Fast and Scalable MapReduce-Based Vertical Mining

by

Jialiang Yu

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

July 2018

Thesis advisor

Dr. Carson K. Leung

Author

Jialiang Yu

Fast and Scalable

MapReduce-Based Vertical Mining

Abstract

Mining uncertain data is challenging because uncertainty is usually represented as real numbers which are infinite (cf. representing finite occurrence counts when mining precise data). This means that they are not easy to store in a data structure. Although there exist some data mining algorithms for handling uncertain data, these algorithms become inefficient when the size of data becomes so big. Vertical data mining algorithms have advantages in that they run fast and require low memory space. Hence, for my M.Sc. thesis, I propose two vertical mining algorithms that mine big uncertain data. Analytical and experimental evaluation results show that, between these two MapReduce-based vertical mining algorithms, MR-UV-Eclat is fast and scalable.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my deepest gratitude to my research supervisor Dr. Carson K. Leung. This thesis would not have been possible without his encouragement and academic support. Back when I was an undergraduate student, I acquired my database and data mining knowledge from Dr. Leung. His enthusiasm and inspiration made me interest in the research area of data mining. After I finished my undergraduate degree and started working, Dr. Leung gave me an opportunity to come back to conduct research on this M.Sc. thesis. In addition, I also acknowledge the Database and Data Mining Laboratory, Department of Computer Science, and University of Manitoba for providing me a comfortable research environment, plenty of research equipment, and rapid technical support.

Moreover, I would like to thank my M.Sc. thesis examination committee members, Dr. Carl N.M. Ho (ECE) and Dr. Yang Wang (Computer Science), for their precious time to read and examine my thesis. I would also like to thank my M.Sc. thesis defense chair, Dr. James E. Young, who also served as the Acting Head of Computer Science on the week when my defense was held (i.e., the week of July 12th, 2018).

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

Jialiang Yu

B.C.Sc. (Hons.), The University of Manitoba, Canada, 2009

The University of Manitoba
July 2018

This thesis is dedicated to my parents and my special friend Yuwa.

Chapter 1

Introduction

As technologies advance, we are living in a world where many things are managed by machines. As computers and digital devices generate increasing data, these datasets are much easier to acquire than ever before. As suggested by Fayyad et al. [FPS96], "across a wide variety of fields, data are being collected and accumulated at a dramatic pace. There is an urgent need for a new generation of computational theories and tools to assist humans in extracting useful information (knowledge) from the rapidly growing volumes of digital data." Data mining is a non-trivial extraction of previously unknown and potential useful relationships from data. It provides many means to find patterns and knowledge from data. There are many data mining methods. For instance, social mining is very widely [BCJ+17, CL17, LJPC18]. Data mining is also being applied in the Internet of Things (IoT) data [FLS+18, SYZ+18]. Among all different data mining approaches, association rule mining is a very important data mining task. Liu et al. [LHM98] defined that association rule mining finds all rules in the database that satisfy some user-specified minimum support and

minimum confidence constraints.

At the very early stage, most associate mining algorithms only mine precise data. Precise data are data that users are certain about the existence in transactions in databases. If items are presented in current transactions, their existences are in a form of integers. One of the earliest associate mining algorithms is Apriori algorithm [AIS93] which was proposed by Agrawal and Srikant in 1994. This algorithm relies on generate and test approaches. It first generates candidates and then verifies their frequencies by doing database scans. Users are required to define a minimum support threshold which is the least frequency. This number is being compared with support values of all candidates and finally only itemsets which their frequencies are larger than minimum threshold are kept in the frequent data sets. This process is repeated until no frequent itemsets can be generated.

Two large drawbacks of the Apriori algorithm are the very large candidate generations and the many database scans. To overcome these drawbacks, Han et al. [HPY00] proposed a tree-based algorithm called FP-growth. This approach requires two database scans. During the first database scan, it finds all frequent singletons. At the second database scan, it generates a global FP-tree. The algorithm then recursively builds up all different projection trees. Eventually, all frequent itemsets are generated from these projection trees. This algorithm is fast because it reduces database scans to two times and it can greatly reduce candidate generations as it only generates candidates from very small subsets. However, this algorithm still has a risk that data might not fit into the main memory and also this algorithm becomes less efficient when there are so many leaves in trees. Thus, H-mine [PHL$^+$01] and some

vertical mining approaches [SHS+00, ZPOL97] are proposed to adjust some of these issues.

So far, all those have been mentioned above relate to precise data. Still, there exists uncertain data in many real life applications. As suggested by Sun et al. [SCCC10], "data managed in many emerging applications are often uncertain." Uncertain data are referred to as the existence of items in transactions that are not certain. This means the form of these data is represented as probabilities of existences of items. Two challenges of uncertain data mining are the data size is larger compared with precise data and data cannot easily fit into a data structure. Recall that precise data is data that users are certain about the existence in transactions. If an item is presented in a transaction, it shows as singular in a transaction. Otherwise, it will not be present in the transaction. Thus, the size of data can be very small. On the contrary, uncertain data are presented as the likelihood of existence in transactions. These datasets are decimal numbers and so can be indefinite. Second, the likely hoods of items can be different in transactions. This means even the same items present in two or more transactions may have very different uncertain values. Hence, these data cannot be packed into a small data structure. For example, if we want to put them into a FP-tree, it may create many leaves. Great amount of precise data mining algorithms have been extended to apply to mining uncertain data. Many algorithms like CAP [NLHP98], U-Apriori [CKH07], UF-growth [LMB08] and SOP-tree [KD13] are proposed to handle different kind of data, they still lack efficiencies when data size becomes big.

Data are easier to acquire than ever before because more and more Internet tech-

nologies are being used in our life. "These data arise from the Internet (searches, social networks, blogs, pictures), smartphones, scientific studies (genomics, brain imaging, epidemiology, environmental research), businesses (customer records, transactions, financial indicators), governments (population, healthcare, weather, automatic sensors) and other sources" [Ros14]. Zikopoulos et al. [ZED+12] defines three characters of big data. They are Volume, Velocity and Variety. That means big data have high volume, fast generation speed and so many resources. For example, 294 billion emails sent every day and over 1 billion Google searches every day[1]. We are living in a world with a flood of data but still we lack the knowledge and useful data relationships that can be potentially refined from these big data. Hence, we are interested in utilizing data mining technologies to find useful knowledge from data. However, many of hardware architectures and data management techniques of today are not sufficient to handle big data. As mentioned, big data have very high volume, fast generation speed and many different data resources. Much of the existing hardware has very limited hard drive spaces and insufficient memory. Furthermore, data can be only centralized in one physical location. Thus, servers are quickly depleted of hard disk space and have very slow data retrieval. For new hardware architectures to handle big data, we require decentralizing the databases access and enable concurrent processing on different servers. Limitations also exist in traditional data management techniques. Many existing data technologies only allow a single process at a time in only one physical machine. For example, for data retrieval, these technologies only allow one search at a time. As a matter of fact, we should allow multiple searches in different machines in order to speed up the overall efficiency. Cloud computing is an

[1] `http://www.ibmbigdatahub.com/gallery/quick-facts-and-stats-big-data`

ideal architecture to use and also data management techniques should enable decentralized database allocations and concurrent accesses. Currently, Hadoop, Spark and Storm are among the three most popular big data tools. Still, there are not many data mining algorithms adopted to big data technologies.

1.1 Thesis Statement

Motivated by the problems and solutions previously mentioned, my thesis statement is as follows: I proposed a vertical mining approach that performs the following tasks;

- Retrieve frequent patterns from uncertain data
- Conduct association rule mining in big data
- Provide a scalable and fast vertical mining approach
- Save memory and speed up execution time as well as adapt spark big data techniques

In this thesis, we adapt Spark technologies and make an improved version of Map-Reduce approach of Apriori called MR-U-Apriori-impr. This algorithm improved the overall efficiencies compared with the original Map-Reduce version. We then describe our proposed mining approach called MR-UV-Eclat. This algorithm adopts the approach of U-Eclat using Spark technologies. We also describe the Spark version of U-VIPER called MR-U-VIPER. Performances compared on MR-U-Apriori-org,

MR-U-Apriori-impr, MR-U-VIPER and MR-UV-Eclat are conducted. Experimental results in Chapter 5 are displayed.

