# Scalable Vertical Mining for Big Data Analytics

by

**Hao Zhang**

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

November 2016

Thesis advisor | Author

**Dr. Carson K. Leung** | **Hao Zhang**

**Scalable Vertical Mining for Big Data Analytics**

# Abstract

The increasing size of modern applications produces huge amounts of data, which in turn leads to a new challenge to data mining or big data analytics. Researchers often use the five V's (Volume, Velocity, Variety, Veracity, and Value) to describe the features of big data. The interest of discovering patterns from a large collection of data has risen in both academic and industrial areas. Examples of rich sources of big data are on-line social networks like Facebook or Twitter. Embedded in these user online social activities are useful information and knowledge. Recently, although some algorithms have been proposed to mine a large scale of data, they mostly focused on the volume aspect. Unfortunately, not that many approaches have been focused on data variety which is also a critical criterion for mining process. The composition of a dataset could either be sparse or dense, or not evenly uniformly distributed. For example, a list of common friends in an on-line social network can be dense if two people share a lot of common friends; it could be sparse otherwise. For my MSc thesis, I design and implement a big data analytic algorithm that tackles both volume and variety aspects of big data.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgements

First, I express my great respects to Dr. Carson K. Leung as my academic supervisor during my studies here. As an international student here, sometimes I ran into some troubles that I had no idea how to deal with it. But, he gave me a lot of usefully advices to help me to get through and survive here. Whenever I have a question, he has always been there to help no matter it was in his office hour at the university or late at night when he was in somewhere else, he always replied my email in a very short time. Without his guidance, I could not finish my work here.

I thank Dr. Melanie Soderstrom (Psychology) for her constructive comments on the early stage of this research work. I also thank Dr. Yang Wang (Computer Science) and Dr. Carl Ngai Man Ho (Electrical and Computer Engineering) for examining this MSc thesis, as well as Dr. Noman Mohammed for chairing my MSc defence.

Moreover, I thank my lab mates here in Database and Data Mining Lab, including those who are going to graduate and also who are already graduated. It is my honor to meet you guys. Thanks for your help and accompany, I really had a great time here.

The last but not least, thanks to my parents for both financial and moral support. It is a life changing idea to send me to study in Canada. I have not only learned a lot of useful knowledge, but also met so many interesting people and things. I really want to share these with you.

Hao Zhang
B.Sc., Northwest A&F University, China, 2009

*The University of Manitoba*
*November 2016*

*This thesis is dedicated to my parents.*

# Chapter 1

# Introduction

It is common to see that many things are connected in life. For example, a customer who buys new cell phone would also likely to buy a phone case and a screen protector at the same time, but it is less likely that the customer would want to buy a microwave at the same time. So, a question arises: How to reveal the things that are related and how strong (or weak) are the internal relations?

The 21st century is known as Information Age. Enormous quantities of data are generated and collected daily to about every aspect of society, such as engineering, scientific researches and social media. On the one hand, the fast growth of information shows the potential to learn from new knowledge. On the other hand, it creates some obstacles for people to understand it in a short time. The knowledge or useful information is often hidden inside and there is also a lot of noise in data which prohibits us from understanding the useful information without some powerful tools.

## 1.1 Data Mining

**Data mining**, also known as knowledge discovery from data (KDD) [PF91], is a useful tool to discover hidden knowledge inside data. Companies use the mined knowledge to design item categories, decide promotional bundles, etc. For this reason, some data processing tools such as data mining and machine learning were invented to ease the gap between raw data and the real knowledge that we need. People use the term "mining" to vividly describe the process of finding gold (knowledge) from a wild space (data). We prefer to say that **data mining** is a procedure for finding non-trivial, previously unknown and potentially useful information in data. These data can be in a variety of formats such as text, graphs, audio, or even video. Usually, we know how many times actual data had appeared in real-life scenarios such as sales records. But, sometimes we do not know the actual data such as weather forecast or a series of data transmitted from wireless sensors. Because of inaccuracy of hardware, these data may not accurate. We call this type of data **uncertain data.** [LMB08]

Generally, there are two stages in data mining. The first stage is extracting *information* from data. This *information* could be in the form of patterns, associations, or relationships that are inside the data [HPK11]:

1. **Classes:** Determining the category of new item among a groups' predefined labels. For example, airline companies can predict whether a customer wants to buy an economy or business seat. The corresponding technology is classification.

2. **Clusters:** Grouping a set of items based on the attributes of the items. An example of clusters would be when biologists want to find out which parts

of a DNA sequence have similar functions. The corresponding technology is clustering.

3. **Sequential patterns:** Finding related information in a sequence, usually in a time series. For example, in an environment surveillance, people want to see the differences between the current status and previous status.

4. **Association Rules:** Finding the co-occurring itemsets from a transaction dataset. For example, a rule $\{milk, bacon\} \rightarrow \{bread\}$ found in supermarket records will indicate that customers who bought milk and bacon would also be likely to buy bread too. The corresponding technology is frequent pattern mining. I will focus on this is type of relationships in this thesis.

The second stage in data mining is converting useful information into *knowledge*. This *knowledge* is what I am interested and I use it later in context such as predicting future trends.

## 1.2 Frequent Pattern Mining

**Frequent itemsets mining (FIM)** [AS94] is one of the major topics in data mining. The procedure of FIM is finding the *interesting* itemsets from a transaction dataset. Here, *interesting* means that the pattern appears frequently in the dataset. One of the major challenges for FIM is that sometimes the size of outputs is large than the size of inputs. For example, with three domain items $a, b$ and $c$ in a database, FIM can generate $\{a\}$, $\{a, b\}$, $\{a, b, c\}$, $\{a, c\}$, $\{b\}$, $\{b, c\}$, $\{c\}$, which are up to $2^3 - 1 = 7$ interesting patterns as output. So, one important criterion for FIM algorithms is the ability to efficiently check if a subset is interesting or not. More details will be discussed in Chapter 2.

In practice, the outputs from FIM are usually used as important intermediate results for other mining technologies like clustering and classification. Frequent itemsets mining can be used in a variety of domains and some of unique problems may require a correspondingly modified version of FIM correspondingly. Below are some areas where applying frequent itemsets mining is very helpful:

1. **Basket Analysis:** Determining the frequent patterns in users' purchasing histories. It requires FIM to handle transaction datasets.

2. **DNA sequence analysis:** Determining which parts of a DNA sequence have similar functions. It requires FIM to do stream mining.

3. **Context analysis in Websites:** Finding the related pages on network, such as search engine. It requires the use of FIM to process online data and make real time analysis.

4. **Abnormal events detection:** Detecting the outlier from previous results. It requires FIM to process audio and video data.

Many FIM algorithms like Apriori [AS94] and FP-growth [HPY00] have been proposed and have proved to be efficient in recent decades.

## 1.3 The "Big Data" Problem

The increasing size of modern applications that are producing huge amounts of data everyday leads to a new challenge to data mining known as the "big data" problem. Not only is the volume of data huge, but the information is also generated at enormous speed. The speed is faster than the current system can process. Researchers often use the five V's (Volume, Velocity, Variety, Veracity and Value) to describe the features of "Big Data". The interest in extracting patterns from a large collection of data has risen in both academic and industrial users. For instance, millions of people are using online social networks like Facebook or Twitter every day. Embedded in these user activities are useful information and knowledge [LJ15]. In one internet minute, 1,572,877 GB of global IP data are transferred, including 4.1 million searches with 10 million ads displayed [Cor13]. The speed of information generation is still accelerating, and it is expected that the number of connected devices will be three times more than the number of people on earth in 2017.

It has been shown that classical data mining algorithms developed over the past decades may not suitable for the modern size of applications because the data cannot be held in the main memory [LWZ$^+$08, SEB13, ZJT13, QGYH14]. Another problem is that classical data mining algorithms may not reach the time requirements for current applications. If we cannot process or mine the data fast enough, we may never get the useful knowledge. In addition, as hardware becomes cheaper and cheaper, parallel and distributed systems composed by multiple computers are being widely used to cope with the big data problem. As the work is distributed across multiple computers, the entire system is much faster and able to process more data than a single computer. Several parallel computing frameworks like Open MPI, Hadoop MapReduce and Hadoop Spark have been used by both academic and industrial users recently. Based on these platforms, researchers have proposed parallel FIM algorithms such as YAFIM [QGYH14], PFP [LWZ+08].

## 1.4 Motivation

As multiple computers work simultaneously on a network, it is important to utilize the maximum load for a parallel computing algorithm. Existing parallel and distributed data mining approaches that have been adopted from previous single machine algorithms ignore the variety of datasets. A dataset can either be sparse or dense, or some parts of it can be sparse and other is dense. For example, "common friends" in an online social network can be dense if two people share many common friends. It can be sparse if two people do not know each other well. To get a good load balance, we should handle different density of datasets.

Another major challenge for a parallel computing algorithm is the communication cost among the network. How to distribute data and how to collect results efficiently between different computers, will affect the running time of a algorithm significantly.

Based on the above challenges, the following are some scientific hypotheses I investigate in this thesis:

1. Can we design an algorithm that produces a complete answer (i.e., finds all frequent itemsets)?

2. Can we design an algorithm that can be used in different densities of datasets?

3. Can we design an algorithm that is faster than existing algorithms?
4. Can we design an algorithm which can distribute workloads equally?
5. Can we design an algorithm with less communication cost than existing algorithms?

## 1.5 Thesis Statement

To answer the above questions, I design and implement—in this MSc thesis research—an efficient parallel computing algorithm to the purpose of big data analysis with following qualities:

1. To find all frequent patterns and their transactions.
2. To be scalable to different density of datasets.
3. To show the efficiency of using the vertical format for parallel computing.
4. To create a balanced workloads among workers.
5. To achieve a low communication cost.

## 1.6 My Contributions

I propose an algorithm called **scalable vertical mining (SVT)**, which can be applied to sparse or dense datasets.

### 1.6.1 Quality 1: Completeness

Most the existing algorithms using a horizontal approach will only show the frequent patterns and number of transactions that appeared together. However, they cannot reveal which transactions these frequent patterns occur in because that would cause a lot of communications. I adopt a vertical mining approach that not only shows the frequent patterns with their counts but also reveals the transactions that these patterns are found in. Researchers found that those unpopular items in the same transactions with frequent items are also useful for a recommendation system [LSY03].

### 1.6.2 Quality 2: Scalability

As the name indicates, I want to make this algorithm to be more flexible to different densities of dataset. My algorithm can switch to different data structures base on characteristic of datasets. For example, if the current level of frequent patterns are sparse it use the Eclat [Zak00] model, and when the frequent patterns are dense it use the dEclat [ZG03] model as it maintains efficiency through measuring density of datasets at each level of mining progress. Thus, it can be used in either dense datasets or sparse datasets. In addition, this self-adjusting measurement can be done separately on each processing unit. In different part of datasets, different schemes may be applied to take advantage every computer. Each worker can calculate results simultaneously and independently without additional communication.

### 1.6.3 Quality 3: Efficiency

Through the evaluation part, SVT shows the competitive run-times and also minimum memory requirement. I present a performance comparison between my proposed algorithm and other parallel mining algorithms like YAFIM, PFP, MREclat and BigFIM. Two types of data are involved in this thesis. Synthetic datasets are used to measure scalability and real life dataset are used as benchmark to compare with existing algorithms.

### 1.6.4 Quality 4: Load Balance

SVT is optimized for balancing the distribution of workloads across a computing network and avoid overload on single node. Instead of taking a fixed number of prefixes to balance distribution are used in previous algorithms, I propose to use a dynamic strategy based on the capacity of current working units. SVT will determine the best time for distributing data. Compared with existing algorithms, it provides a better load balance among different density of dataset. This algorithm can be applied to multiple scenarios. It does not matter whether the datasets are sparse or dense, or if it is unbalanced. This algorithm will try to achieve high efficiency on each scenario by applying different strategies.

### 1.6.5 Quality 5: Cost

Economy is achieved by fully distributed with low communication cost. Each worker will work individually at certain levels without any extra communications. And each worker can choose their strategies separately based on the density of datasets without additional information. Instead of transmitting all data to other workers we can just transmit the local sum instead of all data. With a large amount of data, the differences for the shuffled data will be significant.

## 1.7 Thesis Organization

The remainder of this thesis is organized as follows. To make this thesis self-contained, I produce some background information about frequent pattern mining (which includes a comparison between horizontal data format and vertical data format) and a parallel computing framework (which is relevant to this thesis) in Chapter 2. Chapter 3 reviews some related works on notable data mining algorithms in the big data environment and their advantages and disadvantages. Chapter 4 presents the details of my proposed algorithm for processing big data and discusses how I balance the load and reduce the communication cost. Chapter 5 shows the experimental results and comparisons with existing algorithms. Finally, conclusions are drawn in Chapter 6.

# Chapter 2

# Background

In this chapter, I first introduce some preliminary notations about frequent pattern mining. Frequent patterns are a model that targets the items that frequently appeared together in the database. Here, a "frequent" itemset means that an itemset having a support or frequency satisfying the user-defined threshold, and frequent pattern mining (FIM) is the procedure for finding this useful model. This includes what kind of data we are analyzing based on what kind of rules that we use to do the mining and how we represent the results. In the second part, I will talk about a new generation of big data processing framework—Spark, and how it outperforms the previous frameworks like Open MPI (an open source message passing interface implementation) and Apache Hadoop. Spark outperforms not only because it is faster and more scalable, although the authors of spark claimed it is at least 10 times faster than Hadoop MapReduce on disk and can be 100 times faster if it is running in memory. Unlike MapReduce in Hadoop, developers spend a lot of time on how to group the operations together to minimize the communication cost, but Spark provides a naive way solve this problem by using a resilient distributed dataset. It also inherits some good features from Apache Hadoop like fault tolerance and Hadoop distributed file system (HDFS).

## 2.1 Frequent Patterns

Consider a transaction database $D$ consisting of $n$ transactions $t_1, t_2, \ldots, t_n$ with $m$ domain items $I = i_1, i_2, \ldots, i_m$. Each transaction has one or more non-repeated domain items. You can also view this transaction database as a grocery receipt. It contains two parts; the first part is transaction ID and second part is domain items which are the purchased products listed on the receipt. In FIM, *support* can be defined as the fraction of transactions that contain an item. *Support count* is the absolute value for support. *Confidence* is the percentage of transactions containing itemset A also containing C. Accordingly, *minsup* is a user-specified support that describes whether an item or itemset (group of items) is frequent or not. An item or itemset $X$ is *frequent* if this item or itemset appeared at least *minsup* times (i.e., $support(X) \geq minsup$) in all transactions; otherwise, it is infrequent. The *projected* or *conditional* database of an item, is the set of transactions that include this item. The goal of frequent pattern mining is trying to find all the items that frequently appear together.

Here is an example to illustrate my previous comment and considering a transaction database like Table 2.1.

