithm in many threads. Hence, the choice to distribute the mining of conditional trees to multiple threads by means of using the ForkJoin framework can be justified when there is a limited number of high-performance machines in the MapReduce network. Unfortunately excessive memory requirements of the ForkJoin framework renders it inapplicable for MapReduce networks built using commodity hardware. Often, these machines do not have multi-core processors and/or their main memory does not have the capacity to hold all conditional trees. In such cases, users should opt to employ the MapReduce framework alone. This can be done by setting the thread count to 1, which effectively turns off the ForkJoin mechanisms. In essence, depending on the network specifics, each of the above techniques can play an important role in speeding up tree-based frequent pattern mining algorithms.

The MapReduce versions of the UF-growth and UFP-growth algorithms are also tailored towards specific use cases. When an analyst needs to rapidly mine a given dataset, he ought to consider using the UFP-growth algorithm, as it yields a high quality approximate result

faster than the UF-growth algorithm. Then, to get the exact mining result, if necessary, the dataset can be processed by means of using the UF-growth algorithm again, which has a higher execution time.

Here are some additional aspects to consider when evaluating the results produced by using the Parallel UF-growth and the Parallel UFP-growth algorithms:

- In cases when the input datasets are constructed based on the data obtained from satellite imaging or doctor diagnosis reports, the probabilities associated with items in the input dataset are also approximated. For example, even highly skilled doctors cannot indicate the *exact* probability of a patient having a certain disease.

- As the UF-growth algorithm uses numerous floating point calculations, the final minimum support values are not exact, due to the inexact nature of floating point computations. The sampling approach in the parallel UFP-growth algorithm does not have this problem.

Given these observations, there might not be a big difference in terms of result accuracy when comparing the Parallel UFP-growth and the Parallel UFP-growth algorithm.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

There are many real-life situations, where we observe uncertain data, such as in temperature and wind speed readings, patient diagnosis, and satellite imaging, among many others. Given the probabilistic nature of this data, it takes more time and resources to mine it. Currently, the state-of-the-art frequent pattern mining of uncertain data algorithms do not provide sufficient performance, as most of them are not crafted to execute in parallel. This M.Sc. thesis research described techniques that reduced the execution time of two uncertain data mining algorithms—UF-growth and UFP-growth, by executing them using the MapReduce programming model and the ForkJoin framework.

Chapter 3 of my thesis discussed my proposed improvements to tree-based algorithms. The **first improvement** reduced the time to construct conditional trees by removing the construction of intermediate projected trees. This improvement reduced both the number of visited nodes and the number of redundant memory allocations and deallocations. The

second improvement employed the ForkJoin framework for concurrently mining conditional trees on multiple cores of a multi-core processor and the **third improvement** used tree caching to reduce memory consumption. Chapter 4 overviewed the steps for executing the UF-growth and UFP-growth algorithms in parallel using MapReduce. Both of the algorithms use the proposed improvements, discussed in Chapter 3. Finally, Chapter 5 presented experimental results, confirming the effectiveness of the proposed optimizations to tree-based algorithms. The MapReduce programming model leveraged numerous performance benefits for algorithms which process large amounts of data, and the ForkJoin programming model further distributed computations between the processing cores of a multi-core processor.

In Section 1.1, I asked the questions: "Can we decompose our frequent pattern mining of uncertain data problem to multiple smaller problems, which can be attacked by multiple machines, such that each smaller-sized problem can fit in the main memory on one computing machine? Moreover, will such means of distribution speed-up the execution of uncertain data mining algorithms? As to the parallel techniques, will we be able to employ the MapReduce programming model together with the ForkJoin framework as tools for parallelizing UF-growth and UFP-growth? If so, will the use of MapReduce and ForkJoin be beneficial for executing UFP-growth and UF-growth in parallel?"

In response to these questions, Chapter 4 introduced algorithms, which provided the possibility to construct and mine smaller-sized FP-trees and UF-trees on distributed machines as well as provided experimental results demonstrating the effectiveness of employing the MapReduce programming model for mining frequent patterns from uncertain databases. As for the use of the ForkJoin framework, Section 4.5 discussed the use cases, where the use of ForkJoin together with MapReduce yielded the most benefits. In my experiments, I used computing nodes, provided by the Amazon cloud for benchmarking my algorithms.