1.2 Problem Description and Thesis Contribution

Uncertain data are usually in the format of floating points and can have different values even though they are in the same itemsets in different transactions. Thus, they are not easy to fit into a data structure. For many existing mining algorithms that mine uncertain data, they consume much memory and have very long execution time. To make the case worse, when data become so big, these algorithms may have out of memory issues. Based these issues, some logical questions to ask are:

1. Are there any more scalable data structures that can efficiently store uncertain data?

2. Can we provide mining algorithms that can save memory?

3. Can we provide mining algorithms that efficiently handle big uncertain data?

4. Can we provide mining algorithms that improve overall performances when dealing with big data

In response to these above questions, we explore different approaches. We find that vertical mining models can fit the large data structure into main memory. Compared with other mining models, vertical mining models can save much more memory. At the same time, many existing Map-Reduce association-mining algorithms utilize

Hadoop. Hadoop applications encounter so much I/O cost that leads to low performances. Spark can boost up overall performance by putting some major processes in memory. In this case, in order for a Spark algorithm to perform well, it is always a good idea for the Spark algorithm to have low memory usage consumptions.

Hence, we propose two Map-Reduce vertical mining approaches using Spark techniques. Our key contributions are:

1. Proposal of Map-Reduce version of Eclat algorithm on mining uncertain data
2. Proposal of Map-Reduce version of VIPER algorithm on mining uncertain data
3. Both of these algorithms utilize Spark techniques
4. Analytical and experimental evaluation of these algorithms

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 provides background and related work. When talking about background, the first association mining algorithm, which is Apriori algorithm, is demonstrated in detail. We then talk about two vertical mining algorithms. They are VIPER and Eclat. These three algorithms are on precise data. Specifically, we explain why vertical mining algorithms have great success on mining precise data. We then describe uncertain data and three algorithms on mining uncertain data. They are U-Apriori, U-VIPER and U-Eclat. Next, we outline characteristics on big data. From that, we explain some challenges on mining big data and describe existing technologies on mining big data.

Chapter 3 is the main chapter for describing our main contributions for this thesis. MR-UV-Eclat are adopted by UV-Eclat. We provides a detail example to explain the algorithm. On describing the algorithm MR-UV-Eclat, we also introduce a data structure called projection tree. This is used in MR-UV-Eclat. We also explain the MR-UV-Eclat by providing a detailed example.

To make vertical mining complete, we also adopt Spark technologies in U-VIPER. This called MR-U-VIPER. In Chapter 4, we first explain the MapReduce behaviors in MR-U-VIPER and the memory saving techniques in MR-U-VIPER. After that, details steps on MR-U-VIPER are provided. At the end of this chapter, we provide a detail example for this algorithm.

In Chapter 5, we compare our two algorithms MR-UV-Eclat and MR-U-VIPER with two varianets of MapReduce version of Apriori (namely, MR-Apriori-org and MR-Apriori-impr). Analytical evaluation and experimental evaluation of our proposed algorithm will be provided. All these evaluations are on comparisons of MR-Apriori-org, MR-Apriori-impr, MR-U-VIPER and MR-UV-Eclat.

Finally, in Chapter 6, we present conclusions of this thesis. Moreover, we also suggest some future research directions.

Chapter 2

Background and Related Work

In this chapter, we provide background and related work to understand the thesis. We start with definitions and tasks to accomplish in association mining. This leads to presentations on three mining algorithms on precise data. They are Apriori algorithm [AIS93] and two vertical mining algorithms which are VIPER [SHS+00] and Eclat [ZPOL97]. By comparing these three algorithms, we conclude reasons why vertical mining algorithms have advantages over other algorithms on mining precise data. Next, we demonstrate algorithms on uncertain data. Specifically, we continue to explain vertical mining on uncertain data. Hence, U-VIPER [LTBZ12] and UV-Eclat [BCL12] are two main algorithms described. We then detail explain characters on big data, its related technologies and data mining approaches on big data.

2.1 Association Mining from Precise Data

Many people realize that identifying data relationships and data trends are important for decision-making. It is also observed that many organizations store massive data in database systems but these systems lack functionalities of searching for recurrently occurring patterns and data trends. For instance, supermarkets store large data on daily transactions. However, these data do not necessarily consist of items bought together at the same point of time so that decision makers may miss valuable data knowledge. As mentioned, association mining is an important task in data mining and can help decision makers to identify data relationships. For instance, by using association rule mining, Agrawal et al. [AIS93] indicates that "90% of transactions that purchase bread and butter also purchase milk". Klemettinen et al. [KMR+94] state that "The propose of data mining is to facilitate understanding large amount of data by discovering interesting regularities of exceptions" Agrawal et al. [AIS93] gave the definition of association rule mining. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Associated with each transaction is a unique transaction identifier (TID). We say that a transaction T contains X (which a set of some items in I) if $X \subseteq T$. An association rule is an implication of the form $X \rightarrow Y$, where $X \subset I, Y \subset I$, and $X \cap Y = \emptyset$. At the very beginning, data association mining deals with are precise data. The present of items in transactions is either 1 or 0. This indicates that the existence of items in transactions is either 100% or 0%. Three algorithms are demonstrated in this section. They are Apriori [AIS93], VIPER [SHS+00] and Eclat [ZPOL97].

Table 2.1: Sample database 1

TID	**Content**
t_1	{a, b, c, d, e}
t_2	{b, c, e}
t_3	{a, b, d, e}
t_4	{a, b, c, e}
t_5	{a, b, c, d}
t_6	{b, c, d}

2.1.1 Horizontal Mining with the Apriori Algorithm

In 1994, Agrawal et al. [AS94] proposed the first association rule mining algorithm. In this approach, it repeatedly generates next level candidates from the current level of frequent itemsets and then verifies these candidates from data. This step is repeated until there is no new candidate generated.

Pattern	Sup Value
{a}	4
{b}	6
{c}	5
{d}	4
{e}	4

(a) C_1

Pattern	Sup Value
{a}	4
{b}	6
{c}	5
{d}	4
{e}	4

(b) L_1

Pattern	Sup Value
{a, b}	4
{a, c}	3
{a, d}	3
{a, e}	3
{b, c}	5
{b, d}	4
{b, e}	4
{c, d}	3
{c, e}	3
{d, e}	2

(c) C_2

Pattern	Sup Value
{a, b}	4
{a, c}	3
{a, d}	3
{a, e}	3
{b, c}	5
{b, d}	4
{b, e}	5
{c, d}	3
{c, e}	4

(d) L_2

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

(e) C_3

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

(f) L_3

Figure 2.1: Apriori Algorithm Demonstration

Consider the sample database shown in Table 2.1 and set the minimum support

threshold to be 3. It first scans the database once to find all frequent singletons. In Figure 2.1(a), the first level candidate is called C_1. If support values of candidates are less than the minimum support threshold, these items are removed. After the first database scan, frequent singletons are verified. In the example, after removing the infrequent candidates from C_1, L_1 is generated. Then C_2 is generated by doing self-combination of L_1. The algorithm then conducts the other database scan to calculate the support value of each candidate in C_2. After removing infrequent candidates from C_2, we get L_2. This process is repeated to get C_3 and L_3. Note that when trying generating C_4, we never consider any candidates. From L_3, we should have generated $\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}$ and $\{b, c, d, e\}$. Nevertheless, we can eliminate them before doing database scan as we can find subset of these candidates are infrequent. For example, $\{a, c, d\}$ is the subset of $\{a, b, c, d\}$ and $\{a, c, d\}$ is infrequent. This step is called pruning.

Apriori algorithm is a well-known algorithm for association rule mining. However, this algorithm suffers from two drawbacks. The first one is it generates many candidates. This drawback can lead to very slow elapsed time and the system can be out of memory easily. The second one is that it requires many database scans.

2.1.2 Vertical Mining with the VIPER Algorithm

Many existing association mining algorithms view data "horizontally". Take super market data as an example, each record represents a transaction and each transaction contains purchased items. Many approaches can apply these data directly to perform mining. However, these data can also be converted to vertical data. In the example

of super market data, we can convert them from transaction view basic to itemset view basic. The conversion of vertical data is demonstrated on Figure 2.2. Two strengths of vertical mining algorithms are that (1) they only do database scan once; and (2) they consume much less memory. Two algorithms are demonstrated in this section. They are VIPER and Eclat.