The support count for $\{a\}$ is denoted as sup($\{a\}$) whose value is 3, since it appeared in transaction 1, 2 and 3 separately, and the support for $\{a\}$ is $\frac{sup(\{a\})}{\#transactions}$ =

Table 2.1: An example of a transaction database with *minsup*=3

| TID | Transactions |
|---|---|
| 1 | $\{a, b, c, d, f, g, i, m\}$ |
| 2 | $\{a, b, c, f, m, o\}$ |
| 3 | $\{b, f, h, j, o\}$ |
| 4 | $\{b, c, k, p, s\}$ |
| 5 | $\{a, c, e, f, l, m, n, p\}$ |

60%. It is easy to see that sup($\{a, c, m\}$) is also 3 and sup($\{b, f\}$) is 2. If we set the minsup to 3, then items like $\{a\}$ and itemsets like $\{a, c, m\}$ are considered to be frequent and itemsets $\{b, f\}$ are considered to be infrequent.

### 2.1.1 Apriori Properties

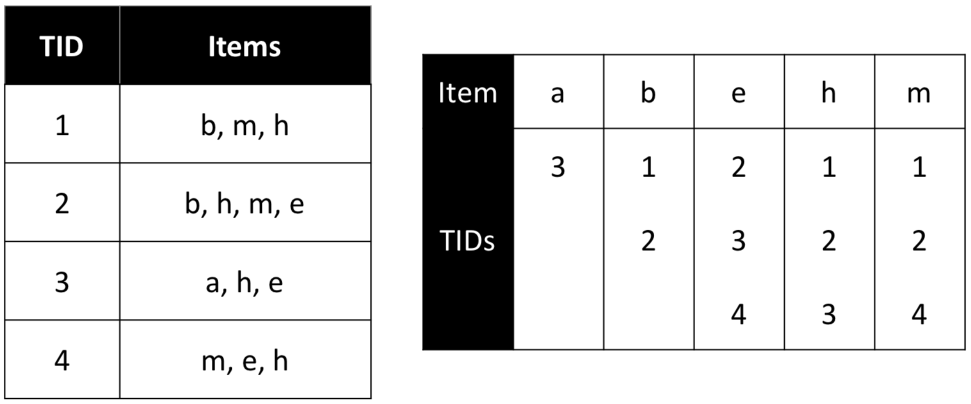
Apriori properties, also known as downward closure or anti-monotonic, are widely used properties to generate frequent patterns in frequent pattern mining. It was first mentioned in the Apriori algorithm [AS94] and soon became one of the fundamental theories in frequent pattern mining. We can derive the following two rules:

- **Rule 1:** If one subset is infrequent, then its supersets must be all infrequent.
- **Rule 2:** If one superset is frequent, then its subsets must be all frequent too.

It is not difficult to prove Apriori properties. Here, I will use a example to demonstrate the rules. In Table 2.1, we already know that if itemsets $\{b, f\}$ are infrequent, then any supersets containing $\{b, f\}$ must be in infrequent, for example $\{b, f, m\}$ or $\{b, f, p\}$, no matter the items $\{b\}$, $\{f\}$, $\{m\}$, $\{p\}$ are all frequent separately. We also know that if $\{a, c, m\}$ is frequent then any subsets of it must be frequent like $\{a\}$, $\{c\}$, $\{m\}$, $\{a, c\}$, $\{c, m\}$, $\{a, m\}$. The benefit of Apriori's properties is to greatly reduce the number of itemsets that we need to check; either they are frequent or infrequent. Here, it is also worth mentioning that the counter arguments are not valid. It is not guaranteed that if one subset is frequent, then its supersets will also be frequent. Also, if the superset is infrequent, then its subsets may still be frequent. We already see this from the example above.

## 2.1.2 Vertical Data Representation

Generally, there are two ways to represent datasets. One is using a horizontal format and the other one is using a vertical format. Transaction databases are also called horizontal datasets because in a transaction database each row represents one transaction. However, in a vertical format each row is considered as one domain item with all transactions that contain this domain item. Figure 2.1 shows the differences between two formats. In general, the horizontal dataset is used more widely for



| TID | Items |
|---|---|
| 1 | b, m, h |
| 2 | b, h, m, e |
| 3 | a, h, e |
| 4 | m, e, h |

| Item | a | b | e | h | m |
|---|---|---|---|---|---|
| TIDs | 3 | 1 | 2 | 1 | 1 |
| | | 2 | 3 | 2 | 2 |
| | | | 4 | 3 | 4 |

Figure 2.1: Horizontal format vs. vertical format

single-machine data mining algorithms. This is because it is easy to generate a tree-structure like FP-tree [HPY00] globally and tree mining is very fast. But to generate a global tree in a parallel mining algorithm is expensive. If this tree is too big, the main memory may not be able to hold it. So, the vertical data representation is more suitable for parallel computing.

## 2.2 Parallel Computing

The definition of parallel computing from Wikipedia is "*a type of computation in which many calculations are carried out simultaneously, or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.*" The computer processing resources are shared through the Internet. In the last decade, several distributed frameworks for parallel computing have been proposed. People also refer the services on these frameworks as "cloud" services like Amazon cloud, Microsoft cloud, etc. The MapReduce framework [DG08] is relatively new framework for the distributed computing model, which was originally proposed by Google as their searching service. It is designed for solving the big data issues and soon became a very popular framework for analyzing big data. Compared with other frameworks like Open MPI, the MapReduce framework simplifies the whole parallel computing procedure into two functions: map function and reduce function. In the map function, the original data is distributed into the cluster and transformed into key-value pairs. In the reduce function, the key-value pairs are collected from different computation nodes and combined to get the result. Apache Hadoop [Whi09] is a famous open-source implementation of the

MapReduce platform. It is designed for distributed computing together with managing distributed storage, which includes fault tolerance. It contains two roles, one is driver and the other is worker. There is only one driver (which spreads resources and collects results) but multiple workers (which are responsible for calculations simultaneously in one network). An overview on MapReduce is shown in Figure 2.2.

Researchers observed that Apache Hadoop sometimes takes a long time when processing interactive data analysis. The processing time ranged from hours to days to receive the results. So, a group of researchers in the UC Berkeley AMPLab started the Spark project in 2009 [ZCF$^+$10]. Instead of saving the intermediate results on disk, Spark stores these values in memory. It also extends the MapReduce framework to support more complicated computations like interactive queries and stream processing. Data scientists can use SQL to communicate directly with Spark to get results. Engineers can write Java (or Scala, Python, R) programs to handle complicated tasks. Since the Spark project is an open-source project and has received a lot of attention from company like Intel and Yahoo! It soon became a new trend for handling big data problems. Figure 2.3 shows a general structure for the Spark framework. Two roles involved in the Spark framework. One is driver program works as resources distributor and results collector. The other one is worker nodes work as computing units. Cluster Manger is also built in driver program.

### 2.2.1 Scalability

One of most important features that distinguish Spark from Hadoop is the resilient distribute dataset (RDD) [ZCD$^+$12]. It is an elastic structure that can be distributed

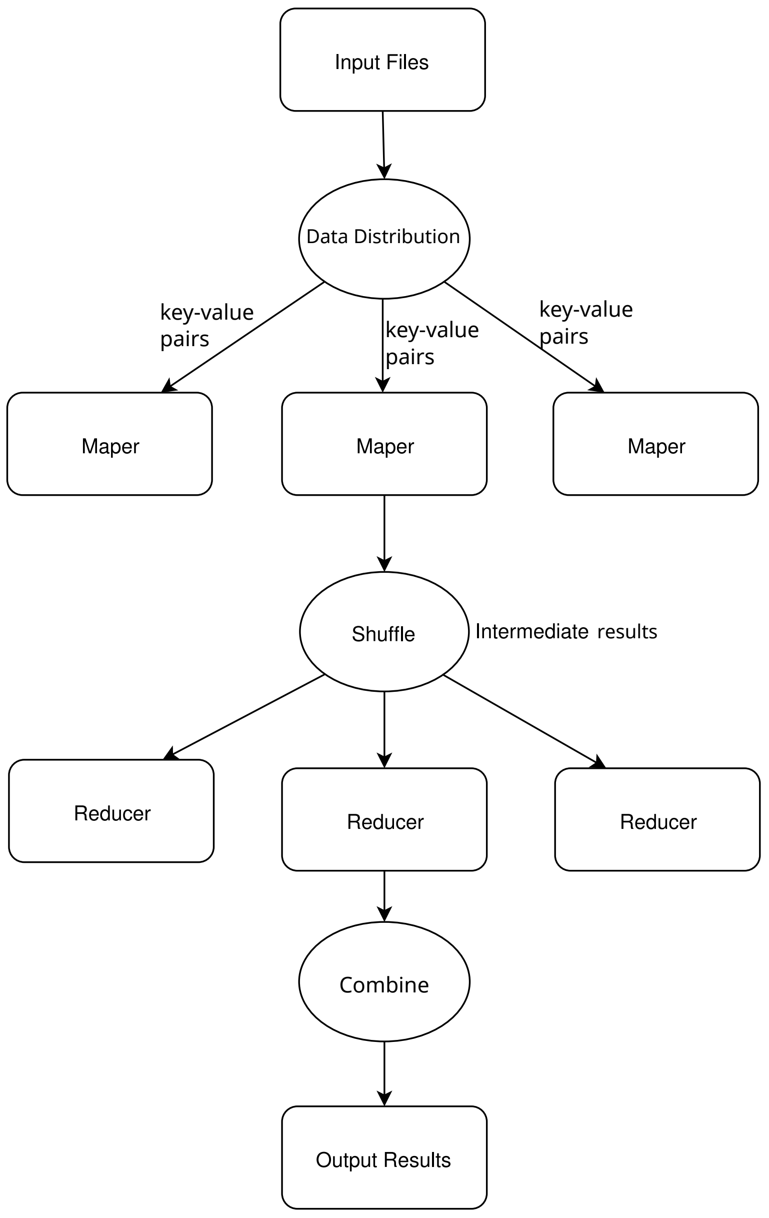


Figure 2.2: MapReduce overview

across different nodes. Basically, it is a container that holds the elements. RDD is immutable so that, once it is assigned, its value cannot be changed. It is elastic because it can be programmed by Python, Java or Scala directly.

Most of the time, we need to use RDD to do the calculations. There are two

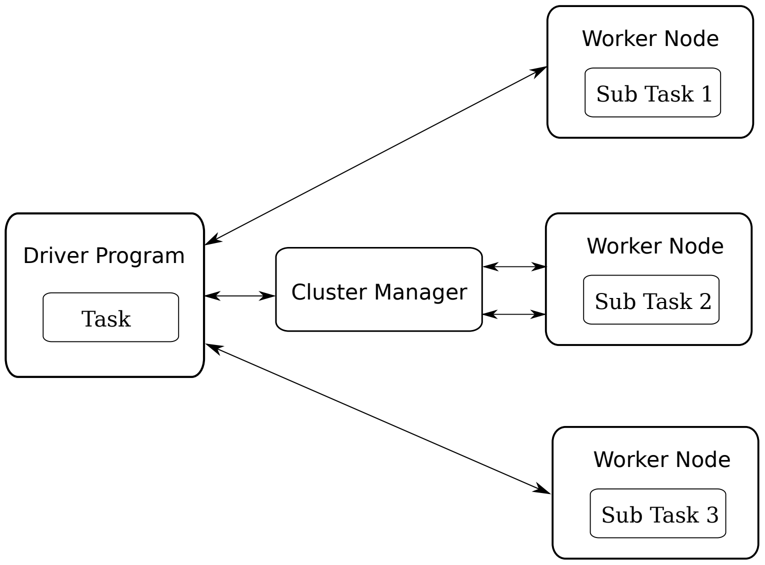


Figure 2.3: Spark overview

types of operations on RDD, one is transformations and the other one is actions. Transformations will create a new RDD from an old RDDḞor example, add 1 to every number in RDD1 will be return a new RDD2 contains every number + 1 in RDD1Ȧctions are another type of operations on RDD and usually this related to the driver. The transformations are "lazy" evaluated, they are not executed until an action happen. In the previous example, RDD2 contains a series of instructions on how to compute on RDD1, but there are no actual values in it yet until we call an actual action like **collect** to gather results. The advantages from RDD are compressed data on grouping distributed operations which saves communication time, and includes easily recovering from an error, which leads to another important feature in Spark.

### 2.2.2 Fault Tolerance

When there is a large quantity of data, fault tolerance will become a vital factor in making computations. Many situations can lead to error in this progress such as network error, disk error. Spark is designed to have an efficient fault tolerance. Each part of the data is duplicated and stored on different machines. If a node failure happens in a cluster, it will not affect the results. Only a short time is needed to recover from the previous checkpoint.

## 2.3 Summary

In this chapter, the aim of frequent pattern mining is to discover implicit, previously unknown and potentially useful knowledge—in the form of frequently occurring sets of items—from transactional data that are in horizontal or vertical representation. To discover knowledge from these data, parallel computing such as Spark is useful because it is scalable and fault tolerant.

# Chapter 3

# Related Work

Many parallel data mining algorithms are extended from data mining algorithms designed for a single machine. Classical algorithms for mining data in a single machine include Apriori and FP-growth (which both process horizontal datasets), as well as dEclat and VIPER (which both process vertical datasets). There are also extensions of parallel data mining such as YAFIM, PFP, MREclat and BigFIM. Although horizontal data representation is very popular for the classical serial algorithms that using single machines, vertical data representation can be more efficient for the parallel data mining algorithms.

## 3.1 Serial Algorithms

There is a long history of humans analyzing the useful patterns hidden in data. In general, there are two types of classical data mining algorithms. The first type is candidate generation algorithms on behalf of Apriori-based algorithms. This type needs to generate candidates before discarding the unnecessary ones. The second type is pattern growth algorithms on behalf of FP-growth based algorithms. The second type is more efficient in general since it does not need to generate extra candidates.

### 3.1.1 Apriori

Apriori [AS94] is the first approach that systematically depicts the problem of frequent pattern mining and uses downward closure to solve the problem. Apriori adopts a candidate-generation approach to recursively generate the candidates and then use anti-monotone rule to prune out infrequent items. The main pseudo code is in Algorithm 1 and there are two steps are involved in it:

- **Join step:** By doing breadth first search, two of the previous frequents are joined together.

**Algorithm 1** The Apriori algorithm

```
$L_1 = \{\text{frequent 1-itemsets}\}$
for $k = 2; L_{k-1} \neq \emptyset; k++$ do
    $C_k = \text{join } (L_{k-1})$
    for candidate $c \in C_k$ do
        count support as $sup(c)$
    end for
    $L_k = \{i \in C_k, \text{ if } sup(i) \geq minsup\}$
end for
return $L_k$
```

- **Prune step:** The algorithm scans the database to find the frequency of current candidates and discard the infrequent ones.

These two procedure run recursively until there is no candidate to generate.