The nodes provided by Amazon had virtual (and not real) processing cores, hence the use of the ForkJoin framework did not significantly improve the running time of my proposed algorithms, presumably because the multiple threads were run on virtual machines mapped to the same physical machine. Nevertheless, the use of MapReduce alone yields significant speedups for the proposed frequent pattern mining of uncertain data algorithms.

6.2 Future Work

Currently, the primary limitation of the proposed MapReduce implementations of uncertain data mining algorithms is that they do not consider the associated amount of work, required for mining a given conditional tree. The algorithms uniformly divided the responsibility for mining first-level conditional trees between available nodes in the MapReduce network. They did not, however, consider cases, where a small number of level 1 conditional trees took the bulk of execution time. In such cases, most of the nodes would stall, waiting for the few nodes to finish the most computationally challenging work. In such cases of load imbalance, we would not reduce the execution time of the algorithm by employing more computing power. Although Zhou et al. [ZZC+10] proposed a load balancing technique for PFP-growth, it addressed and was beneficial for only a limited subset of datasets restricting its wide application in practice. Briefly, the proposed technique uses a basic heuristic, instrumented to predict the amount of work associated with mining a given conditional tree.

To the best of my knowledge, the baseline version of MapReduce does not currently provide the capacity to schedule new MapReduce sub-phases on the fly; MapReduce algorithms are required to follow a predefined execution path, specified in the main controller. The primary issue with the stock version of MapReduce is that it cannot perform load balancing in

a straightforward way. As such, future work could be targeted at finding such a framework, possibly an extension of MapReduce, which would allow us to recursively build sub-trees and schedule their mining on available computation resources.

Furthermore, no research work has been done in the area of executing stream mining algorithms using MapReduce. Condie et al. [CCA+10] developed a system called MapReduce Online, which was capable of working with data streams. This system allowed the processing of batches of transactions on the fly. Such a system can be used to analyze uncertain data, which is arriving over time. Leung and Jiang [LJ11] proposed a number of methods for mining data streams of uncertain data using tree-based algorithms and focused mainly on the sliding window model, the time-fading model and the landmark model as means of measuring and maintaining the frequency of itemsets overtime. Nevertheless, Leung and Jiang have not yet considered sampling the input data streams and applying algorithms, which mine data streams from precise data. In this scenario, MapReduce Online could be used as a module for fetching and sampling the incoming transaction batches. In this scenario, the computing nodes in the MapReduce network would have to be working in an online mode waiting for the arrival of new data batches, which could be costly when executed using the Amazon EC2 computing nodes. Nevertheless, the use of MapReduce would enable us to process data at a much higher rate.

Bibliography

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, DC, USA, May 1993. ACM. 10.1145/170035.170072.

[ALWW10] Charu C. Aggarwal, Yan Li, Jianyong Wang, and Jing Wang. Frequent pattern mining with uncertain data. In *KDD '09: 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 29–38, Paris, France, June 2010. ACM. 10.1145/1557019.1557030.

[Ama12a] Amazon. Amazon elastic compute cloud. `http://aws.amazon.com/ec2`, Accessed on March 1, 2012.

[Ama12b] Amazon. Amazon elastic compute cloud—instance types. `http://aws.amazon.com/ec2/instance-types/`, Accessed on March 1, 2012.

[AN12] Arthur Asuncion and David J. Newman. UCI machine learning repository. `http://mlearn.ics.uci.edu/MLRepository.html`, Accessed on March 1, 2012.

[Apa12a] The Apache Software Foundation. Apache Hadoop Wiki. `http://wiki.apache.org/hadoop`, Accessed on January 1, 2012.

[Apa12b] The Apache Software Foundation. Apache Hadoop home page. `http://hadoop.apache.org`, Accessed on March 1, 2012.

[Apa12c] The Apache Software Foundation. Apache Mahout home page. `http://mahout.apache.org`, Accessed on March 1, 2012.

[Apa12d] The Apache Software Foundation. Apache Pig home page. `http://hadoop.apache.org/pig/`, Accessed on March 1, 2012.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *VLDB '94: 20th International Conference on Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994. Morgan Kaufmann Publishers. 645920.672836.