TID	Content
t1	{a, c}
t2	{b}
t3	{b, c}
t4	{a, c, c}
t5	{a, b, c}

(a) Horizontal Data

TID	Items				
	a	b	c	d	e
t1	1	0	1	0	0
t2	0	1	0	0	0
t3	0	1	1	0	0
t4	1	0	1	0	1
t5	1	1	1	0	0

(b) Vertical Data

Figure 2.2: Vertical Data Conversion

Shenoy et al. [SHS+00] proposed the VIPER algorithm. This algorithm performs a database scan once. In precise data, the existence of an item in a transaction is either 1 or 0. Hence, we can only convert all vertical data to be a bit-vector. In the VIPER algorithm, each bit-vector of an item has the same size equal to the total number of transactions. The value on each transaction for an item is either 1 or 0. 1 means the item exists in the transaction and 0 means the item does not exist in the corresponding transaction. When trying to calculate the support value on a particular pattern, the algorithm sums up the all values in the bit-vector to form the support value. In order to form the next level of candidates, the algorithm intersects two bit-vectors to form a new bit-vector. Note that in this case, we do not need to go back to

the original database to do such operation. We still use the candidate generate and test approach to get all frequent itemsets. The process will be terminated when no new candidates can be generated.

Example 2.1.1. In this example, we still use the sample database in Table 2.1. The minimum support is still 3. This has been demonstrated in Figure 2.3. The algorithm first converts the database to vertical data. All data become bit-vectors. All support values on all singletons are calculated by summing up all 1s in each bit vector. Only candidates with a support value more than the minimum support threshold are kept for forming the next level candidates. Note that all vectors of all frequent singletons are kept in the memory to form other level of candidates. In level 2, each bit-vector joins with other bit-vector to form level 2 bit-vectors. For example, bit-vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ intersect by doing [101110] AND [111111] = [101110], so we get $\overrightarrow{ab}$ = [101110]. After forming frequent level 2 bit-vectors, we utilize these bit vectors to join with frequent singleton bit-vectors to form level 3 bit-vectors. For instance, bit-vectors $\overrightarrow{ab}$ and $\overrightarrow{c}$ intersect by doing [101110] AND [110111] = [100110], so we get $\overrightarrow{abc}$ = [100110]. Note that we prune away acd and cde as they are infrequent. Finally, no bit-vector can be generated in level 4. Hence, the process stops.

	a	b	c	d	e
t1	1	1	1	1	1
t2	0	1	1	0	1
t3	1	1	0	1	1
t4	1	1	1	0	1
t5	1	1	1	1	0
t6	0	1	1	1	0
Sup Value	4	6	5	4	4

(a) Convert to Vertical Data

	ab	ac	ad	ae	bc	bd	be	cd	ce	**de**
t1	1	1	1	1	1	1	1	1	1	**1**
t2	0	0	0	0	1	0	1	0	1	**0**
t3	1	0	1	1	0	1	1	0	0	**1**
t4	1	1	0	1	1	0	1	0	1	**0**
t5	1	1	1	0	1	1	0	1	0	**0**
t6	0	0	0	0	1	1	0	1	0	**0**
Sup Value	4	3	3	3	5	4	4	3	3	**2**

(b) Candidate Level 2

	abc	abd	abe	**acd**	**ace**	bcd	bce
t1	1	1	1	**1**	**1**	1	1
t2	0	0	0	**0**	**0**	0	1
t3	0	1	1	**0**	**0**	0	0
t4	1	0	1	**0**	**1**	0	1
t5	1	1	0	**1**	**0**	1	0
t6	0	0	0	**0**	**0**	1	0
Sup Value	3	3	3	**2**	**2**	3	3

(c) Candidate Level 3

Figure 2.3: VIPER Algorithm

VIPER algorithm has all data in bit-vectors. This greatly reduce memory consumption. Also, this algorithm does not need to go back to the original database to verify the candidates. Instead, it uses the simple operation bit-wise operation AND

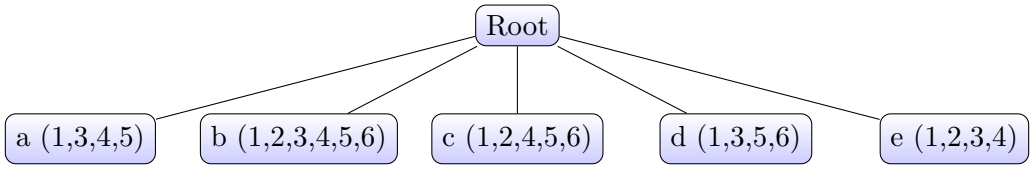
of two bit-vectors to form the new bit-vector. Hence, the execution time is very fast. Still, VIPER algorithm cannot handle uncertain data. Hence, we will demonstrate the extension of this algorithm U-VIPER.

2.1.3 Vertical Mining with the Eclat Algorithm

Zaki et al. [ZPOL97] proposed Eclat algorithm. Unlike the data format in VIPER algorithm, the vertical data used in Eclat algorithm only keeps the transaction data (transaction ID) that belongs to the item in the vector of the item. Hence, vectors store lists of integers. On calculating frequencies of items, this algorithm counts the number of transactions. On forming the next level of vectors of candidates, it does the intersection of two vectors.

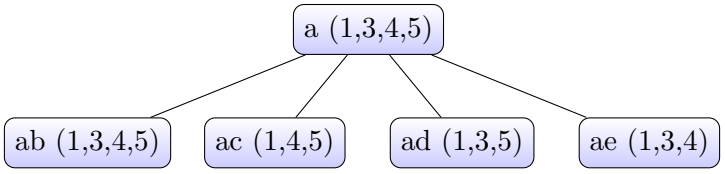
Example 2.1.2. We still use database Table 2.1 in this case and set minimum support threshold value as 3. Eclat algorithm converts original data into vertical data that only store exist transaction IDs in each vector. The algorithm forms new itemsets by intersecting vectors of current level with vectors of singletons. For example, we can obtain the vector of $\overrightarrow{abc}$ by intersecting the vector $\overrightarrow{ab}$ with the vector $\overrightarrow{c}$. From figure 2.6, we can observe that $\overrightarrow{abc} = \overrightarrow{ab} \cap \overrightarrow{c} = \{1,3,4,5\} \cap \{1,2,4,5,6\} = \{1,4,5\}$. In Figure 2.4, Figure 2.5, Figure 2.6, they demonstrate how each level of itemsets form. The formation of next level of itemsets is by searching prefix and suffix. In this database, we know that the frequent singletons are $\{a, b, c, d, e\}$. For current itemset ab, in order to form the next level of itemsets which have prefix of ab, the suffix dataset are $\{c, d, e\}$. Hence, we can form level 3 candidates of $\{abc, abd, abe\}$. The algorithm goes through each itemset in the current level and looks for their possible

suffixes. New candidates are formed by intersecting prefix and suffix.

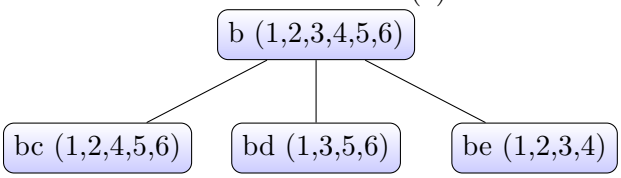


(a) Candidate level 1

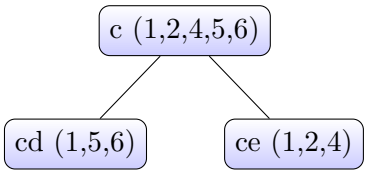
Figure 2.4: (Eclat) Convert to Vertical Data



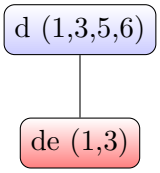
(a) Candidate level 2 for a



(b) Candidate level 2 for b

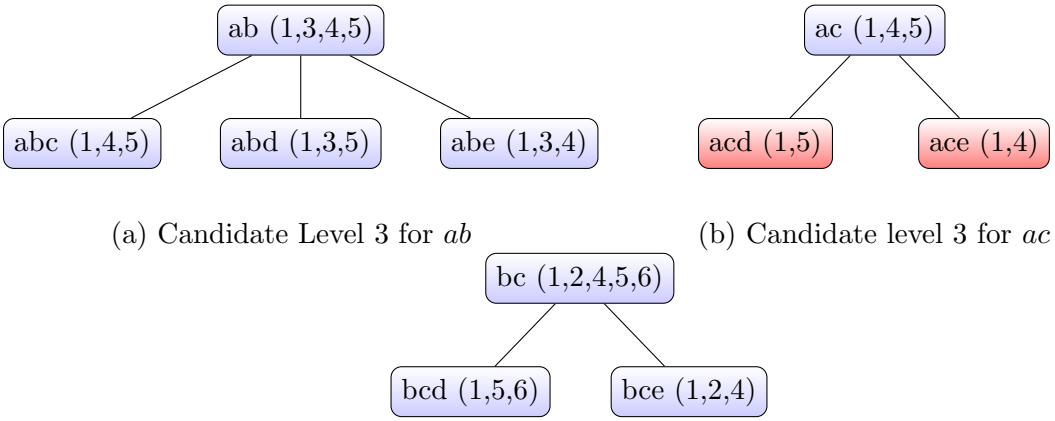


(c) Candidate level 2 for c



(d) Candidate level 2 for d

Figure 2.5: (Eclat) Candidate level 2



(a) Candidate Level 3 for *ab*

(b) Candidate level 3 for *ac*

(d) Candidate level 3 for *bc*

Figure 2.6: (Eclat) Candidate level 3

Eclat algorithm only contains transaction information in each vector. This can save memory especially for the sparse database. Still, Eclat cannot handle uncertain data. Hence, UV-Eclat is introduced.