The number of items in an itemset is called *size* of this itemset and an itemset of size $k$ is a $k$-itemset. $L_k$ is a set of frequent $k$-itemsets generated from the first phase. $C_k$ is a set of candidate $k$-itemsets that are potentially frequent.

The join function takes the set of frequent $(k-1)$-itemsets and generates the potential frequent $k$-itemsets. Two candidates are joined together if they have same $k-2$ prefix.

For example, let $k$ be set to 3. For the transactions database in Table 2.1, we generate 3-itemsets from 2-itemsets. We know that $\{a, b\}$ is frequent, $\{a, c\}$ is frequent, so in the join step we will get $\{a, b, c\}$. In the prune stage that we need to scan the database to verify if $\{a, b, c\}$ is frequent. Followed by anti-monotone property, if $\{b, c\}$ is not in 2-itemsets, then $\{a, b, c\}$ must not be frequent.

Further improvements for Apriori include the use of a hash table [PCY95] or using an upper bound [BMUT97] to reduce the number of scans. Algorithms adopting this strategy are called *candidates generation* approaches.

The Apriori approach is efficient because it uses the anti-monotone property, which reduces the number of candidate patterns. However, if the frequent patterns are long, these types of approaches are inefficient. A large number of candidates will be generated during the whole procedure which is a big challenge both for memory and space. For example, if there are 100 frequent 1-itemsets, the Apriori algorithm will generate more than 10,000 2-itemsets. Another disadvantage in Apriori is that it needs to scan the database multiple times to prune infrequent candidates. In the previous example, 10,000 2-itemsets need to be checked if they are in each transaction. This procedure could significantly decrease the performance if the datasets are large.

### 3.1.2 FP-growth

As it shows the Apriori based approaches cannot handle the long frequent patterns and are inefficient to check the support of candidates. a tree-based algorithm called FP-growth [HPY00] was proposed to overcome these limitations.

Instead of using a bottom-up candidate generation approach like Apriori, the FP-growth algorithm adopts a divide-and-conquer approach to finding frequent patterns. First, a scan of the databases will generate the frequent 1-itemsets. Items in frequent 1-itemsets in FP-growth are ordered in frequency ascending order. This order is important to the FP-growth algorithm to build a corresponding tree structure. Each transaction will contribute to building a small extended prefix-tree called a frequent pattern tree (FP-tree for short) which will be merged into a global FP-tree later. Compared to less frequent items, the more frequent items mean they have more common paths to share with other items.

The global FP-tree contains all the transaction information with frequent items. The second step is building the projection trees and mining the frequent patterns from this global FP-tree. Each projection tree represents one frequent pattern, which is the one or more branches from the global FP-tree. Mining global FP-tree is usually less costly than the Apriori because building projection trees only needs to follow the tree paths and these paths are compressed.

Table 3.1: A trimmed database with only ordered frequent items

| TID | Transactions |
|---|---|
| 1 | $\{b, c, f, a, m\}$ |
| 2 | $\{b, c, f, a, m\}$ |
| 3 | $\{b, f\}$ |
| 4 | $\{b, c\}$ |
| 5 | $\{c, f, a, m\}$ |

If we consider the trimmed transaction in Table 2.1 with only ordered frequent items. In the first scan of database, all domain items are arranged by frequency in a descending order, and we get $\{b\}$:4, $\{c\}$:4, $\{f\}$:4}, $\{a\}$:3, and $\{m\}$:3. As we just care about those frequent items, infrequent items can be removed. The resulting database is shown in Table 3.1.

Then, a FP-tree can be built from this information. Each time, the algorithm will read one transaction in frequency descending order. If there is already a branch contains these items, it just needs to increase the support for these items on the branch. Otherwise, a new branch is needed to create a composite these items and set their support to 1. It is obvious to see that the height of FP-tree is the maximum number of frequent items in one of the transactions in the database. Figure 3.1 is the global FP-tree build from this dataset.

After building the global FP-tree, the next step is mining frequent patterns by building small sub-trees called projection trees. To do this in a systematic way, the algorithm starts from the least frequent item, so it is guaranteed that no patterns will be missed. In the previous example, $\{m\}$ is least frequent, so it will start with building

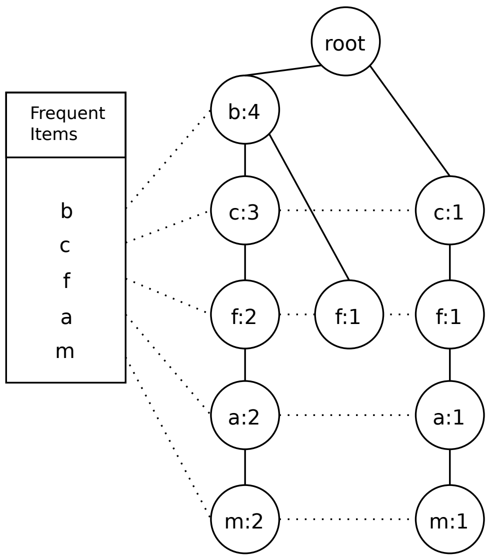


Figure 3.1: Global FP-tree

the $\{m\}$-projection tree. There are two branches containing $\{m\}$; they are $\{b$:4, $c$:3, $f$:2, $a$:2, $m$:2$\}$ and $\{c$:1, $f$:1, $a$:1, $m$:1$\}$. The first path indicates that the maximal sets $\{b, c, f, a, m\}$ appeared twice in the database. Here, although the support for $\{b\}$ is 4, the support for $\{b, c, f, a, m\}$ together is just 2. As 2-itemsets, support for $\{m, a\}$ is 3 and support for $\{m, f\}$ is 3. For 3-itemsets, support for $\{m, a, f\}$ is 3. In order to find the complete set of frequent patterns, it is necessary to follow frequent items list to mine each level recursively. Other sub-tree mining procedures are shown in Figure 3.2, 3.3, 3.4 and 3.5.

The algorithms based on FP-growth are called *pattern growth approaches*. FP-growth applied to many files such as uncertain data [LMB08] and high utility patterns [ATJL09]. The FP-growth is one of the fastest algorithms for mining frequent patterns, but it requires a large amount of memory to build the global FP-tree. If there are a lot of small patterns in the database, the FP-growth can be slow and memory inefficient.

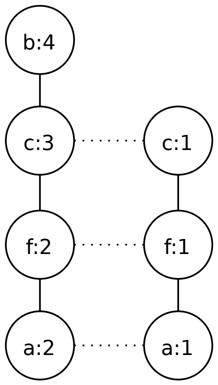


Figure 3.2: Mining the $\{m\}$-projected tree

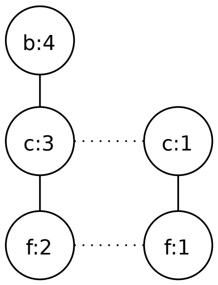


Figure 3.3: Mining the $\{m, a\}$- projected tree

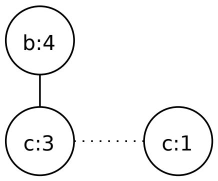


Figure 3.4: Mining the $\{m, a, f\}$- projected tree

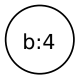


Figure 3.5: Mining the $\{m, a, f, c\}$- projected tree

### 3.1.3 Eclat

For some large and dense datasets, a new way to mine frequent patterns without building a global FP-tree in memory [Zak00] by taking advantage of vertical datasets. The pseudo code for Eclat is in Algorithm 2. There are two components of this algorithm:

1. Finding frequent items
2. Mining equivalent class

**Algorithm 2** Equivalence class transformation

```
$sup(x) = |tid(x)|$
**if** $sup(x) \geq$ minSup **then**
    $x \rightarrow L$
**end if**
**for** $x, y \in L_k$ **do**
    **if** $prefix(x) == prefix(y)$ **then**
        $tid(xy) = tid(x) \cap tid(y)$
        $sup(xy) = |tid(xy)|$
        **if** $sup(xy) \geq$ minSup **then**
            $L_{k+1} \rightarrow$ (xy)
        **end if**
    **end if**
**end for**
mine $L_{k+1}$
```

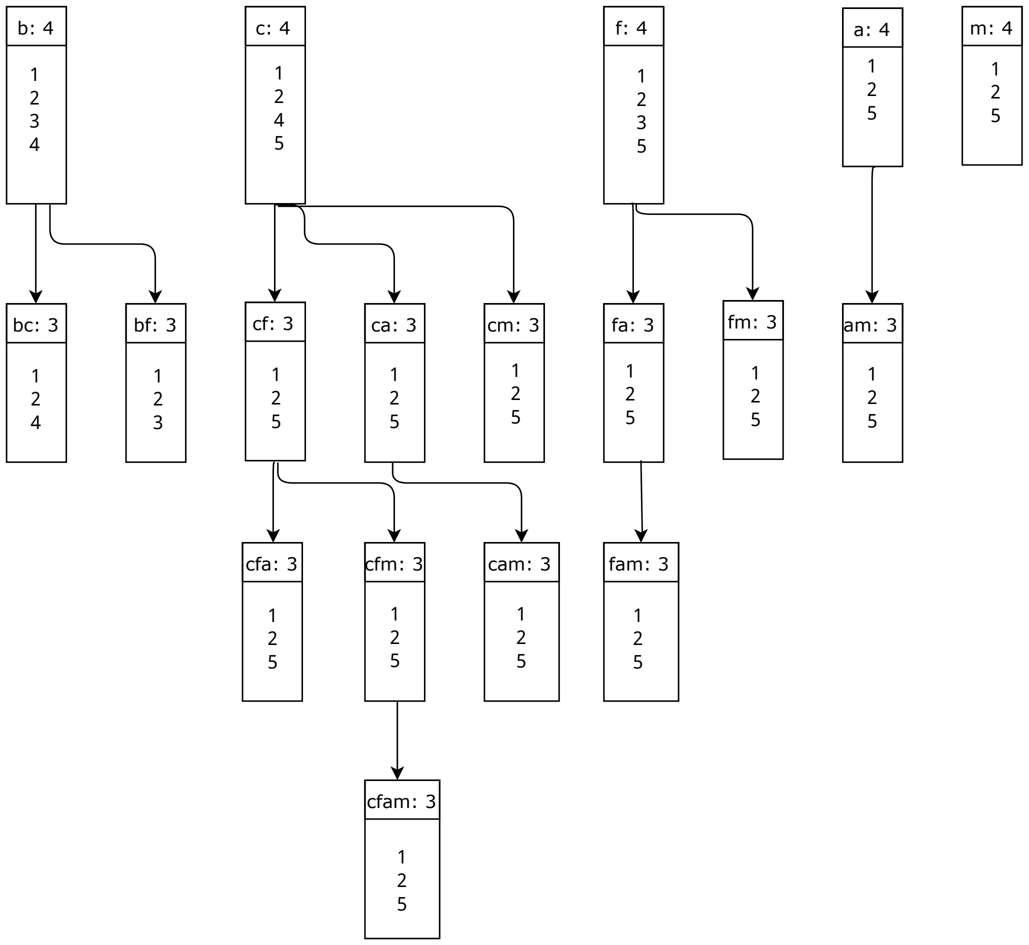


Figure 3.6: Eclat

In the first step, Eclat scans the input database entirely to calculate frequent items and then converts the frequent items into vertical format. Using the previous database in Table 2.1 as an example, results from the first step are $\{b$:1,2,3,4$\}$, $\{c$:1,2,4,5$\}$, $\{f$:1,2,3,5$\}$, $\{a$:1,2,5$\}$, $\{m$:1,2,5$\}$. The second step calculates the *equivalent class*. Equivalent class is a list of itemsets that have same prefix. For example, $\{ab\}$, $\{ac\}$ and $\{abc\}$ are in the same equivalent class since they have same prefix $\{a\}$.

After finding all the equivalent classes, Eclat just needs to mine each equivalent class recursively until there are no more frequent patterns in these equivalent classes.

As shown in Figure 3.6, there are 5 equivalent classes as $\{b\}, \{c\}, \{f\}, \{a\}, \{m\}$ to begin with. Each equivalent class can be mined separately.

Other improvements on using vertical data representation include the use of vectors to compress data like VIPER [SHS$^+$00] and the computation of differences between the sets as in dEclat [ZG03].

The major benefit of vertical mining approaches is that they need only to scan the database once in the whole procedure. In most cases, vertical mining approaches are faster than Apriori-based algorithms. However, since vertical mining approaches still need to generate candidates to calculate next level frequent patterns. So, it could be inefficient if frequent patterns are long. Another advantage of vertical mining approaches is flexibility; they can easily be extended to parallel computing frameworks.

## 3.2 Parallel Mining Algorithms

Due to the limitations of hardware, single node mining algorithms are generally not efficient for the big data. With the increasing size of data, parallel computing is becoming more and more popular for processing large amount of data. In this circumstance, several parallel frequent pattern mining algorithms have been proposed to extend serial algorithms to solve the big data problem. Some others proposed MapReduce-based frequent pattern mining algorithms.

### 3.2.1 YAFIM

YAFIM (Yet Another Frequent Itemset Mining) algorithm [QGYH14] is a parallel version of the widely-used Apriori algorithm that works as a memory-based computing model. It needs $k$ iterations of a MapReduce job ($k$ is the maximum length of frequent itemsets) to find all frequent $k$-itemsets using a Breadth-First-Search strategy. YAFIM is the fastest parallel version of Apriori so far, and works by taking advantage of the memory-based calculation framework from Spark [ZCF+10]. When compared with other parallel Apriori algorithms, YAFIM does not need to store the intermediate results on the disk, which can save a lot of I/O operations. Additionally, a Hash-Tree was used to accelerate candidate verification. Figure 3.7 shows the flow of the YAFIM algorithm. In the first stage, transaction datasets are loaded from HDFS into the Spark RDD. All the items are counted through the MapReduce framework, then it filters out infrequent items by looking at user-defined *minsup*. In the second stage, it needs to generate all the candidates and prune infrequent candidates. All the frequent items are collected from the driver to build a hash tree to speed up the candidate generation and verification procedure. When compared with the single-node Apriori algorithm, YAFIM can handle bigger sizes of data and is faster, depending on the setup of the cluster. Candidates are generated in the distributed workers and only the frequent items or itemsets in the driver. So, it lowers the big demanding for memory in one machine.

Like other Apriori-like algorithms, the bottleneck with YAFIM is that it must collect all the information from the previous level to do the next level of mining. When the dataset size is huge and dense, mining can be quite slow because some of the workers who finish their work earlier must wait those that have not finished yet.