[AS12] Rakesh Agrawal and Ramakrishnan Srikant. Quest synthetic data generator. IBM Almaden Research Center, Accessed on March 1, 2012.

[CCA+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *NSDI '10: 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 313–328, San Jose, CA, USA, April 2010. The MIT Press. UCB/EECS-2009-136.

[CGG10] Toon Calders, Calin Garboni, and Bart Goethals. Efficient pattern mining from uncertain data with sampling. In *PAKDD '10: 14th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 480–487, Hyderabad, India, June 2010. Springer. 10.1007/978-3-642-13657-351.

[CJT+11] Robson L. F. Corderio, Caetano Traina Jr., Agma J. M. Traina, Julio Lopez, U Kang, and Christos. Faloutsos. Clustering Very Large Multi-dimensional Datasets with MapReduce. In *KDD '11: 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 690–698, San Diego, CA, USA, August 2011. ACM. 10.1145/2020408.2020516.

[CKH07] Chun-Kit Chui, Ben Kao, and Edward Hung. Mining frequent itemsets from uncertain data. In *PAKDD '07: 11th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, pages 47–58, Nanjing, China, May 2007. Springer. 10.1007/978-3-540-71701-0_8.

[CKL+06] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Ng, and Kunle Olukotun. Map-Reduce for machine learning on multicore. In *NIPS 2006: Advances in Neural Information Processing Systems 19*, pages 281–288, Vancouver, BC, Canada, December 2006. The MIT Press. http://www.willowgarage.com/map-reduce-machine-learning-multicore.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 10.1145/1454008.1454027.

[DYM+10] Xiangyuan Dai, Man Lung Yiu, Nikos Mamoulis, Yufei Tao, and Michail Vaitis. Probabilistic spatial queries on existentially uncertain data. In *SSTD 2005: 9th International Symposium on Spatial and Temporal Databases*, pages 400–417, Angra dos Reis, Brazil, August 2010. Springer. 10.1.1.105.1925.

[Goe11] Bart Goethals. Frequent itemset mining implementations repository. `http://fimi.ua.ac.be`, Accessed on August 26, 2011.

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31:532–533, May 1988. 10.1145/42411.42415.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1):10–18, 2009. 10.1145/1656274.1656278.

[HLL+11] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, F B Cetin, and S Babu. Starfish: A self-tuning system for big data analytics. In *CIDR 2011: 5th Biennial Conference on Innovative Data Systems Research*, pages 261–272, Asilomar, CA, USA, January 2011. CIDR. http://www.cs.duke.edu/~shivnath/papers/starfish_cidr11.pdf.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00: 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Dallas, TX, USA, May 2000. ACM. 10.1145/342009.335372.

[JK04] Asif Javed and Ashfaq Khokhar. Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, 16(3):321–334, 2004. 10.1023/B:DAPD.0000031634.19130.bd.

[Kea93] Michael Kearns. Efficient noise-tolerant learning from statistical queries. In *STOC'93: 25th Annual ACM Symposium on Theory of Computing*, pages 392–401, San Diego, CA, USA, May 1993. ACM. 10.1145/167088.167200.

[KKR10] Sathish T. Kumar, V. Kavitha, and T. Ravichandran. Efficient tree-based distributed data mining algorithms for mining frequent patterns. *International Journal of Computer Applications*, 10(1):11–16, 2010. 10.5120/1447-1957.

[KPP03] Walter A. Kosters, Wim Pijls, and Viara Popova. Complexity analysis of depth first and FP-growth implementations of APRIORI. In *MLDM '03: 3rd International Conference on Machine Learning and Data Mining in Pattern Recognition*, pages 284–292, Leipzig, Germany, July 2003. Springer-Verlag. 10.1007/3-540-45065-3_25.

[Lea00] Doug Lea. A java fork/join framework. In *Java '00: 2000 ACM conference on Java Grande*, pages 36–43, San Francisco, CA, USA, June 2000. ACM. 10.1145/337449.337465.

[Leu11] Carson Kai-Sang Leung. Mining uncertain data. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(4):316–329, 2011. 10.1002/widm.31.

[LJ11] Carson Kai-Sang Leung and Fan Jiang. Frequent pattern mining from time-fading streams of uncertain data. In *DaWaK'11: 13th International Conference on Data Warehousing and Knowledge Discovery*, pages 252–264, Toulouse, France, August 2011. Springer-Verlag. 10.1007/978-3-642-23544-3_19.