2.2 Association Rule Mining from Uncertain Data

There exists many uncertain databases (UDB). For instance, Budhia et al. [BCL12] stated "uncertainty in the UDB can be caused by our limited perception or understanding of reality, inherited measurement inaccuracies, inappropriate sampling frequencies of sensors, wireless transmission errors, and network latencies". Chui et al. [CKH07] provided problem definitions on uncertain data mining. Given a possible world W_i and an itemset X, let us define $P(W_i)$ be the probability of world P_i and S(X,W_i) be the support count of X in world W_i. Furthermore, we use $T_{i,j}$ to denote the set of items that the j-th transaction, i.e., t_j, contains in the world W_i, then

$P(W_i)$ and the expected support $expSup(\mathrm{X})$ of X are given by the following formula:

$$expSup(X) = \sum_{i=1}^{|W|} P(W_i) \times S(X, W_i) \tag{2.1}$$

where W is the set of possible worlds derived from an uncertain dataset UDB, and

$$P(W_i) = \prod_{j=1}^{d} \Big(\prod_{x \in T_{i,j}} P_{t_j} \cdot \prod_{y \notin T_{i,j}} (1 - P_{t_j}(y)) \Big). \tag{2.2}$$

Computing $expSup(\mathrm{X})$ according to Equation (2.1) requires enumerating all possible worlds and finding the support count of X in each world. This is computationally infeasible since there are 2^m possible worlds where m is the total number of items that occur in all transactions of D. Fortunately, if items within X are independent, then

$$expSup(X) = \sum_{j=1}^{|D|} \prod_{x \in X} P_{t_j}(x) \tag{2.3}$$

Thus, $expSup(\mathrm{X})$ can be computed by a single scan through the dataset UDB. The above model determines that the calculation processing of association rule mining in uncertain data is different than mining precise data.

Uncertain data are represented as decimal data, this nature indicates that the size of database can be very large. These proposed algorithms may encounter a problem that all data cannot fit into main memory. Especially, when the number of transactions becomes large, we have to store data in hard disks and use very different technologies to handle these data. Chui et al. [CK08] proposed the U-Apriori algorithm that relies on the candidate generate-and-test paradigm for mining. Leung

et al. [LMB08] proposed a tree-base algorithm, called UFP-growth, which uses a tree structure called UF-tree to mine uncertain data. In the next two section, two algorithms are demonstrated. They are U-VIPER and UV-Eclat.

2.2.1 U-VIPER Algorithm

Leung et al. [LTBZ12] proposed the U-VIPER algorithm. This approach first converts the database to vertical data. Take a small database as an example. Suppose the input data is in the format {TID: Item:value ...}. Suppose we have three transactions:

TID	content
t_1	a:0.9, b:0.8, c:0.9
t_2	a:0.1
t_3	b:0.2

Figure 2.7: Sample Database

We can use the rule

$$\vec{x}[i] = \begin{cases} P(x,t) & \text{if } x \in t \\ 0 & \text{if } x \notin t \end{cases} \tag{2.4}$$

to transfer the database into

TID	a	b	c
t_1	0.9	0.8	0.9
t_2	0.1	0	0
t_3	0	0.2	0

Figure 2.8: Vertical Data (U-VIPER)

Let minsup = 1. We compute the frequency of each item by the following equation:

$$expSup(x) = || \vec{x} || = \sum_i \vec{x}[i] = \sum_i P(x, t_i) \tag{2.5}$$

and obtain expSup(a) = 0.9+0.1+0=1, expSup(b) = 0.8+0+0.2=1, expSup(c) = 0.9+0+0 = 0.9 < 1 (pruned). Then, we generate 2-candidate itemsets by all frequent items to obtain the candidate ab. We can then $\vec{a} \cap \vec{b}$ to get $\overrightarrow{ab}$, the value of each transaction can be calculated by

$$expSup(xy, t_i) = expSup(x, t_i) \times expSup(y, t_i) \tag{2.6}$$

Consequently, we obtain (ab: 0.72, 0, 0). This process is continued until no more candidates can be generated. See Algorithm 1 (U-VIPER).

Algorithm 1 U-VIPER

1: Transform the probabilistic dataset of the form $\{\langle\ t_i, \{\ \text{x:P(x,}t_i\text{)}\ \rangle\ |\text{x} \in t_i\}\}$ into vertical format with $\overrightarrow{x}$ such that the i-th element in $\overrightarrow{x}$ is either P(x,t_i) when x $\in t_i$ or 0 when x $\notin t_i$.
2: For each domain item x, compute expSup(x) as a L_1-norm.
3: L_1 = { ⟨x:expSup(x)⟩|expSup(x) ≥ minsup}; k = 2.
4: C_k = {⟨xy:expSup(xy)⟩ | expSup(xy) can be computed as a dot product of $\overrightarrow{x}$ and $\overrightarrow{y}$ }.
5: **while** $C_k \neq \varnothing$ **do**
6: L_k = { ⟨α:expSup(α)⟩ | ⟨α:expSup(α)⟩ ∈ C_k and expSup (α) ≥ minsup}; k = k + 1.
7: C_k = {⟨βy:expSup(βy)⟩ | ⟨β:expSup(β) ∈ $L_{(k-1)}$, ⟨y:expSup(y)⟩ ∈ C_1, and expSup(βy) can be computed as a dot product of $\overrightarrow{\beta}$ and $\overrightarrow{y}$}.
8: **end while**
9: Return $\cup_i L_i$.

The advantage of this algorithm is that all vertical vectors are small in size. The operation is very simple as it only requires intersections of vectors to acquire new vectors. However, this algorithm is still needed to adopt to big data technologies as this algorithm can be only executed in a single machine.

2.2.2 UV-Eclat Algorithm

Budhia et al.[BCL12] proposed UV-Eclat algorithm. The vector representation is different than the one in U-VIPER. We only keep IDs for transactions that contain the item. The transformation of vertical data is shown below:

a	b	c
t_1:0.9 t_2:0.1	t_2:0.8 t_3:0.2	t_1:0.9

Figure 2.9: Vertical Data(UV-Eclat)

Let minsup = 1. We compute frequency of each item by the following equation:

$$expSup(x) =|| \vec{x} ||= \sum_i \vec{x}[i] = \sum_i P(x, t_i) \tag{2.7}$$

and obtain expSup(a) = 0.9+0.1=1, expSup(b) = 0.8+0.2=1, expSup(c) = 0.9 < 1 (pruned). Then, we generate 2-candidate itemsets by all frequent items to get the candidate ab. The formation of new vector is different than U-VIPER, it is formed by following rules: atidset(αxy)={$\langle t_i$: expSup(αx,t_i) × P(y,t_i)$\rangle$ | $\langle t_i$:expSup(αx,t_i)$\rangle$ ∈ atidset(αx) ∧ $\langle t_i$:P(y,t_i)$\rangle$ ∈atidset(y)}.we can get (ab: 0.72, 0, 0), this process continues until no more candidates can be generated. See Algorithm 2 (UV-Eclat).

UV-Eclat only keeps transaction data to which that item belongs. Hence, it keeps a small amount of data. However, this algorithm still needs to adopt big data technologies to mine big data.