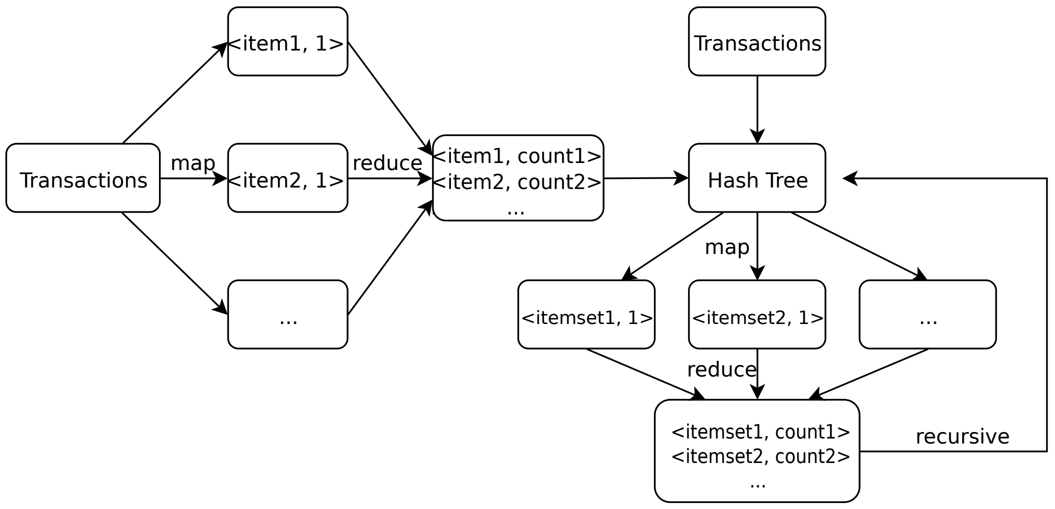


Figure 3.7: YAFIM

### 3.2.2 PFP

PFP (Parallel FP-growth) [LWZ+08] is a parallel version modified from the FP-growth algorithm for query recommendation. PFP is featured as "partitions computation" that each machine can work on an independent sub-mining tasks. This strategy helps to eliminate the computational and communication costs between each working machine. There are five steps for PFP. There are five stages of computation in PFP, which are:

1. Step 1: Sharding. This divides the database into small partitions and assigns these partitions to different workers in the cluster.

2. Step 2: Parallel Counting. This uses the MapReduce framework to count the support of singletons. First map each item as 1 and later merge all the items with same item.

3. Step 3: Grouping Items. Find the frequent singletons and make a group-

Table 3.2: An example of PFP—Part 1 Mapper

| Original Transactions | Transactions with Only Frequent Items | Conditional Transactions |
|---|---|---|
| $\{a, b, c, d, f, g, i, m\}$ | $\{b, c, f, a, m\}$ | $m$: $\{b, c, f, a\}$<br>$a$: $\{b, c, f\}$<br>$f$: $\{b, c\}$<br>$c$: $\{b\}$ |
| $\{a, b, c, f, m, o\}$ | $\{b, c, f, a, m\}$ | $m$: $\{b, c, f, a\}$<br>$a$: $\{b, c, f\}$<br>$f$: $\{b, c\}$<br>$c$: $\{b\}$ |
| $\{b, f, h, o\}$ | $\{b, f\}$ | $f$: $\{b\}$ |
| $\{b, c, k, p, s\}$ | $\{b, c\}$ | $c$: $\{b\}$ |
| $\{a, c, e, f, l, m, n, p\}$ | $\{c, f, a, m\}$ | $m$: $\{c, f, a\}$<br>$a$: $\{c, f\}$<br>$f$: $\{c\}$ |

dependent transaction based on the frequent singletons.

4. Step 4: Parallel FP-growth. Map the original transactions into small independent FP-trees based on the previous groups.

5. Step 5: Aggregating. Collect and merge independent FP-trees and find frequent patterns by searching conditional trees.

Consider an example based on the database shown in Table 2.1. The first part is the "mapping", which generates key-value pairs. The mapper is shown in Table 3.2. The second part is the "reducing", which merge the results from previous mapper with same key. The reducer is shown in Table 3.3.

Table 3.3: An example of PFP—Part 2 Reducer

| Reduce inputs | Conditional FP-trees | Reducer outputs |
|---|---|---|
| $m$:$\{b,c,f,a\},\{b,c,f,a\},\{c,f,a\}$ | $\{c,f,a\}$:3 $\mid m$ | $\{m,f,c,a\}$:3 |
| $a$:$\{b,c,f\},\{b,c,f\},\{c,f\}$ | $\{c,f\}$:3 $\mid a$ | $\{a,c,f\}$:3 |
| $f$:$\{b,c\},\{b,c\},\{b\},\{c\}$ | $\{b:3,c:3\} \mid f$ | $\{f,b\}$:3, {f,c}:3 |
| $c$:$\{b\},\{b\},\{b\}$ | $\{b:3\} \mid c$ | $\{c,b\}$:3 |

### 3.2.3 MREclat

While PFP is a fast frequent-pattern algorithm in a general, it is not optimized for balancing loads in a cluster. For example, if more frequent patterns start with one common prefix, all the calculations can be calculated in one worker while the other workers are idle. Another improvement includes balancing the load use frequency of items [ZZC$^+$10]. A later study [SEB13] showed that this is not the best solution.

In some situations, the datasets can be dense, which means there are a lot of common items in each transaction. If we use rule generation based algorithms like PFP, they will create a lot of branches and later cause a lot of communications across the cluster. Also, some branches may be so big they cannot fit in the memory. Hence, a new way was proposed to use Eclat to solve the big data problems by taking advantage of vertical representation of datasets. The resulting algorithm—called MREclat (MapReduce based Eclat) [ZJT13]—started with a horizontal database and later converts it to a vertical database. Instead of converting frequent 1-itemsets to a vertical database, they claim that starting with frequent 2-itemsets is faster and more memory-efficient. In order to balance the load, they also proposed to use $w_i = \lg(n-1) + \lg(\sum_{j=0}^{j<n} len_j)$, where *len* is the length, to depict the weight

of equivalent classes with the same prefix. Here, $n$ is the number of frequent 2-itemsets in this equivalent class. This approach is faster than previous horizontal based approaches. They demonstrated an increased speed when they increased the nodes in the cluster. But, they assumed that 2-itemsets with the same prefix will fit in the memory all the time, and that is not always the case. This varies in situations with different densities of data.

### 3.2.4 BigFIM

To solve the problem of some parts of cluster do not have enough memory to process data, two density-based self-adjusting algorithms called Dist-Eclat and BigFIM [SEB13] were proposed. There are three phases in both algorithms. In the first stage, they just calculate the frequent singletons with one MapReduce routine. In the second stage, they keep splitting the datasets until they find the equivalent classes that fit the current circumstance. Dist-Eclat builds some independent sub-trees by applying the classical eclat algorithm while the BigFIM adapts a bottom-up strategy like the Apriori algorithm to build these sub-trees. Then in the third stage, they mine independent sub-trees separately and in parallel. To reduce the communication cost, instead of using distributed data space they divide the search space. In other words, they only need to send out results instead of whole datasets, so there is no extra communication cost between workers. They also claim that Eclat with a diffset format is more efficient for mining large datasets. They show that assigning a longer prefix to workers will gain a better load balance. Their experiments showed that Dist-Eclat is faster than BigFIM although BigFIM can handle a larger dataset than

Dist-Eclat.

They did not consider the performance of different densities. Instead of just using one format, it is reasonable to use different representations for different situations. Even in the same data, some parts may be dense and some parts may be sparse because the datasets not balance.

## 3.3 Summary

Existing serial frequent itemsets mining algorithms like Apriori, FP-growth and Eclat, etc. showed efficiency in a variety of applications. But, they are no longer efficient when handling big data because either of the large memory requirement or the long computation time. Several parallel frequent itemsets mining algorithms like YAFIM, PFP and BigFIM, etc. are proposed recently to handle the big data problem. They are efficient in some datasets but not flexible. Tables 3.4 and 3.5 show the key differences among these algorithms.

In this chapter, I reviewed and analyzed related works. Understanding the advantages and disadvantages of these serial or parallel frequent pattern mining algorithms,

Table 3.4: Analysis 1 on algorithms

| Name | Based Algorithm | Data Presentation | Data Distribution |
|---|---|---|---|
| YAFIM | Apriori | horizontal | sparse |
| PFP | FP-growth | horizontal | dense |
| MREclat | Eclat | vertical | sparse and dense |
| Dist-Eclat | dEclat | vertical | sparse and dense |
| BigFIM | Apriori and dEclat | vertical | sparse and dense |

Table 3.5: Analysis 2 on algorithms

| Name | Search Strategy | Communication Cost Handled | Load Balance Handled |
|---|---|---|---|
| YAFIM | Breadth First | Yes | No |
| PFP | Depth First | Yes | No |
| MREclat | Depth First | Yes | Yes |
| Dist-Eclat | Depth First | Yes | Yes |
| BigFIM | Depth First | Yes | Yes |

I design my algorithm for mining frequent patterns from big data. I maintain advantages of these algorithms while resolve the problems or disadvantages associated with these algorithms in the next chapter.

# Chapter 4

# My Proposed Scalable Vertical Mining

In this chapter, I present my **scalable vertical mining (SVT) algorithm**, which is a parallel mining algorithm for processing big data. As such, it can be applicable to a real-life dense or sparse datasets. My SVT algorithm also takes in account both load balancing and communication cost reduction.

## 4.1 Balancing Load

The aim of load balancing is to optimize the distribution of workloads across a computing network and avoid overload on single node. For any parallel computing algorithm, the total execution time will highly depend on the running time of the longest running working node. It is inefficient to have some units in a cluster working while the others are just waiting to get necessary resources because of an unbalanced

Table 4.1: An example of a transaction database with *minsup*=3

| TID | Transactions |
|---|---|
| 1 | $\{a, b, c, d, f, g, i, m\}$ |
| 2 | $\{a, b, c, f, m, o\}$ |
| 3 | $\{b, f, h, j, o\}$ |
| 4 | $\{b, c, k, p, s\}$ |
| 5 | $\{a, c, e, f, l, m, n, p\}$ |

schedule strategy. So, if I can reduce the time for the longest-running working node, I can reduce the total execution time. However, as the frequent itemsets are generated in the middle of the mining process and the counting of the number of frequent itemsets takes polynomial time computation [BG09], it is difficult to ensure that all the workers have exactly same amount of work. Inspired by both the Eclat and dEclat algorithms, I use prefixes to distribute frequent patterns into independent groups so that they can be mined separately on different machines.

When using prefixes to distribute frequent patterns into independent groups, I encounter a new problem: How to deciding the boundary for the prefixes? To solve this problem, I choose the prefixes with size $k - 1$ for $k$-itemsets for $k \geq 2$.

**Example 4.1** *Consider the database shown in Table 2.1, which I repeat in Table 4.1 for convenience. Then, I choose prefix 1 (i.e., common 1-itemset prefix "1") for 2-itemsets. The corresponding equivalence classes become* $\{b$*:1, 2, 3, 4*$\}$*,* $\{c$*:1, 2, 4*$\}$*,* $\{f$*:1, 2, 3, 5*$\}$*,* $\{a$*:1, 2, 5*$\}$*, and* $\{m$*:1, 2, 5*$\}$*.*

*Similarly, I choose prefixes 2 (i.e., common 2-itemset prefix "1, 2") for 3-itemsets. The corresponding equivalence classes become* $\{bf$*:1, 2, 3*$\}$*,* $\{cf$*:1, 2, 5*$\}$*,* $\{ca$*:1, 2, 5*$\}$*,* *{cm:1, 2, 5}, {fa:1, 2, 5}, {fm:1, 2, 5}, and {am:1, 2, 5}.*

Note that, if I were to choose the equivalence classes with longer prefixes, I would have a better balance because the job would be more evenly distributed. However, it would take more time to calculate the prefixes. Alternatively, I could also try to speed time in balancing frequent patterns having with frequent patterns having short prefix. However, it would increase the communication cost because I would then need to move patterns with short prefix to the working units. Hence, among these choices, *I distribute workloads equally to each working node in my algorithms.*

Previous studies [ZZC$^+$10, SEB13] show that load balance relies of the length of prefix. Hence, instead of taking a fixed number of prefixes to balance distribution as performed in many existing algorithms, *I propose to use a dynamic way to calculate the prefix based on the capacity of current working units.*

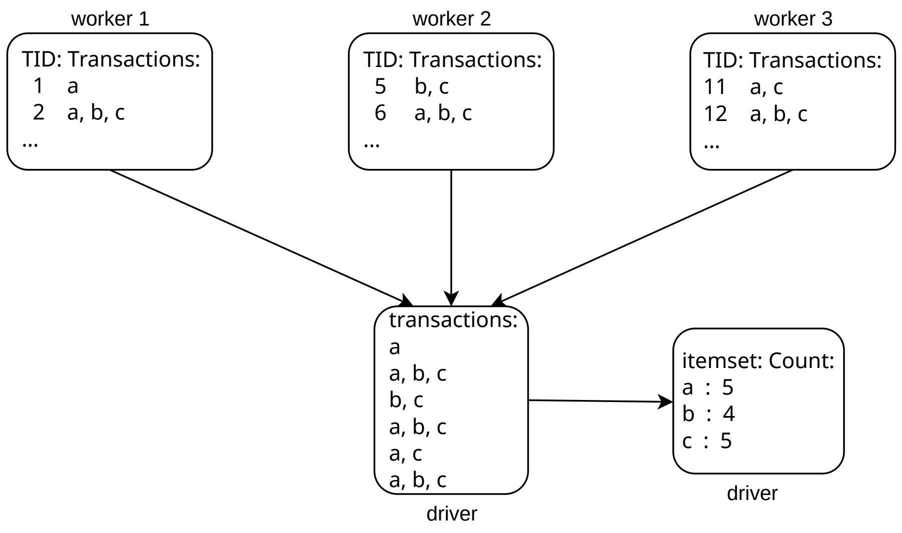
## 4.2 Communication Cost

Besides load balancing, another challenge for modifying a serial algorithm into a parallel or distributed system is the communication cost. Specifically, I measure speedup performance when increasing the number of computing units in parallel algorithms. Theoretically, if I use more computers, the calculation would be faster. But, practically, the performance of distributed computing system is often limited by the communication cost. Adding a new unit to the computing network may also increase the communication cost. So, the speedup with the number of computers is not necessarily linear.