[LL10] Kawuu W. Lin and Yu-Chin Luo. Efficient strategies for many-task frequent pattern mining in cloud computing environments. In *SMC 2010: 2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 620–623, Istanbul, Turkey, October 2010. IEEE. 10.1109/ICSMC.2010.5641816.

[LLZT07] Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of frequent itemset mining on multiple-core processor. In *VLDB '07: 33rd International Conference on Very Large Data Bases*, pages 1275–1285, Vienna, Austria, September 2007. VLDB Endowment. http://portal.acm.org/citation.cfm?id=1325997.

[LMB08] Carson Kai-Sang Leung, Mark Anthony F. Mateo, and Dale A. Brajczuk. A tree-based approach for frequent pattern mining from uncertain data. In *PAKDD '08: 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 653–661, Osaka, Japan, May 2008. Springer. 10.1007/978-3-540-68125-0_61.

[LPD+11] Wes Lloyd, Shrideep Pallickara, Olaf David, Jim Lyon, Mazdak Arabi, and Ken Rojas. Migration of multi-tier applications to Infrastructure-as-a-Service clouds: An investigation using Kernel-based virtual machines. In *GRID 2011: 12th IEEE/ACM International Conference on Grid Computing*, pages 137–144, Lyon, France, September 2011. IEEE. 10.1109/Grid.2011.26.

[LS11] Carson Kai-Sang Leung and Lijing Sun. Equivalence class transformation based mining of frequent itemsets from uncertain data. In *SAC 2011: 26th Annual ACM Symposium on Applied Computing*, pages 983–984, TaiChung, Taiwan, March 2011. ACM. 10.1145/1982185.1982399.

[LWZ+08] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Chang. PFP: Parallel FP-growth for query recommendation. In *RecSys'08: 2008 ACM Conference on Recommender Systems*, pages 107–114, Lausanne, Switzerland, October 2008. ACM. 10.1145/1454008.1454027.

[MPI12] MPI Forum. MPI: A Message-Passing Interface Standard Version 2.2. `http://www.mpi-forum.org`, Accessed on March 1, 2012.

[Ope12] The OpenMP Architecture Review Board. OpenMP: C and C++ Application Program Interface Version 3.1. `http://openmp.org`, Accessed on March 1, 2012.

[PHL+01] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-Mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM '01: 1st IEEE International Conference on Data Mining*, pages 441–448, San Jose, CA, USA, November 2001. IEEE. 10.1109/ICDM.2001.989550.

[TGA08] Akhilesh Tiwari, Rajendra K. Gupta, and Dharma P. Agrawal. A survey on frequent pattern mining: Current status and challenging issues. *Information Technology Journal*, 9(7):1278–1293, 2008. 10.3923/itj.2010.1278.1293.

[TS11] Raja Tlili and Yahya Slimani. Executing association rule mining algorithms under a grid computing environment. In *PADTAD '11: The Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 53–61, Toronto, ON, Canada, July 2011. ACM. 10.1145/2002962.2002973.

[vG12] Martijn van Groningen. Introduction to Hadoop. `http://www.searchworkings.org/blog/-/blogs/introduction-to-hadoop/`, Accessed on March 1, 2012.

[Zak00] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000. 10.1109/69.846291.

[Zhe02] Gengbin Zheng. Parallelizing FP-growth Frequent Patterns Mining Algorithm using OpenMP. 2002. Urbana, IL, USA.

[ZW10] XiLu Zhu and Bai Wang. Community mining in complex network based on parallel genetic algorithm. In *ICGEC 2010: Fourth International Conference on Genetic and Evolutionary Computing*, pages 325–328, Xiamen, China, December 2010. IEEE. 10.1109/ICGEC.2010.87.

[ZZC+10] Le Zhou, Zhiyong Zhong, Jin Chang, Junjie Li, J. Z. Huang, and Shengzhong Feng. Balanced parallel FP-growth with MapReduce. In *YC-ICT 2010: 2010 IEEE Youth Conference on Information Computing and Telecommunications*, pages 243–246, Beijing, China, November 2010. IEEE. 10.1109/YCICT.2010.5713090.