Algorithm 2 UV-Eclat

1: Transform the probabilistic dataset of the form $\{\langle\ t_i, \{\ \text{x:P(x,}t_i\text{)}\ \rangle\ |\text{x} \in t_i\}\}$ into vertical format with $\overrightarrow{x}$ such that the i-th element in $\overrightarrow{x}$ in P(x,t_i)
2: For each domain item x, compute expSup(x) as a L_1-norm.
3: $L_1 = \{\ \langle\text{x:expSup(x)}\rangle|\text{expSup(x)} \geq \text{minsup}\}$; k = 2.
4: $C_k = \{\langle\text{xy:expSup(xy)}\rangle$ | expSup(xy) can be computed as a dot product of $\overrightarrow{x}$ and $\overrightarrow{y}$ $\}$.
5: **while** $C_k \neq \varnothing$ **do**
6: $L_k = \{\ \langle\alpha\text{:expSup}(\alpha)\rangle \mid \langle\alpha\text{:expSup}(\alpha)\rangle \in C_k$ and expSup $(\alpha) \geq$ minsup$\}$; k = k + 1.
7: $C_k = \{\langle\beta\text{y:expSup}(\beta\text{y})\rangle \mid \langle\beta\text{:expSup}(\beta) \in L_{(k-1)}, \langle\text{y:expSup(y)}\rangle \in C_1$, and expSup($\beta$y) can be computed as a dot product of $\overrightarrow{\beta}$ and $\overrightarrow{y}\}$.
8: **end while**
9: Return $\cup_i L_i$.

2.3 Association Rule Mining from Big Data

We are living in the era of big data. More and more software systems are being used in our lives and businesses that generate a lot of data day by day. Thus, this creates both opportunities and challenges. As we acquire more and more data, data mining methods can be applied to provide more accurate results so that vast useful data relationships can be found. Three main challenges defined by Zikopoulos et al. [ZED+12] are volume, velocity, and variety. Let first talk about variety, one reason databases become so big is database integration. This means that there are many diverse data resources forming many database varieties. Hence, this requires prepossessing steps. Velocity of big data referred as generation speed of data is very fast. The database system requires having very fast speed to process data. Volume of big data means that big data has a very big volume. In order to process big data, database servers are required to have more steps to process data, much more computation power and memories. Traditional database methods that can be run by one physical server will no longer be sufficient to process big data. MapReduce is a data

process technology that original raw data will be partitioned into different blocks and delivered to different servers with a mapping schema like ⟨key, value⟩. Mapped data will be shuffled to the specific node by their keys and end up with the format of ⟨key, list of values⟩. The reduce phase of this process will run operations such aggressions, summarizations or filtering and then return outputs in each node concurrently.

2.3.1 Mining from Precise Big Data

Hadoop is the most widely used MapReduce technology. This is implemented through Hadoop Distributed File System (HDFS). Many data mining algorithms (e.g., [YLF10, LLH12, LZHS12, LMJ16, SSA+16]) have been proposed that use Hadoop technology. Still, as Hadoop has to communicate with I/O disks every time through MapReduce processes, this is much slower than consuming data resources from shared memory. Spark is claimed to have 10 times the speed faster than Hadoop as it uses Resilient Distributed Dataset (RDD) [ZCD+12]. On mining precise big data, YAFIM [QGYH14] is the first Apriori-based algorithm of using Spark. This algorithm has two phases. Phase 1 of YAFIM is to load the original database into RDD. It then uses MapReduce to get L_1. In Phase 2, YAFIM loads $L_{(k-1)}$ into RDD and then generates C_k. The algorithm searches the RDD referred to in the original database to generate L_k. The pseudocode is shown Algorithms 3 (Phase 1 of YAFIM) and 4 (Phase 2 of YAFIM). YAFIM greatly improves the speed but this algorithm cannot handle uncertain data.

Algorithm 3 Phase 1 of YAFIM

1: Load database to RDD
2: Map items from each transaction using flatMap
3: Compute frequency of each singletons by reduceByKey
4: Output items that are larger than minsup

Algorithm 4 Phase 2 of YAFIM

1: Load L_{k-1} to RDD
2: Load database to RDD
3: Generate C_k
4: Read through database
5: Map each itemset from C_k found transaction by flatMap
6: Compute frequency of each itemset by reduceBykey
7: Output to L_k for those are larger than minsup

2.3.2 Mining from Uncertain Big Data

On mining uncertain big data, Leung et al. [LMJ14] proposed a new algorithm that has adopted the big data technologies to mine uncertain data. It first conducts a database scan and then maps each item as the key and its uncertain value to be the value of the mapped record. In reduce phase, the process matches item keys and sum up all frequency of singletons. In the second MapReduce phase, the algorithm groups all transactions by frequent singletons and then delivers all mapped data to different nodes. In the reduce phase, all different nodes build projection sub trees to mine data concurrently. This approach is sufficient provided that all of the project trees fit into local memory. However, when mining big data, it is likely that projected trees cannot fit into local memory. Also, in the second MapReduce phase, as each projection tree requires all related data to singletons, it ends up having many duplicates of transaction data so that the overall data size becomes huge.

2.4 Summary

In this chapter, we explained association mining in precise data and in uncertain data. We observed that vertical mining (a) achieve great success in reducing memory consumption when mining precise data and (b) have great advantages in reducing

overall main memory in mining uncertain data. Big data provides challenges in data mining. Some current solutions in solving big data issues on data mining are demonstrated. From these solutions, we found that saving on memory consumptions is very important when designing data mining algorithms on big data. As vertical mining algorithms achieve great success in both mining precise data and uncertain data, we would like to extend vertical mining to mine big data.

Chapter 3

MapReduce-Based Vertical Mining with MR-UV-Eclat

Recall from Chapter 2 that advantages of vertical mining algorithms are scanning the original database only once and consuming very low resources. However, these algorithms cannot handle big data. On the other hand, YAFIM is very efficient in processing big data but cannot deal with uncertain data. Also, many big data algorithms have difficulties allocating balanced data. In order to resolve these issues and combine all these advantages from these algorithms, we propose MapReduce based UV-Eclat algorithm called MR-UV-Eclat.

3.1 Overview

MR-UV-Eclat is MapReduce version of UV-Eclat. For this conversion, we first need to understand how to create MapReduce behaviors in UV-Eclat. In this section, we first explain observations and approaches to accomplish these MapReduce behaviors. We then explain key steps in this algorithm. Finally, we provide an example to

further explain this algorithm.

3.2 MapReduce Behaviors

Recall that UV-Eclat only scans the original database once, it converts original data to vertical data. UV-Eclat only keeps track of items that exist in transactions. For example, let vector $\vec{d}$ be ⟨d, (1, 0.7), (3, 0.7), (5, 0.6), (6, 0.6)⟩, in order to form a new itemset, it requires doing intersection between two vectors. For the operation of doing intersections in UV-Eclat, only entries that have the same transactionIDs are kept and multiplied each other. For example, let us have two vectors ⟨a, (1, 0.1), (2, 0.2), (3, 0.3)⟩ and ⟨b, (1, 0.2), (2, 0.3)⟩. For forming the vector $\vec{ab}$, we intersect vector $\vec{a}$ with vector $\vec{b}$. we eliminate (3, 0.3) in vector $\vec{a}$ as it can not find any transaction ID match in vector $\vec{b}$. As the result, we have ⟨ab, (1, 0.1*0.2), (2, 0.2*0.3)⟩. We observe that there are two key matching behaviors here. First, the item key needs to be matched. For example, in order to generate vector $\vec{ab}$, we have to change vector $\vec{a}$ and vector $\vec{b}$ the item keys to be ab so that they match. Second, the transaction IDs have to be matched. Hence, in MR-UV-Eclat, the key of the mapped record becomes "item|transactionID". In this case, it can enable reduce phase to map not only the item keys but also transactionIDs. Hence, this algorithm first convert original data to be the format in ⟨item|transactionID, value⟩. In reduce phase, it collects data with the same key of "item|transactionID" and does the multiplication.

Suppose we have frequent singletons of {a, b, c, d, e}, we want to generate level 3 candidates from ab. It is to find the last entry of b and then base on the position of b presented in {a, b, c, d, e}, we find the subset of {c, d, e}. The level 3 candidates generated from ab are formed by doing concatenation of ab with one of {c, d, e}. As a result, we get candidates of {abc, abd, abe}. We observe that the key matching

process of the MapReduce behavior in MR-UV-Eclat is in transaction level. For the transaction of (a, b, c), the level 3 candidate generated by ab is only abc. In this case, we could have generated less candidates if we can generate candidates based on the information provided by each transaction.