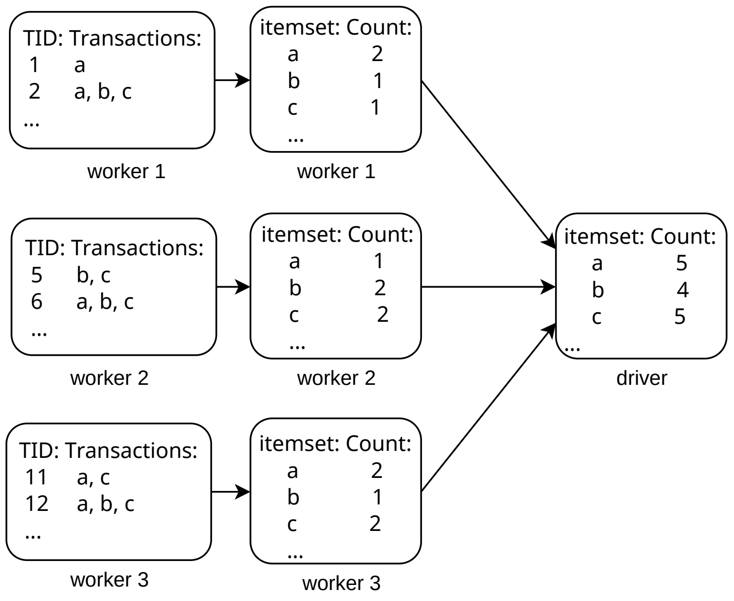
Recall from Chapter 2 that, in the MapReduce framework, there are two types of communications as shown in Figure 2.2. The first one happens when driver needs to collect calculation results from the workers. Most of the communication cost from this type of communication is inevitable. The second type of communication happens when I need to move data from one worker to another worker (i.e., the "shuffle" procedure). This "shuffle" procedure is expensive in the entire running time because it involves high disk I/Os, data serialization, and network I/Os due to the redistribution of resources. As it is not possible to completely skip the shuffle procedure, I need to apply some strategies to shuffle less data. The final goal is that, once I distribute resources at certain level, each worker needs to work as a "stand-alone" application separately without requiring any more resources.

My proposed frequent pattern mining algorithm is a recursive one. Each level of the mining process is highly reliant on the previous results. There are multiple shuffle procedures occurring in it. In the Spark framework, certain operations trigger a shuffle operation such as (a) re-partition operations, (b) ByKey operations and (c) join operations. My algorithm is optimized by using local computation results as a priority (instead of using a global calculation). Take a ByKey operation as an example. Data resources are stored across a cluster. If I want to calculate the support for the items in transitions, need to ask each worker to do the calculation and bring the results to the driver. But, instead of transmitting all data to the driver, I just transmit the local sum as shown in Figure 4.1. With a large amount of data, the difference in the reduction in the quantity of shuffled is significant.

My other way to optimize and reduce communication costs is by *using a dynamic data format between Eclat and dEclat* because dEclat mines less data than Eclat when



(a) the inefficient way



(b) the efficient way

Figure 4.1: Two ways to compute the support of itemsets

the datasets is dense, and vice versa. I also use bitmap to further reduce the size of the data to be transmitted in SVT.

More precisely, my proposed algorithm for mining frequent patterns from massive data in a parallel MapReduce framework is called **SVT**. SVT can be used for a general purpose whenever a dataset is dense, or sparse, by distributing calculations as evenly as possible.

## 4.3 Key Steps of SVT

Our proposed SVT consists of three key steps, as depicted in Figure 4.2. In the MapReduce framework, each of these key steps consists of the "map" function and the "reduce" function. Specifically, map function generates candidates; the reduce function distributes the computation and collects the results.

The first key step is finding the global frequent singletons among all distributed datasets. Then, the second key step is calculating the proper size for equivalence classes that can fit into workers' memory. Finally, the third key step is distributing the equivalence classes to different workers. This size varies for different situations. Unlike existing approaches that use a fixed number, here I get a dynamic value based on the maximum load of the current cluster. Moreover, vertical mining is simultaneously performed in each worker in this third key step. As a benefit, *SVT only needs to scan the database once* in the entire mining process

### 4.3.1 Key Step 1: Find Frequent Distributed Singletons

First of all, the data is serialized and distributed from driver to workers. The input transaction database is partitioned into equally sub-datasets and is also equally distributed among the workers. The sub datasets are known as *shards*. The shards

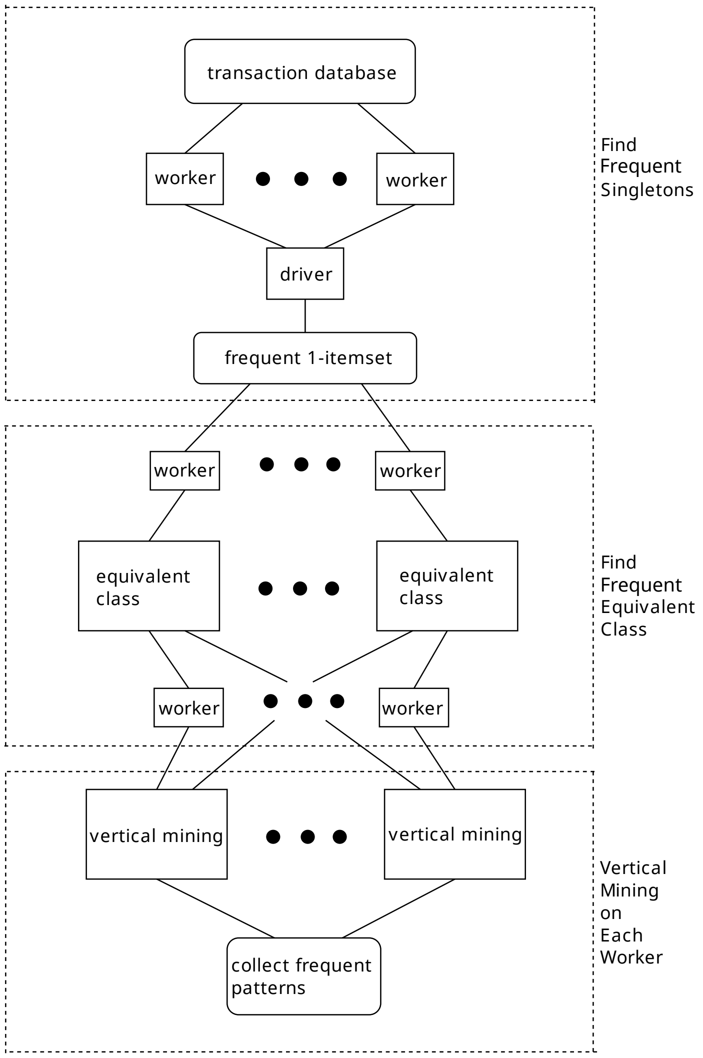


Figure 4.2: The overall SVT framework

in workers are in the form of key-value pairs. The key is item name and value is transaction number. After the work is evenly distributed each worker can work simultaneously. SVT calculates all frequent singletons (i.e., 1-itemsets) by counting the local singletons and then collecting all singletons together to find the items having frequency higher than or equal to the user-specified frequency threshold *minsup*. As mining process use a vertical data representation, it is also necessary to convert datasets from the usual horizontal format into vertical format. A local hash table is generated in each partition to accelerate this process. Each domain item and number of its transactions are stored as key-value (i.e., key=item, value=support) pairs in the hash table. Notice here, to reduce communication cost, most computation happen among workers. Then, workers calculate and send the local key-value pairs to the driver node. After aggregating all the keys, the driver node filters out the global frequent singletons that satisfy the user-specified *minsup* and broadcasts this frequent 1-itemset list to each processing unit for further usage. Algorithm 3 gives a skeleton of the first key step.

**Algorithm 3** Key Step 1 of SVT: Find frequent distributed singletons

```
parallelize(DB)
for transaction $T_i$ in transactions do
    map($T_i$) $\rightarrow \{item_k : i\}$
end for
for all workers do
    $C1$ = reduceByKey $\{item_k : i\} \rightarrow \{item_k : sup(k)\}$
end for
$L1 \leftarrow C1$.filter($item_k$, if $sup(k) \geq minsup$)
$L1 \leftarrow L1$.sortBy($L1.support$)
broadcast($L1$)
```

**Example 4.2** *Let us consider the previous transactions database in Table 4.1. Suppose that three workers in this computing cluster. Then, our SVT algorithm equally divides this transactions database into three parts:*

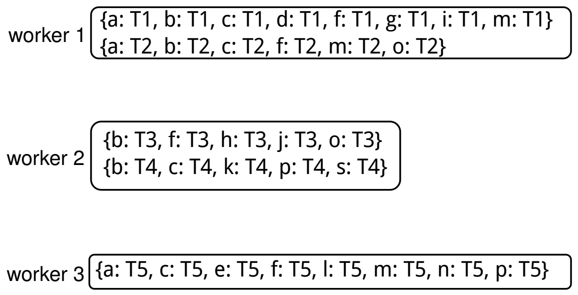

worker 1
{a: T1, b: T1, c: T1, d: T1, f: T1, g: T1, i: T1, m: T1}
{a: T2, b: T2, c: T2, f: T2, m: T2, o: T2}
worker 2
{b: T3, f: T3, h: T3, j: T3, o: T3}
{b: T4, c: T4, k: T4, p: T4, s: T4}
worker 3
{a: T5, c: T5, e: T5, f: T5, l: T5, m: T5, n: T5, p: T5}

Figure 4.3: Serialize the datasets and distribute to workers

1. *transactions 1 and 2 in Worker 1,*

2. *transactions 3 and 4 in Worker 2, and*

3. *transaction 5 in Worker 3.*

*After serialization, each worker stores one part of datasets as in Figure 4.3. With the "map" function, each worker emits a list of key-value pairs. Specifically,*

- *Worker 1 emits a list of key-value pairs {a:2, b:2, c:2, d:1, f:2, g:1, i:1, m:2, o:1};*

- *Worker 2 emits a list of key-value pairs {b:2, f:1, c:1, h:1, k:1, j:1, p:1, o:1, s:1}; and*

- *Worker 3 emits a list of key-value pairs {a:1, c:1, e:1, f:1, l:1, m:1, n:1, p:1}.*

*These workers send out their lists of key-value pairs to the driver node, as shown in Figure 4.4. With the "reduceByKey" function, the driver node combines values having the same keys. Consequently, it gets {a:3, b:4, c:4, d:1, e:1, f:4, g:1, h:1, i:1, j:1, k:1, l:1, m:3, n:1, o:2, p:2, s:1}. In this lists of key-value sum pairs, only a:3,*

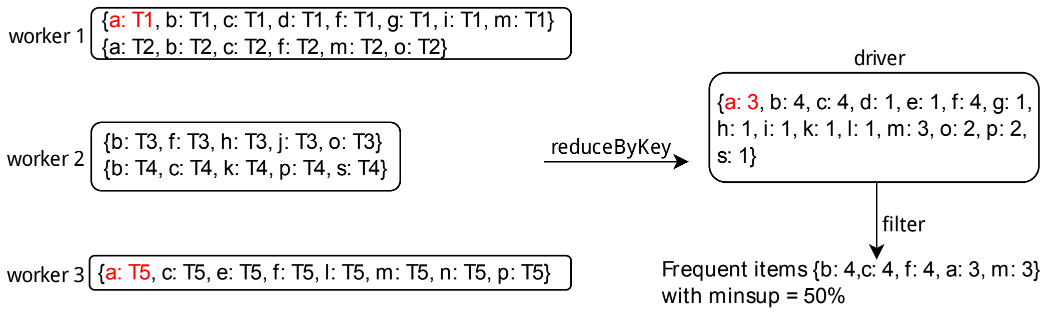
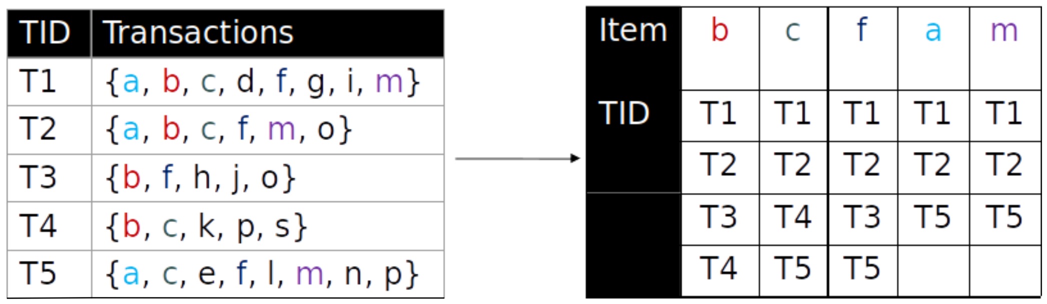

worker 1
{a: T1, b: T1, c: T1, d: T1, f: T1, g: T1, i: T1, m: T1}
{a: T2, b: T2, c: T2, f: T2, m: T2, o: T2}
worker 2
{b: T3, f: T3, h: T3, j: T3, o: T3}
{b: T4, c: T4, k: T4, p: T4, s: T4}
worker 3
{a: T5, c: T5, e: T5, f: T5, l: T5, m: T5, n: T5, p: T5}
reduceByKey
driver
{a: 3, b: 4, c: 4, d: 1, e: 1, f: 4, g: 1, h: 1, i: 1, k: 1, l: 1, m: 3, o: 2, p: 2, s: 1}
filter
Frequent items {b: 4,c: 4, f: 4, a: 3, m: 3} with minsup = 50%

Figure 4.4: Compute the local frequency and send it to the driver

*b:4, c:4, f:4, and m:3 are frequent when minsup = 3. This frequent-item list is then defined as a broadcasting variable, and each worker stores a copy of it.*

*At the end of the first key step, the input transaction datasets transformed from the horizontal datasets to the vertical format with only frequent itemsets, as shown in Figure 4.5,*



| TID | Transactions |
|---|---|
| T1 | {a, b, c, d, f, g, i, m} |
| T2 | {a, b, c, f, m, o} |
| T3 | {b, f, h, j, o} |
| T4 | {b, c, k, p, s} |
| T5 | {a, c, e, f, l, m, n, p} |

| Item | b | c | f | a | m |
|---|---|---|---|---|---|
| TID | T1 | T1 | T1 | T1 | T1 |
| | T2 | T2 | T2 | T2 | T2 |
| | T3 | T4 | T3 | T5 | T5 |
| | T4 | T5 | T5 | | |

Figure 4.5: Convert the horizontal dataset into vertical format with items sorted in descending frequency order

## 4.3.2 Key Step 2: Build Parallel Equivalence Class

After getting the global frequent 1-itemsets, the next key step is determining the size of equivalence classes ($k$-itemsets) that fit the memory of the working machines and distributing these equivalence classes to all the workers. A critical step in the

SVT algorithm is to manage the load balance. This size of equivalence classes may vary among different scenarios based on the density of the dataset and the capacity of the computation environment (e.g., the workers' memory). Each equivalence class is packed into key-value pairs for distribution. Now, the key is the prefix of the equivalence class, and the value is the list of candidate itemsets in this equivalence class. When distribute the equivalence class to each worker, there are two scenarios:

**Algorithm 4** Key Step 2 of SVT: Build parallel equivalence class

```
for all workers do
    $C_k = \langle L_{k-1}, L_{k-1} \rangle$
end for
$L_k$.redeceByKey($item_k : i$) $\rightarrow \{item_k : sup(k)\}$
$L_k \leftarrow L_k$.filter($item_k$, if $sup(k) \geq minsup$)
$m_k \leftarrow$ maximum size of candidate itemsets with same prefix
if $sizeof(m_k) \leq$ memory of single worker then
    $L_k \rightarrow EQ_k$
else
    calculate $L_{k+1}$
end if
```

1. If this worker already has a list of equivalence-class itemsets, then it is necessary to merge with previous transaction records for every itemsets in these two equivalence classes.