Agarwal et al. [AAP01] proposed the tree projection algorithm for the generation of frequent itemsets. This algorithm first reads the original database and then builds the lexicographic tree structure. In addition to building the lexicographic tree, each node also contains potential suffix items for the next level candidate generations. Figure 3.1 illustrates tree project of transaction $abcd$. This tree structure is in lexicographic order and each level of nodes are only generated or expended when the level of itemsets is being mined. For example, when mining the very first level, it tries to generate singletons from the root node. It generates items of a, b, c, d. Also when reading the transaction $abcd$, it stores {b,c,d} as potential suffix set in node a. In node b, it stores {c, d} as potential suffix set. In node c, it generates {d} as potential suffix set. In node d, the suffix set of node d is empty. Each node is examined by calculating the frequency and then comparing with minimum support threshold. The nodes that are infrequent are removed from the tree. Only frequent nodes continue to expend for next level's examinations. One of the main advantage of this algorithm is that it can reduce candidate generation range. The other advantage is that the tree expansion is controllable so that only one level of expansion for each level of mining. This algorithm serves data mining in precise data. In order to apply this algorithm to mine uncertain data, it requires additional storage to store uncertain value for each node and uncertain values for all their potential suffix items. This becomes really disordered as we not only keep track of all uncertain values but also their corresponding transactionIDs.

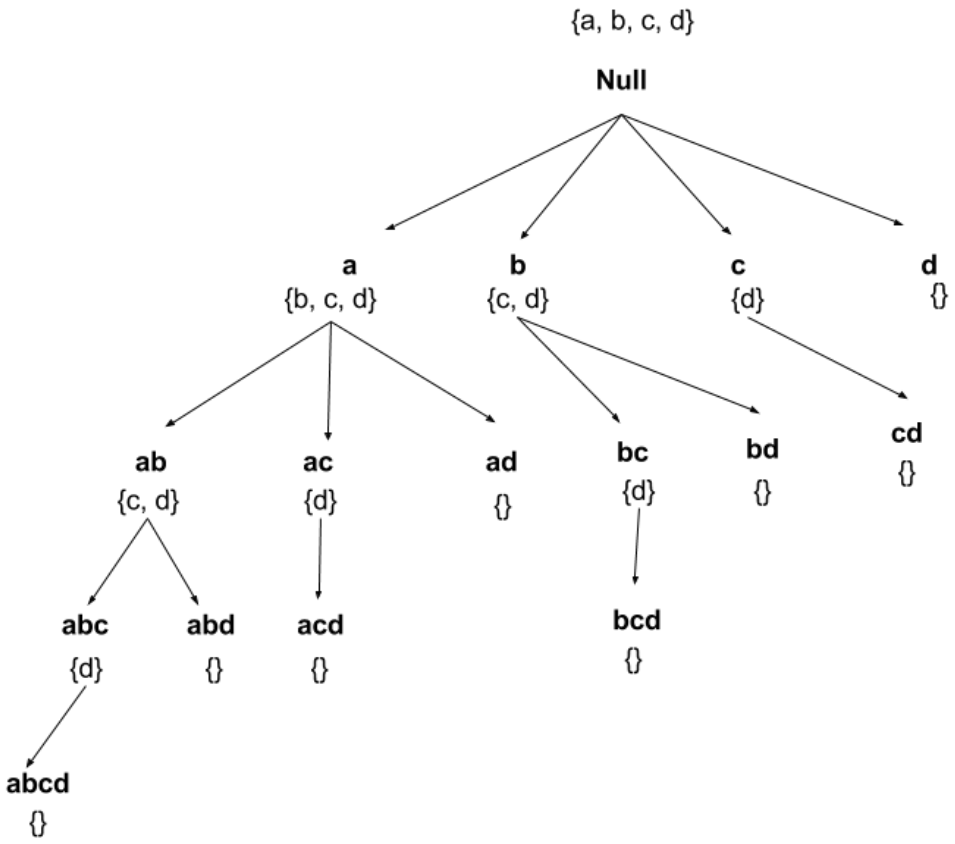


Figure 3.1: Tree-Projection

We borrow ideas of the tree projection algorithm and integrate this tree structure into MR-UV-Eclat. As we try to match item key and transaction ID when doing the reduce operation, the map key needs to contain item key and transaction ID. In addition, we need to store the support value on the corresponding an item and the specific transaction, this value is stored in the mapped key. Hence, the format of a key is like "itemKey|transactionID|supportValue". What should store in the value of the mapped record? We borrow the idea of the tree project structure. Each mapped record represent a node in a tree structure and each tree structure only represents a transaction. Thus, the value of each mapped record contains potential suffix items.

Unlike the original tree project structure, each suffix item not only contains the singleton but also its support value in the corresponding transaction. There is a remap process in each mining process. The remap process is as follow: (1) Form a new key by concatenating the item key with each suffix item, (2) multiply the support value with each of uncertain value of each suffix item. Figure 3.2 illustrates this MapReduce version of the tree-projection structure. We assume the transaction is in lexicographic order. This original data are converted to 4 nodes. For example, for the first node, the item key is a and the transaction ID is 1. In addition, the support value is 0.8. Hence, the mapped key is "a|1|0.8". The potential suffix singletons with their support values are stored in the mapped value in the mapped record. In the case of node a, this value is (b:0.8, c:0.7, d:0.6). Notice node d has empty set in the mapped value. This is because no item is after item d. In order to generate the next level of candidates, this algorithm concatenates the item key with each of suffix items. The support value for the new candidate in the current transaction is calculated by multiplying the support value with the uncertain value of the suffix item. In the case of node a, it generated 3 nodes for next level because there are 3 suffix items in the mapped value. They are ab, ac and ad. The corresponding support values are 0.8*0.8=0.64, 0.8*0.7=0.56 and 0.8*0.6=0.48. The suffix items for the each node are items after the visited item. For example, the suffix items for ab are (c:0.7, d:0.6). The suffix items for ac are (d:0.6) and the suffix items for ad are ().

Now the question we should ask is how to calculate the total frequency of the itemset. Through the remap process, we expand each node to generate candidates and calculate their support values for specific transactions. In order to calculate the final support values for the candidates, we require the other remap process. We observe that each mapped key contains the itemset key, transaction ID and its support

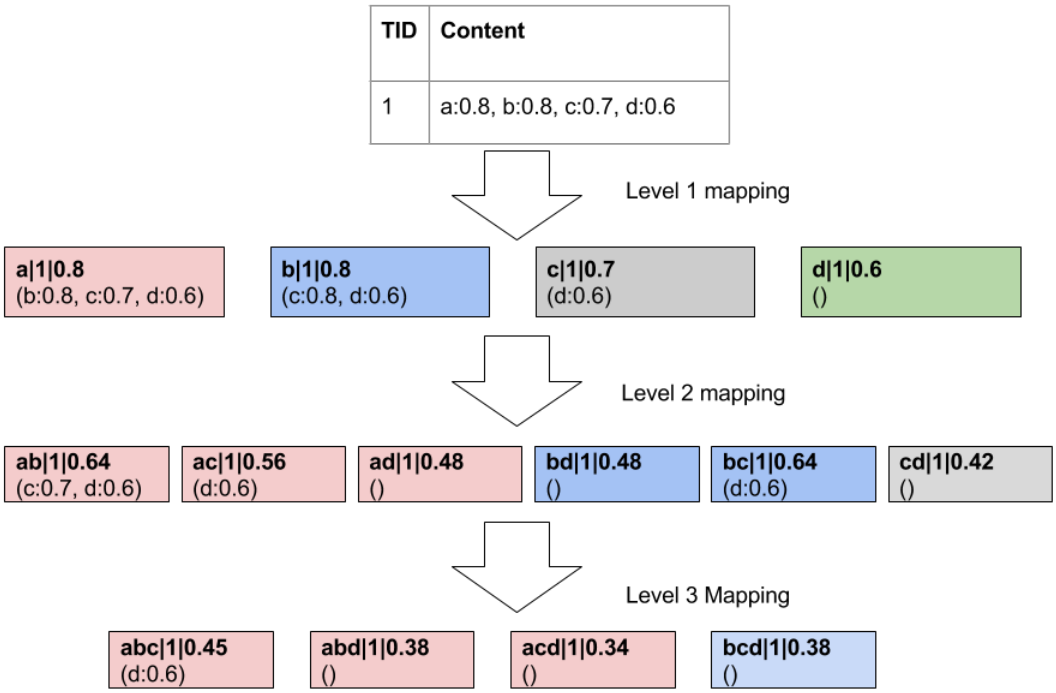


Figure 3.2: MapReduce Tree Projection

value. To get the final support value, we need to sum up these values with the same itemset key. This means that we need one more MapReduce process to conduct this calculation. First of all, we need to build up the other RDD which is in the format of (itemsetKey, supportValue). We can get all these from the mapped key from RDD that we just created before then calculate the final support value in the reduce process.

3.3 Detailed Implementation

MR-UV-Eclat reads the original database and then converts this data to vertical data. Frequent itemsets are generated and collected by each iteration following can-

didate node expansion and frequency verifications. In this section, implementations on vertical data conversion, candidate node expansion and frequency verifications are explained in detail. Finally, steps are outlined with an example.

3.3.1 Conversion to Vertical Data

MR-UV-Eclat first reads each transaction of the original database. When reading each transaction, it constructs tree projection for the first level. Each node becomes the mapped data. As this is for the level 1 data, the tree is only expanded to 1 level below "NULL" node. The value of each mapped record contains potential suffix items for the other level of expansion. After the expansion is done, the next operation is frequency examination. This operation is achieved through the other MapReduce operation. First, the algorithm goes over each mapped key, put the item key as a new key, and support value as the new value. In the reduce phase, it sums up the frequency in all transactions. The final support values are compared with the minimum support threshold. Only frequent items are kept in L and L_n.

3.3.2 Tree-Expansion and Examination

After vertical data conversion, each node is free to expand. From the mapped key, we extract the current itemset, transaction ID and support value. The value of the mapped record is potential suffix nodes, each of them being potential suffix singletons with its support value in the current transaction. The purpose of tree expansion is to concatenate the current itemset with each suffix node in the list. This new formed candidate is then verified by a pruning process with the input of L_{n-1}. If this candidate passes the examination of the pruning process, the new support value is then calculated by multiplying the current support value with the support value

Algorithm 5 MR-UV-Eclat Convert to Vertical Data

```
1: function CONVERT_TO_VERTICAL(D, minsup, L_1)
2:     Load D in RDD
3:     for each transaction t in D do
4:         num_tran = number of transactions
5:         tranID = transaction ID
6:         initial list tranLst
7:         for each item i in t do
8:             ik = item key
9:             sup = support value
10:            Add (ik,sup) to tranLst
11:        end for
12:        len = size of tranLst
13:        for each item i in tranLst do
14:            p = position of i in tranLst
15:            ik = item key
16:            sup = support value
17:            for each item i in lst between (p+1, len) do
18:                initial list suffixList
19:                Add i to suffixList
20:            end for
21:            form key "ik|tranID|sup" and assign to prefixKey
22:            map (prefixKey, suffixList)
23:        end for
24:    end for                                          ▷ This form D_v
25:    for each (prefixKey, suffixList) m in D_v do
26:        ik = m.itemKey
27:        sup = m.SupportValue
28:        map (ik, sup)
29:    end for
30:    Reduce (key, list of support value) → (key, sum of support value)   ▷ This
   form C_1
31:    for each (k, sup) in C_1 do
32:        if sup ≥ minsup then
33:            assign k to L_1
34:        end if
35:    end for
36:    L = L ∪ L_1
37: end function
```

in that suffix node in the list. The new candidate, the current transaction ID and the new support value together form the new mapped key. The new value of the mapped record is the rest of suffix nodes after the current visited suffix node. After this tree expansion, all new candidate vectors are generated. We still need to utilize the MapReduce operation to conduct frequency examination. First, through the map process, we extract the candidate itemset and its support value from the mapped records formed previously. Second, through the reduce process, the algorithm sums up all support value and form the final output format of (itemset, calculated support value). The current support value is compared with minimum support threshold. Only records that are larger than the minimum support threshold are kept. These data are then kept in L_n.

3.3.3 Illustrative Example

The algorithm first converts the original database to a vertical data. L_1 and L store frequent singletons. Tree expansion and examination operations are being called until the size of current level of frequent itemset is smaller than 1.

Let us illustrate this algorithm in the following example. We will still use the sample database in Table 3.1. Let minimum threshold equal 3 and the algorithm is executed in a master-worker structure. There are three machines, one master and

Table 3.1: Sample database 2

TID	content
1	a:0.9, b:0.9, d:0.7, e:1.0
2	b:0.8, c:0.8, e:1.0
3	a:0.9, b:0.9, d:0.7, e:1.0
4	a:0.9, b:0.9, c:0.7, e:1.0
5	a:0.8, b:0.8, c:0.7, d:0.6, e:1.0
6	b:0.8, c:0.7, d:0.6

Algorithm 6 MR-UV-Eclat Tree Expansion and Examination

```
1: function TREE_EXPANSION_EXAMINATION(D_v, minsup, L_{k-1})
2:     for each item i in D_v do
3:         ik = i.key.itemSet
4:         sup = i.key.supportValue
5:         tranID = i.key.TransactionID
6:         suffixList = i.value
7:         for each item v in suffixList do
8:             suf = v.item
9:             p = position of v in suffixList
10:            len = size of suffixList
11:            sv = v.SupportValue
12:            prefixKey = ik + "_" + suf
13:            if pruning(prefixKey, L_{k-1}) then
14:                new_sup = sup * sv
15:                initial newList
16:                for each item v2 in suffixList between (p+1, len) do
17:                    add v2 to newSuffixList
18:                end for
19:                form key new_prefixKey of "prefixKey|tranID|sup"
20:                map (new_prefixKey, newSuffixList)
21:            end if
22:        end for
23:    end for                                          ▷ This form new D_v
24:    for each item i in D_v do
25:        ik = i.key.itemSet
26:        val = i.key.SuportValue
27:        map (ik, val)
28:    end for
29:    Reduce (ik,list of Support Value) to (ik, sum of Support Values) ▷ This form
       C_k
30:    for each item (k, sup) in C_k do
31:        if sup ≥ minsup then
32:            add k to L_k
33:        end if
34:    end for
35:    L = L ∪ L_k
36: end function
```

Algorithm 7 MR-UV-Eclat Main

```
1: function MAIN(D, minsup)
2:     k = 1
3:     Initial L, L_k
4:     D_v = CONVERT_TO_VERTICAL(D, minsup, L_1)
5:     while size of L_k > 1 do
6:         k++
7:         D_v = TREE_EXPANSION_EXAMINATION(D_v, minsup, L_{k-1})
8:     end while
9: end function
```

two workers. The master machine reads the original database and delivers these data to workers. Each worker reads transactions and build them into tree-projection structures. Each node has the key format of "item|transactionID|supportValue". The value of each node has other items in lexicographic order. For example, Worker 1 machine read the transaction with the transaction ID obtaining the value is {a:0.9, b:0.9, d:0.7, e:1.0}. It converts this value to ⟨a|1|0.9, (b:0.9, d:0.7, e:1.0)⟩, ⟨b|1|0.9, (d:0.7, e:1.0)⟩, ⟨d|1|0.7, (e:1.0)⟩ and ⟨e|1|1.0, ()⟩. This process results RDD called D_v. This is used in the tree expansion. After this mapping phase is done, the other mapping phase is executed to extract itemsets and support values from the previous mapped records. The function looks like (itemset|transactionID|supportValue, value) → (itemset, supportValue). In the case of item a, ⟨a|1|0.9, (b:0.9, d:0.7, e:1.0)⟩ → ⟨a, 0.9⟩, ⟨a|3|0.9, (b:0.9, d:0.7, e:1.0)⟩ → ⟨a, 0.9⟩, ⟨a|4|0.9, (b:0.9, c:0.7, e:1.0)⟩ → ⟨a, 0.9⟩, ⟨a|5|0.8, (b:0.8, c:0.7, d:0.6, e:1.0)⟩ → ⟨a, 0.8⟩. In reduce phase, it becomes ⟨a, (0.9, 0.9, 0.9, 0.8)⟩ → ⟨a, 0.9+0.9+0.9+0.8⟩ → ⟨a, 3.5⟩. As 3.5 is larger than minimum threshold 3, item a is kept in L_1 and L. Other items are put through the similar process as we deal with the item a. As a result, we get L={a, b, e} and L_1={a, b, e}. This is illustrated in Figure 3.3.