2. If this worker does not have a list of equivalence-class itemsets, then it just needs to build one with current itemsets.

At the end, each partition keeps one branch of the itemsets, which have the same prefix. Algorithm 4 gives a skeleton of the second key step.

**Example 4.3** *Continue with Example 4.2. Suppose 2-itemsets is the best size of equivalence class. Then, 1-itemset is used as a prefix for each equivalence class which is shown in Figure 4.6.*

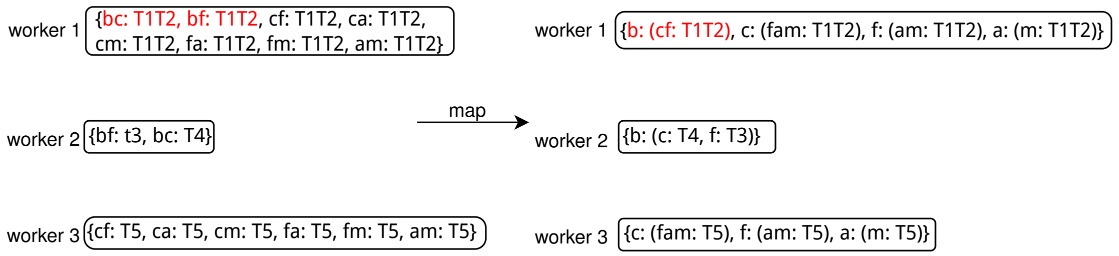

worker 1
{bc: T1T2, bf: T1T2, cf: T1T2, ca: T1T2, cm: T1T2, fa: T1T2, fm: T1T2, am: T1T2}
worker 2
{bf: t3, bc: T4}
worker 3
{cf: T5, ca: T5, cm: T5, fa: T5, fm: T5, am: T5}
map
worker 1
{b: (cf: T1T2), c: (fam: T1T2), f: (am: T1T2), a: (m: T1T2)}
worker 2
{b: (c: T4, f: T3)}
worker 3
{c: (fam: T5), f: (am: T5), a: (m: T5)}

Figure 4.6: Compute the proper size of prefix

*With the "map" function, SVT calculates a list of key-value pairs by performing inner products (i.e., dot products) of the frequent itemsets mined from the previous levels. As the prefix is the key in the key-value pairs and value is a list of frequent candidates, the results in this example are as shown in Figure 4.7:*

- *b:[c:124, f:123] because of b:1234, c:1245 and f:1235;*
- *c:[f:125, a:125, m:125] because of c:1245, f:1235, a:125 and m:125;*

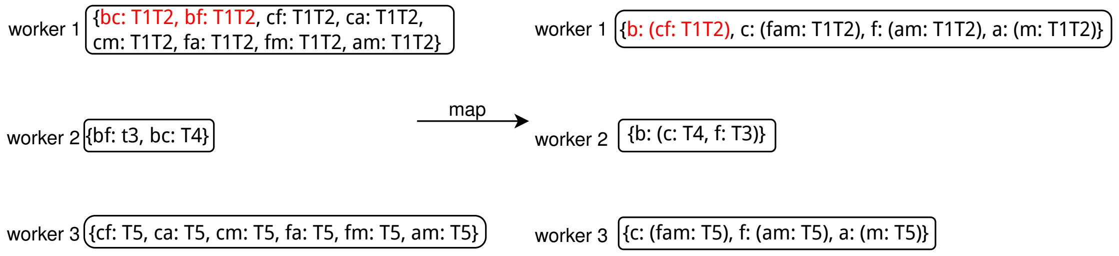

worker 1
{bc: T1T2, bf: T1T2, cf: T1T2, ca: T1T2, cm: T1T2, fa: T1T2, fm: T1T2, am: T1T2}
worker 2
{bf: t3, bc: T4}
worker 3
{cf: T5, ca: T5, cm: T5, fa: T5, fm: T5, am: T5}
map
worker 1
{b: (cf: T1T2), c: (fam: T1T2), f: (am: T1T2), a: (m: T1T2)}
worker 2
{b: (c: T4, f: T3)}
worker 3
{c: (fam: T5), f: (am: T5), a: (m: T5)}

Figure 4.7: Map datasets into independent equivalence class

- *f:[a:125, m:125] because of f:1235, a:125 and m:125;*
- *a:[m:125] because of a:125 and m:125.*

*These results represent the following* $2 + 3 + 2 + 1 = 8$ *frequent patterns:*

- *b:[c:124, f:123] represents two itemsets* $\{b, c\}$ *(which appears in transactions 1, 2 and 4) and* $\{b, f\}$ *(which appears in transactions 1, 2 and 3).*
- *Similarly, c:[f:125, a:125, m:125] represents three additional itemsets* $\{c, f\}$ *(which appears in transactions 1, 2 and 5),* $\{c, a\}$ *(which appears in transactions 1, 2 and 5) and* $\{c, m\}$ *(which also appears in transactions 1, 2 and 5).*
- *f:[a:125, m:125] represents two more itemsets* $\{f, a\}$ *(which appears in transactions 1, 2 and 5) and* $\{f, m\}$ *(which appears in transactions 1, 2 and 5).*
- *Finally, a:[m:125] represents the eighth itemset* $\{a, m\}$ *(which appears in transactions 1, 2 and 5).*

*This data structure is compact because the common prefix only appears once (e.g., prefix "c" only appears once for three itemsets) as shown in Figure 4.8.*

*With the "reduce" function, SVT distributes key-value pairs as equivalence classes to different workers. Notice that each worker may contain more than one equivalence class, which is calculated based on the work load capacity of each worker.*

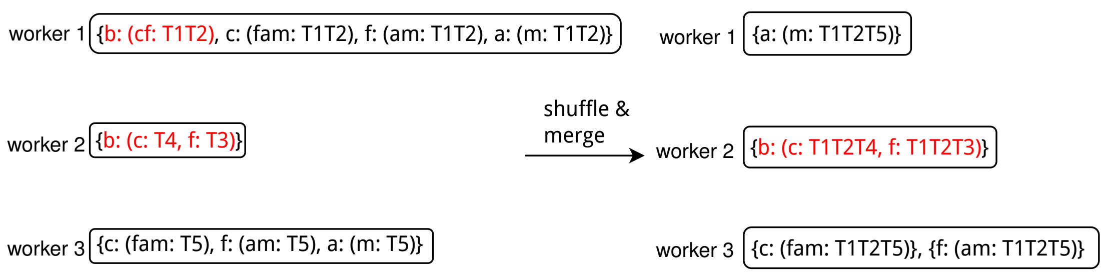

worker 1
{b: (cf: T1T2), c: (fam: T1T2), f: (am: T1T2), a: (m: T1T2)}
worker 2
{b: (c: T4, f: T3)}
worker 3
{c: (fam: T5), f: (am: T5), a: (m: T5)}
shuffle & merge
worker 1
{a: (m: T1T2T5)}
worker 2
{b: (c: T1T2T4, f: T1T2T3)}
worker 3
{c: (fam: T1T2T5)}, {f: (am: T1T2T5)}

Figure 4.8: Distribute equivalence class to workers

### 4.3.3 Key Step 3: Mine Local Equivalence Classes

The final key step is to distribute the original transaction dataset and to store the transactions as frequent equivalence classes in different units. With the "map" function, the mappers apply hybrid vertical mining algorithms on each partition without additional information from other workers. Different from traditional vertical mining algorithms like Eclat or dEclat, our SVT does not choose just a single strategy. It chooses different strategies based on the densities of datasets. Specifically, SVT first captures transaction IDs, which consumes less time in calculating the support. SVT then computes differences among the sets of transaction IDs (i.e., diffsets).

The switching from one strategy to another is based on the densities of datasets. If the dataset is dense, it switches from using transaction IDs to using diffsets early. On the other hand, if the dataset is sparse, it uses transaction IDs for longer period of mining time before it switches to diffsets. Our evaluation results suggest SVT to switch from using transaction IDs to using diffsets when the frequency of the subset is at least half of the superset.

For example, after the second key step in Worker 2, it contains two itemsets in one equivalence class. See Figure 4.9.

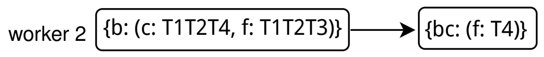


Figure 4.9: Use diffsets to mine equivalence classes

Another benefit of the switch is that, as each worker is working simultaneously, each worker may choose a different strategy based on the current system load.

Algorithm 5 gives a skeleton of the third and final key step.

**Algorithm 5** Key Step 3 of SVT: Mine local equivalence class

```
for all $C_i$, $C_j$ in equivalence class $EQ_k$ do
    $C_{ij} = C_i \cap C_j$
    $sup(C_{ij}) = |C_{ij}|$
end for
if $sup(c_{ij}) \leq 2 \times sup(C_k)$ then
    equivalence class transformation
else
    fast vertical mining using diffsets
end if
```

**Example 4.4** *Let us continue with Examples 4.2 and 4.3. As the following equivalence classes* $\{b{:}[c{:}124,\ f{:}123]\}$, $\{c{:}[f{:}125,\ a{:}125,\ m{:}125]\}$, $\{f{:}[a{:}125,\ m{:}125]\}$, $\{a{:}[m{:}125]\}$ *in each worker, SVT then computes the next level of frequent patterns with equivalence class transformations. The results are* $\{b, c, f\}$*:123,* $\{c, f, a\}$*:125,* $\{c, f, m\}$*:125}*, $\{c, a, m{:}125\}$ *and* $\{f, a, m{:}125\}$.

*When the support of* $\{c, f, a\} \geq 2\times$ *support of* $\{c, f\}$*, our SVT switches from using transaction IDs to using diffsets. Specifically, SVT computes diff*$(\{c, f, a, m\})$ $=$

$\{c, f, m\} - \{c, f, a\} = \emptyset$ *and* $sup(\{c, f, a, m\}) = 3$*. It is easy to see that, at this level,* *diff*$(\{c, f, a, m\})$ *requires less space than* $\{c, f, a, m\}$*.*

## 4.4 Summary

In this chapter, I presented our scalable vertical mining (SVT) algorithm for discovering frequent patterns from big data. The SVT algorithm consists of the following three key steps:

1. In the first key step, SVT applies the first set of "map" and "reduce" functions to find frequent singletons to be distributed or broadcasted to multiple workers. To elaborate, SVT performs the following procedures:

   1.1. SVT first serializes the datasets and distributes them to workers,

   1.2. then computes the local frequency and sends it to the driver with a reduced communication cost, and

   1.3. finally converts the dataset from the horizontal representation to its vertical format with items sorted in descending frequency order.

2. In the second key step, SVT applies the second set of "map" and "reduce" functions to build parallel equivalence classes. During such a building process, SVT balances the load and reduces communication costs to reduce the mining cost. To elaborate, SVT performs the following procedures:

   2.1. SVT first computes the proper size of prefix, 2.2. then maps the dataset into independent equivalence class with long names remapped into shorter ones to achieve reduction in communication cost, and

   2.3. finally distributes the equivalence class to workers.

3. Finally, in the third and final key step, SVT applies the second set of “map” and “reduce” functions to mine frequent $k$-itemsets (for $k \geq 2$) via the use of local equivalence classes. To elaborate, SVT performs the following procedures:

   3.1. SVT mines frequent itemsets vertically by first capturing transaction IDs for each itemset and then switching to capture differences between an itemset and its supersets (i.e., diffsets). The switching point is determined by the density of the input datasets.

   3.2. At the end of the mining process, SVT collects the mined results from workers to the driver.

As a result of the mining process, SVT discovers all frequent itemsets for big data analytics.

# Chapter 5

# Evaluation

In this chapter, I conduct a comparison between my proposed algorithm to other existing parallel mining algorithms like YAFIM, PFP and MREclat. Notice that some of those algorithms like YAFIM, MREclat were originally implemented under Hadoop platform. However, the difference of platforms may have a big impact on performance even for the same algorithm. To get the most reasonable results, I implemented those algorithms based on the published paper under the SPARK platform to the best of my knowledge. For evaluation, I measure running time, speedup, and memory usage.

## 5.1 Experimental Setup

In this section, we describe the environment and datasets for our experiments.

### 5.1.1 Experimental Environment

For this experiment, I have set up six computers to compose the Spark parallel computing cluster. Five computers in this cluster from the Helium system serve as workers, and one computer from the Database and Data Mining Lab serves as the driver. Each worker has 20 GB memory with 8 cores Intel Xenon processing unit. The driver has 8 GB memory and 4 Intel processing unit. All the machines are running on Linux and Spark 2.0.1 (latest version).

### 5.1.2 Experimental Datasets

Two types of data are involved in the evaluation part:

1. The first type is called *synthetic transaction dataset*, which benchmark datasets in the field for the evaluation of the scalability of algorithms. The goal is getting test data in different densities. There are three parameters that can be modified for the Synthetic Dataset Generator [FVGG$^+$14]:

   (a) number of transactions to be generated.

   (b) maximum number of distinct item that the database should contain.

   (c) number of items that each itemset should contain.

   Different combinations of input parameters can produce a dataset of different densities. For instance, by increasing the distinct items and limiting the length of each transaction, the generated datasets become dense.

   The parameters of synthetic datasets used in this thesis are listed in Table 5.1. In this table,

Table 5.1: Properties of benchmark synthetic datasets

| D | C | T | I |
|---|---|---|---|
| 10k | 20 | 0 | 0 |
| 10k | 0 | 25 | 10 |
| 100k | 0 | 20 | 6 |

- D indicates the number of transactions in the dataset,
- C indicates the average number of itemsets per transaction,
- T indicates average number of items per itemset and
- I indicates the average size of itemsets in potentially frequent transactions.

If any values are not specified, then they are defaulted as 0. These synthetic datasets are also available in the open-source data mining library (http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php).

2. The second type of data used in this thesis are benchmark real-life datasets found at Frequent Itemset Mining Dataset Repository [Mos13]. In most real-life scenarios, the density of a datasets is not listed. So, it is necessary to compare performance of different algorithms with different real-life datasets. The properties of datasets are in Table 5.2. In this table,

   - mushrooms datasets include descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms.
   - Accidents datasets is anonymized traffic accident data.

Table 5.2: Properties of benchmark real-life datasets

| Name | Size | Number of Transactions | Number of Items |
|---|---|---|---|
| Mushroom | 590K | 8,416 | 119 |
| Accidents | 34M | 340,183 | 468 |
| chainstore | 45M | 1,112,949 | 46,086 |

- Chain store datasets is a dataset of customer transactions from a retail store.

## 5.2 Analytical Evaluation

In this MSc thesis, I propose the SVT algorithm to find frequent patterns in the big data environment. Considering features that both distributed programs and data mining programs, I analytically evaluate my algorithm in the following three aspects:

1. speedup,
2. memory usage, and
3. runtime.

The first feature to evaluate is speedup. *Speedup* is defined as increasing the performance on execution of a task on multiple architectures with different resources. The ideal situation is a linear relation between running nodes and tasks. Due to the communication cost and resources management, it is impossible to reach that point. Amdahl's law [Rod85] gives the theoretical assumption for the speedup on execution of a task on a fixed workload.

The speedup for SVT is also affected by the density of datasets, because it will adjust to different data structures when the density of datasets changes in the middle of mining progress.

As for the execution time and memory usage, the performances for each algorithm are highly depended on the size and density of datasets. If the datasets are small and sparse, candidates generation algorithms such as YAFIM is fast. However, when the datasets is big and dense, the performance between each algorithm can be quite different. Our SVT is designed to process different density of datasets. Communication cost and resources management also have a significantly impact on the performance as well.

## 5.3 Experimental Evaluation

It is necessary to mention here, the first experiment I used static allocation policy in the Spark framework to show the speedup performance for the SVT algorithm. For the later experiments, I adopted a dynamic allocation policy in the Spark framework to get the maximum performance.

### 5.3.1 Speedup

The first experiment shows the speedup of the SVT algorithm for mining different datasets with SVT. The real-life datasets is the mushroom datasets with $minsup = 0.25$ and the synthetic datasets is the t25i10d10k datasets with $minsup = 0.008$. These datasets are carefully selected to get runtime in a same range (from 30 seconds to 90 seconds). Observed from Figure 5.1 that the total mining time is decreased almost

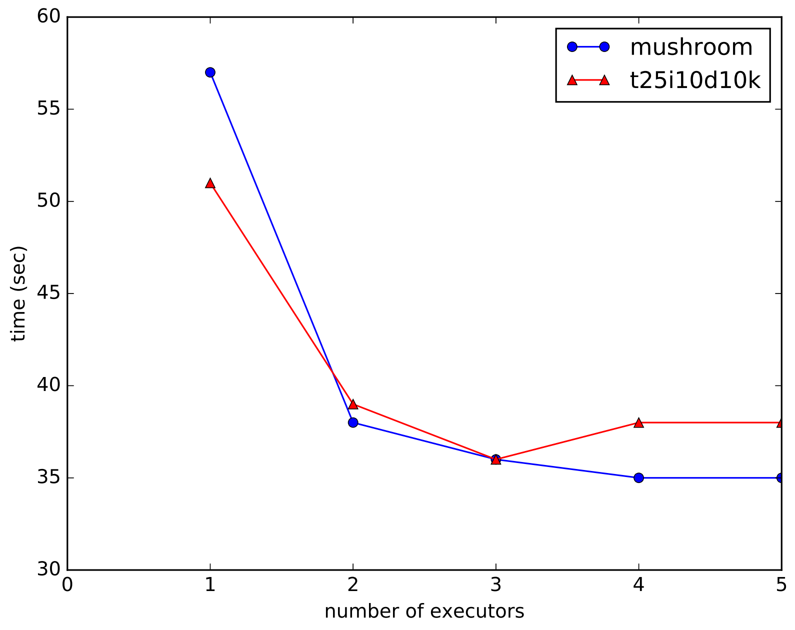


Figure 5.1: Speedup for SVT

by half when the number of executors are increased. However, the performance does not change a lot after the number of executors reached three. One reason is because of distribution policy in the Spark platform. Only minimum number of workers joins the current computation when the datasets fit into the workers' memory. So, three workers already have enough capacity to do current computation. Another reason is that communication cost between each worker is also increased when more workers are added. For example, in t25i10d10k datasets, the mining time is actually increased when there are more workers involved.

### 5.3.2 Evaluation on the Synthetic Datasets

The second part of evaluations compares the performances for different algorithms on the synthetic datasets. The first datasets is the c20d10k datasets and its properties is in Table 5.3.

There are 10,000 transactions in this dataset. The number of frequent items is increased as *minsup* is decreased since there are more items are considered as "frequent" when *minsup* is lower. There are twenty itemsets in each transaction on average. Figure 5.2 shows the performance of different algorithms for this dataset.

In this dataset, the PFP algorithm has the best performance in all. The SVT algorithm also shows a more stable performance compare to the YAFIM algorithm. Since the MREclat algorithm ran into out of memory problem at the beginning of this experiment, so it does not appear in this figure.

The second synthetic datasets is the c20d10k datasets and its properties is in Table 5.4. There are also 10000 transactions in this dataset. The average number of items per itemset is 25 and the average size of itemsets in potentially frequent transaction length is 10. This dataset is considered sparser compared with the previous

Table 5.3: Properties of the c20d10k dataset

| minsup | Number of Frequent Items | Maximum Length of Frequent Items |
|---|---|---|
| 0.5 | 1823 | 10 |
| 0.4 | 2,175 | 11 |
| 0.3 | 5,319 | 11 |
| 0.2 | 20,239 | 13 |
| 0.1 | 89,883 | 14 |

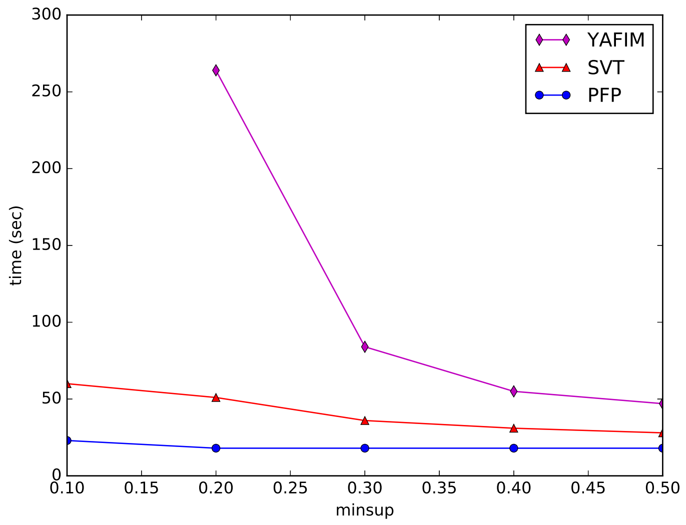


Figure 5.2: Evaluation on the c20d10k dataset

synthetic datasets.

Figure 5.3 shows the SVT algorithm has the best performance for this dataset in all. When *minsup* is less than 0.003, the number of frequent items increased much more than before. So, all the algorithms here rose sharply.

In order to see how the algorithms perform on a big data environment, I run

Table 5.4: Properties of the t25i10d10k dataset

| minsup | Number of Frequent Items | Maximum Length of Frequent Items |
|---|---|---|
| 0.02 | 553 | 2 |
| 0.01 | 2,893 | 7 |
| 0.009 | 10,483 | 12 |
| 0.008 | 14,782 | 13 |
| 0.005 | 302,284 | 16 |
| 0.003 | 1,502,825 | 18 |

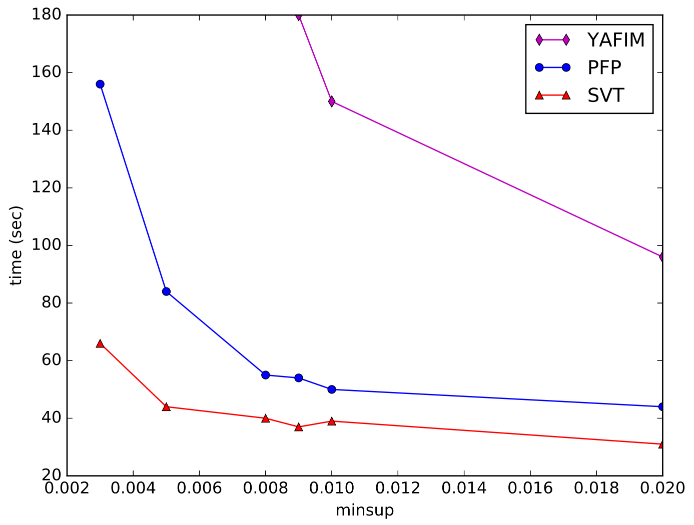


Figure 5.3: Evaluation on the t25i10d10k dataset

the experiment in a ten times bigger datasets. The properties of the t20i6d100k datasets is in Table 5.5. There are 100,000 transactions, the average number of items per itemset is 20 and the average size of itemsets in potentially frequent transaction length is 6.

Figure 5.4 shows the performance of different algorithms for this dataset. Notice that both MREclat and YAFIM cannot work with this dataset because the interme-

Table 5.5: Properties of the t20i6d100k dataset

| minsup | Number of Frequent Items | Maximum Length of Frequent Items |
|---|---|---|
| 0.05 | 26,836 | 11 |
| 0.04 | 41,614 | 12 |
| 0.03 | 118,577 | 14 |
| 0.02 | 199,609 | 14 |
| 0.01 | 393,708 | 14 |

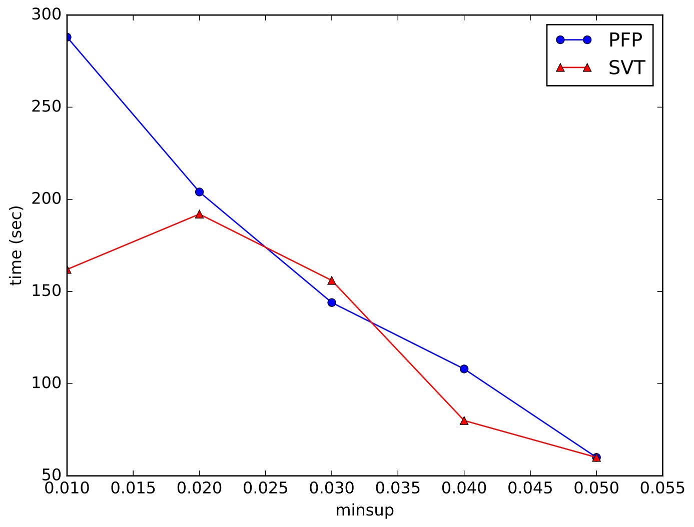


Figure 5.4: Evaluation on the t20i6d100k dataset

diate results are too big to be held in the memory. So I only test the PFP algorithm and the SVT algorithm for this dataset. The SVT algorithm still has the best performance in this larger dataset. As mentioned before, the density of datasets can change during mining progress. Even there are more frequent items were generated when *minsup*=0.01 than *minsup*=0.02, but the running time for this the SVT algorithm is decreased.

### 5.3.3 Evaluation on the Real-life Datasets

The third part of evaluations is comparing the performance on the real-life examples. The first real-life datasets is the mushroom datasets and its properties is in Table 5.6. There are 8416 transactions and 119 items in this dataset. Compared with the previous synthetic datasets, the mushroom datasets is denser and there are more frequent items were generated during mining progress with the same *minsup*.

Table 5.6: Properties of the mushroom dataset

| minsup | Number of Frequent Items | Maximum Length of Frequent Items |
|---|---|---|
| 0.9 | 11 | 3 |
| 0.6 | 51 | 5 |
| 0.5 | 163 | 5 |
| 0.4 | 505 | 6 |
| 0.3 | 2,587 | 9 |
| 0.25 | 5,393 | 11 |
| 0.2 | 53,337 | 15 |
| 0.1 | 600,817 | 16 |

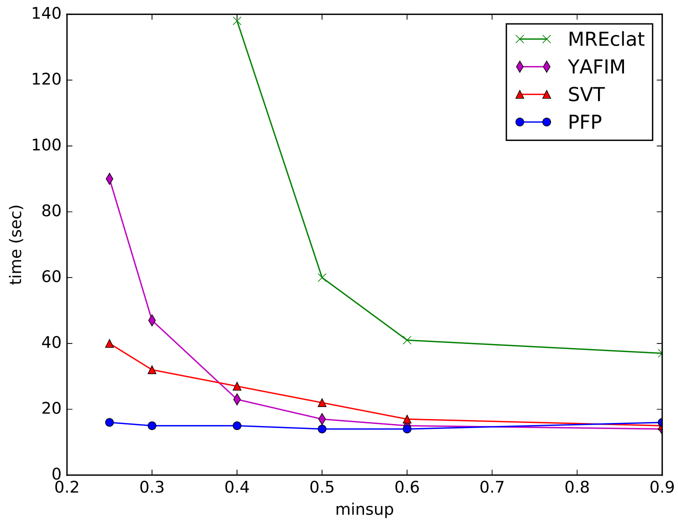


Figure 5.5: Evaluation on the mushroom dataset

Figure 5.5 shows the performance of different algorithms for this dataset. At the beginning, the difference between each algorithm is not too much. They need around 36 seconds to calculate the frequent patterns and collect the results. However when *minsup* reached 0.5, the computing time of MREclat rose sharply. After *minsup* =

Table 5.7: Properties of the accidents dataset

| MinSup | Number of Frequent Items | Maximum Length of Frequent Items |
|---|---|---|
| 0.95 | 15 | 4 |
| 0.9 | 31 | 5 |
| 0.8 | 149 | 6 |

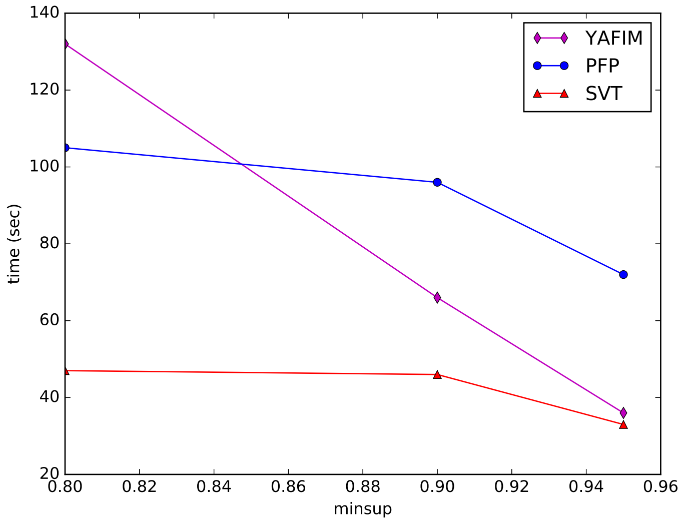


Figure 5.6: Evaluation on the accidents dataset

0.3, the YAFIM rose rapidly. Both SVT and PFP showed a steady rise with *minsup*.

The accidents datasets and its properties is in Table 5.7. There are 340183 transactions and 468 items in this dataset. The size of accidents datasets is much bigger than the size of mushroom datasets.

Figure 5.6 shows the performance of different algorithms for this dataset. Since the datasets is much bigger than the previous datasets, I had to use a high *minsup* as 0.96 to start with this experiment. It is easy to see that, with a small change of

Table 5.8: Properties of the chainstore dataset

| minsup | Number of Frequent Items | Maximum Length of Frequent Items |
|---|---|---|
| 0.01 | 54 | 1 |
| 0.009 | 54 | 1 |
| 0.008 | 62 | 1 |
| 0.007 | 81 | 1 |
| 0.006 | 95 | 1 |

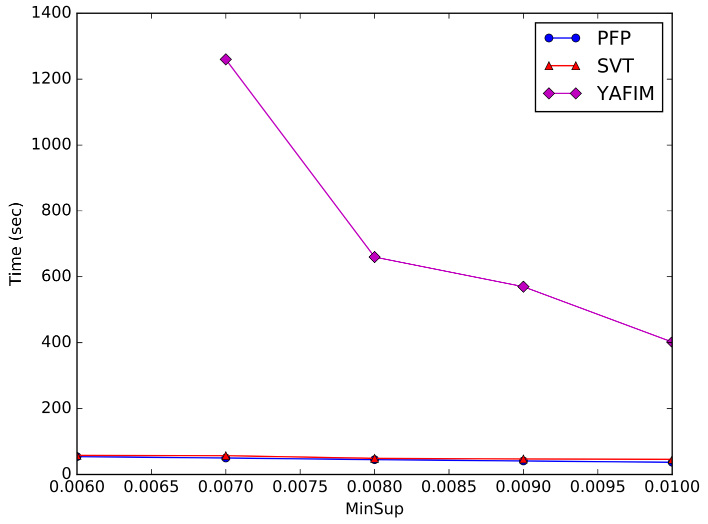


Figure 5.7: Evaluation on the chainstore dataset

*minsup*, both the PFP algorithm and the YAFIM algorithm shrunk a lot. The SVT algorithm shows a relative stable performance on the accidents datasets.

The last datasets is chain store datasets and its properties is in Table 5.8. There are 1,112,949 transactions and 46,086 items in this dataset. From the evaluation I found though the size of the chain store dataset is big but the length of frequent patterns is actually short. Even I already set *minsup* as low as 0.006, I still got the frequent patterns with length equals to one. The importance of datasets is not as important as others real-life datasets.

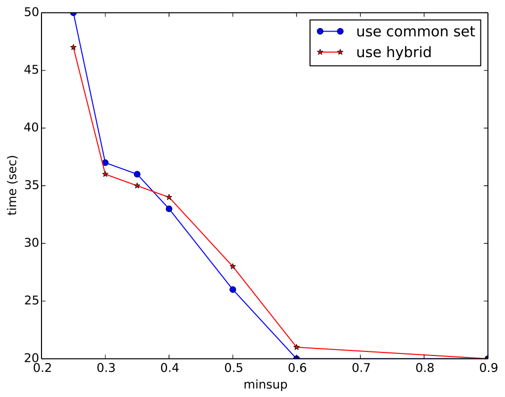
Figure 5.7 shows the performance of different algorithms for this dataset. The SVT has a competitive performance with the PFP algorithm.

## 5.4 Additional Experiments

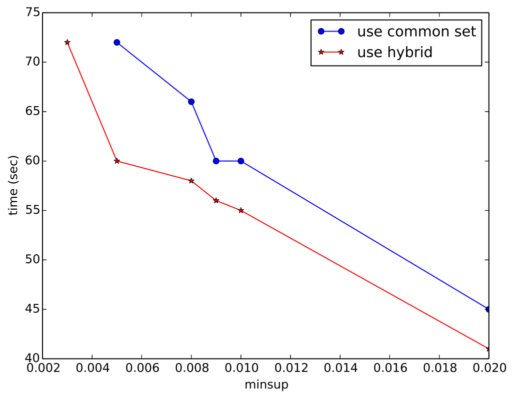
In order to quantify the weight of each factor in my proposed contribution, I also conducted three additional experiments to evaluate these factors.

### 5.4.1 Dynamically Switch between Common Sets and Diffsets

This experiment—as shown in Figure 5.8—compares the performance of using the common sets or using a hybrid of common sets and different sets in the SVT algorithm. Figure 5.8(a) shows the experimental result on the mushroom datasets. The two approach have similar performance in general. At the beginning, it is faster to only have common sets. But, when the minsup is smaller (say, less than 35%), the hybrid approach seemed to get faster than another approach because the hybrid approach is efficient if a superset and a subset share a lot of common parts. When the minsup is small, there are more frequent patterns are generated. Figure 5.8(b) shows the experimental result conducted on the t25i10d10k datasets. It is obsered that using the hybrid approach is faster than the approach using the common set.



(a) Mushroom datasets

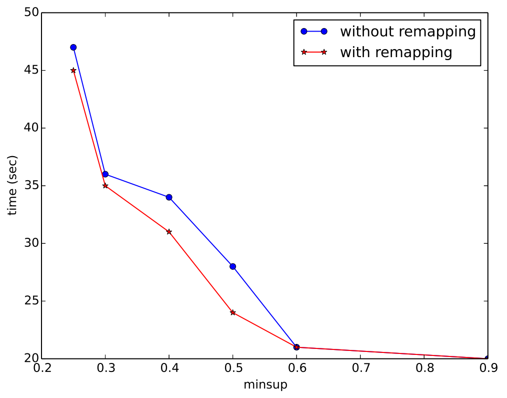


(b) T2t10d10k datasets

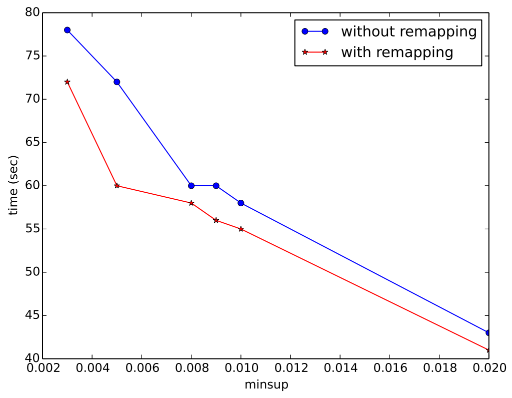
Figure 5.8: Dynamic switch between common set and difference set

### 5.4.2 Use Remapping to Reduce the Communication Cost

This experiment in Figure 5.9 compares the performance of using the remapping to reduce the communication cost in the SVT algorithm. No matter in the real-life datasets (as shown in Figure 5.9(a)) or in the synthetic datasets (as shown in Figure 5.9(b)), the mining speed is faster with using remapping long name to short name than without using remapping because short names generated from hash table are more condense compare with long names.



(a) Mushroom datasets

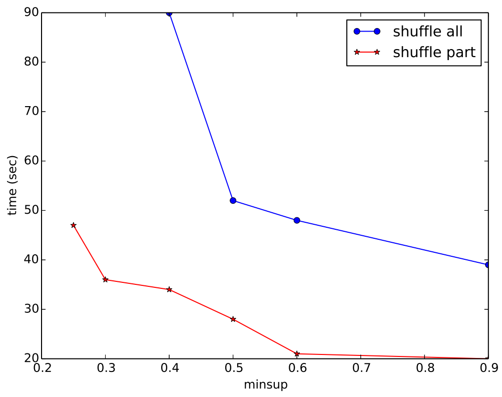


(b) T2t10d10k datasets

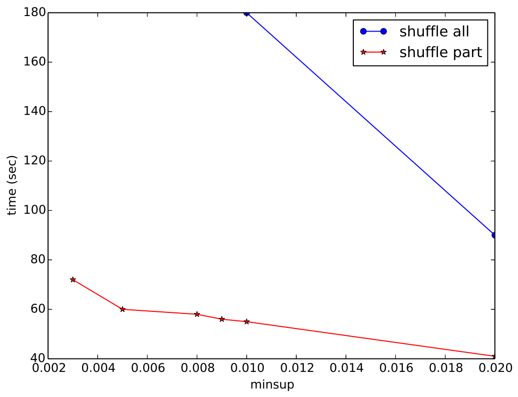
Figure 5.9: Use remapping to reduce the communication cost

### 5.4.3 Dynamically Balance the Workload

This experiment in Figure 5.10 shows how the load balance impact the running time of the SVT algorithm. It is better to shuffle the itemsets to their corresponding equivalent class once it is clear which equivalent class it belongs to rather than calculate the equivalent class and shuffle this equivalent class entirely. This is most important step to keep load balanced. From the mushroom datasets on the left, it almost takes double time to switch the entire equivalent classes rather than just switch part of them. On the t25i10d10k synthetic datasets, shuffle all equivalent classes has a linear relation with running time, which is very inefficient.



(a) Mushroom datasets



(b) T2t10d10k datasets

Figure 5.10: Dynamically balance workload

## 5.5 Summary

Experimental results in this chapter show that our proposed SVT algorithm provides high speedup, requires short runtime, and takes less memory space when mining frequent itemsets vertically from big data.

# Chapter 6

# Conculsions and Future Work

## 6.1 Conculsions

Enormous quantities of data are generated and collected daily about every aspect of society. The knowledge or useful information is often hidden inside and there are also a lot of noise in data which prohibits us from understanding the useful information. Data mining is a procedure for finding non-trivial, previously unknown and potentially useful information in data, is a useful tool to discover hidden knowledge inside data. As one of the commonly used data mining tasks, frequent itemset mining (FIM) finds itemsets appears frequently from a transaction dataset.

The 21st century is known as Information Age. The increasing size of modern applications that are producing huge amounts of data everyday leads to a new challenge to data mining known as the "Big Data" problem. It has been shown that classical data mining algorithms developed over the past decades may not suitable for the modern size of applications, since the data cannot be held in the main memory. Existing parallel and distributed data mining approaches that have been adopted from previous single machine algorithms ignore the variety of datasets. A dataset can either be sparse or dense, or some parts of it can be sparse and other is dense.

In this MSc thesis, I have investigated the following questions:

1. Can we design an algorithm that produces a complete answer (i.e., finds all frequent itemsets)?
2. Can we design an algorithm that can be used in different densities of datasets?
3. Can we design an algorithm that is faster than existing algorithms?
4. Can we design an algorithm which can distribute workloads equally?
5. Can we design an algorithm with less communication cost than existing algorithms?

At the end, I designed and implemented a new algorithm call scalable vertical mining (SVT) to discover frequent patterns in a big data environment. The purposed parallel algorithm satisfies following qualities:

1. SVT finds a complete list of all frequent patterns and their transactions.
2. SVT is scalable to different densities of datasets because of its hybrid nature, which enable the algorithm to switch from capturing transaction IDs to capturing only differences in transaction IDs between the itemset and its supersets based on the data densities.
3. SVT is efficient because it uses the vertical format for parallel computing and hybrid vertical mining

4. SVT creates a balanced workload among workers.

5. SVT also takes a low communication cost.

These qualities (completeness, scalability, efficiency, balance) are critical both for a data mining algorithm and a parallel frequent itemset mining algorithm.

## 6.2 Future Work

During the experimental stage, I noticed that the SVT algorithm may not be extremely efficient under some scenarios. One reason is that the data structure is not fully optimized. To reduce the memory consumption, I need to compress the redundant data in memory. One way is to use bitmap.

Another issue is data may not be evenly distributed among workers. Because of the distribution policy in the Spark platform, only minimum number of workers will join current computation when the datasets fit into the workers' memory. A interesting future direction is to maximize the utility and minimize the current load to achieve a better performance.

# Bibliography

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[ATJL09] Chowdhury Farhan Ahmed, Syed Khairuzzaman Tanbeer, Byeong-Soo Jeong, and Young-Koo Lee. Efficient tree structures for high utility pattern mining in incremental databases. *IEEE Transactions on Knowledge and Data Engineering*, 21(12):1708–1721, December 2009.

[BG09] Mario Boley and Henrik Grosskreutz. Approximating the number of frequent sets in dense data. *Knowledge and Information Systems*, 21(1):65–89, 2009.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 255–264, New York, NY, USA, 1997. ACM.

[Cor13] Intel Corp. What happens in an internet minute?, 2013. Online; accessed 20-January-2016.

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

[FVGG$^+$14] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S. Tseng. SPMF: A Java open-source pattern mining library. *Journal of Machine Learning Research*, 15(1):3389–3393, January 2014.

[HPK11] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 1–12, New York, NY, USA, 2000. ACM.

[LJ15] Carson Kai-Sang Leung and Fan Jiang. Big data analytics of social networks for the discovery of "following" patterns. In Sanjay Madria and Takahiro Hara, editors, *Big Data Analytics and Knowledge Discovery*, volume 9263 of *Lecture Notes in Computer Science*, pages 123–135. Springer International Publishing, 2015.

[LMB08] Carson Kai-Sang Leung, Mark Anthony F. Mateo, and Dale A. Brajczuk. *A Tree-Based Approach for Frequent Pattern Mining from Uncertain Data*, pages 653–661. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, January 2003.

[LWZ$^+$08] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: Parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 107–114, New York, NY, USA, 2008. ACM.

[Mos13] Lichman Moshe. UCI machine learning repository, 2013.

[PCY95] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 175–186, New York, NY, USA, 1995. ACM.

[PF91] Gregory Piateski and William Frawley. *Knowledge Discovery in Databases.* MIT Press, Cambridge, MA, USA, 1991.

[QGYH14] Hongjian Qiu, Rong Gu, Chunfeng Yuan, and Yihua Huang. Yafim: A parallel frequent itemset mining algorithm with spark. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW '14, pages 1664–1671, Washington, DC, USA, 2014. IEEE Computer Society.

[Rod85] David P. Rodgers. Improvements in multiprocessor system design. *SIGARCH Computer Architecture News*, 13(3):225–231, June 1985.

[SEB13] Moens Sandy, Aksehirli Emin, and Goethals Bart. Frequent itemset mining for big data. In *Proceedings of the 2013 IEEE International Conference on Big Data*, pages 111–118, Oct 2013.

[SHS$^+$00] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. VIPER: A vertical approach to mining association rules. In *ACM SIGMOD International Conference on Management of Data, SIGMOD '00)*, 2000.

[Whi09] Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 1st edition, 2009.

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

[ZCD$^+$12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the Ninth USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZCF$^+$10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the Second USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–17, Berkeley, CA, USA, 2010. USENIX Association.

[ZG03] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 326–335, New York, NY, USA, 2003. ACM.

[ZJT13] Zhigang Zhang, Genlin Ji, and Mengmeng Tang. Mreclat: An algorithm for parallel mining frequent itemsets. In *Proceedings of the 2013 International Conference on Advanced Cloud and Big Data*, CBD '13, pages 177–180, Washington, DC, USA, 2013. IEEE Computer Society.

[ZZC$^+$10] Le Zhou, Zhiyong Zhong, Jin Chang, Junjie Li, Joshua Zhexue Huang, and Shengzhong Feng. Balanced parallel FP-growth with MapReduce. In *Proceedings of the 2010 IEEE Youth Conference on Information Computing and Telecommunications, YC-ICT '10*, pages 243–246, November 2010.