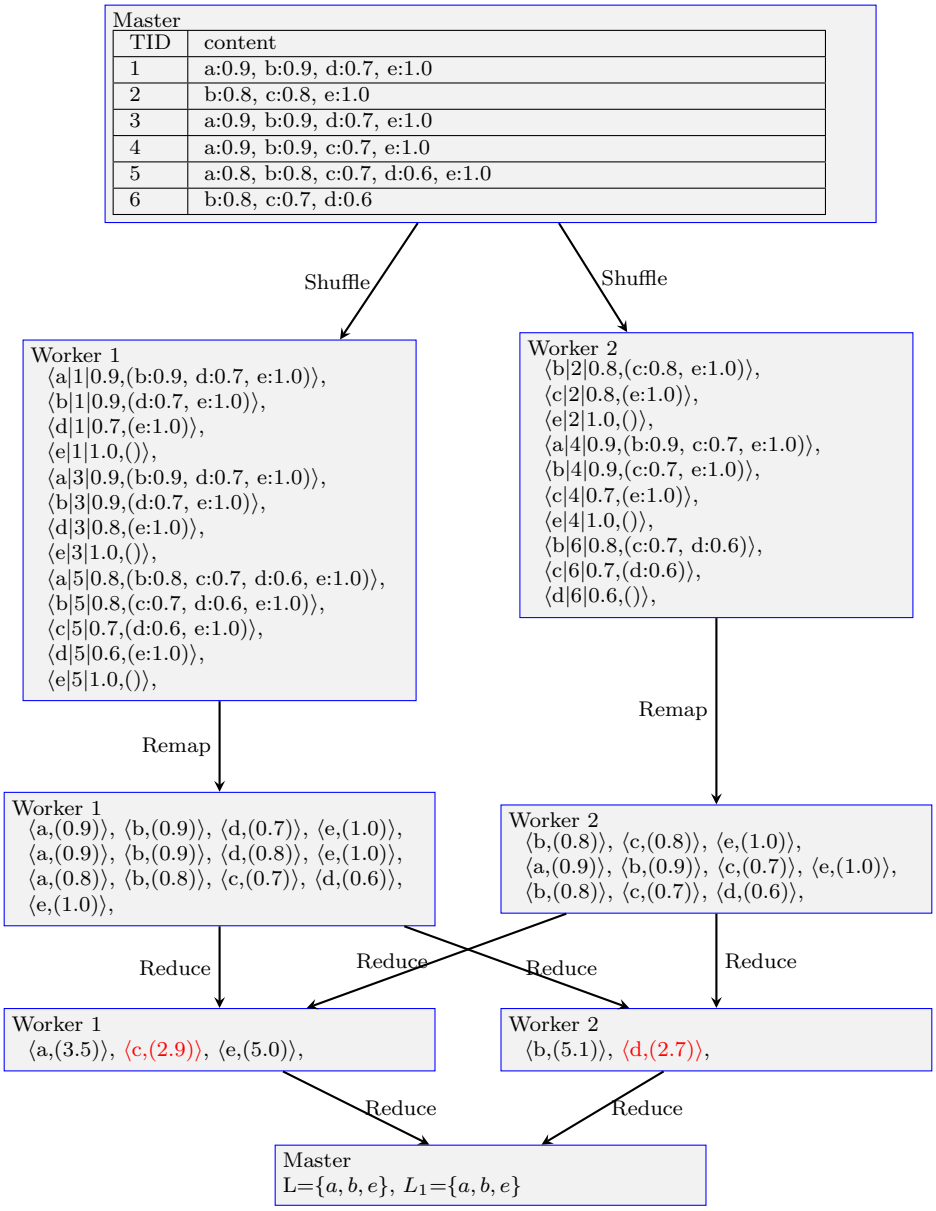


Figure 3.3: MR-UV-Eclat:Convert to Vertical data

Now that we have L_1={a, b, e}, the size is larger than 1. We can generate level 2 candidates. The set of these candidates are generated by expanding nodes that are in D_v. During the tree expansion, the algorithm goes through each node to try to join the current itemset with each of the singletons and multiply the support value of the

current itemset with the support value of the current visited singleton in the value list. For example, for the node ⟨a|1|0.9, (b:0.9, d:0.7, e:1.0)⟩, we first extract item a, support value of 0.9 and transaction ID 1 from the key "a|1|0.9". This algorithm then goes through the list of (b:0.9, d:0.7, e:1.0). It first reads singleton (b:0.9). By combining the current itemset $\{a\}$ and the singleton $\{b\}$, we get the new itemset key ab. During the pruning step, ab can be sub-divided into items a and b, both a and b are in L_1. Hence, item key ab passes the pruning test. For calculating the support value, we simply multiply the support value of a, which is 0.9 and the support value of b, which is 0.9. We get the support value of 0.81. For forming the new value for the new mapped record, we simply grab the singletons after item b in the list and at the same time we evaluate the list and remove singletons that are not in L_1. As the result, we get (e:1.0). Finally, if we combine everything together, we get the new mapped record, ⟨ab|1|0.81, (e:1.0)⟩. This process described above goes through singletons $\{d\}$ and $\{e\}$ in the list. The new itemset ad is pruned away as d is not in L_1. We still form the other new mapped record which is ⟨ae|1|0.9, ()⟩. After this remap process, the other map phase conducts. The algorithm goes through each node to extract the itemset and support value from the key. As a result, new mapped records contain itemsets as keys and their support values as values. We then apply for reduce operations and get the final support value for each generated candidates. Only frequent candidates are kept in L_2 and L. At the end of this process, we have L_2={ab, ae, be} and L={a, b, e, ab, ae, be}.

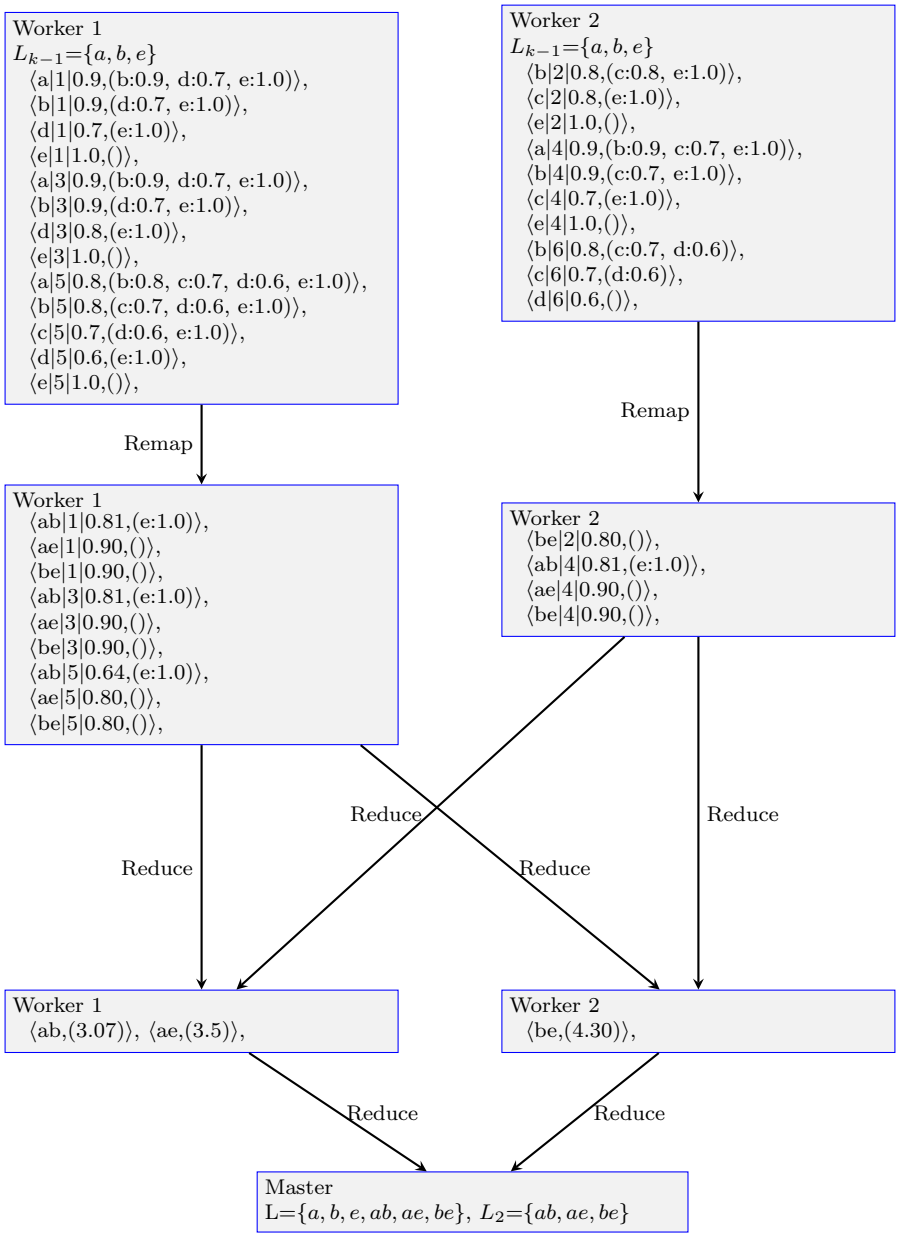


Figure 3.4: MR-UV-Eclat:Tree Expansion and Examination at level 2

As L_2 has more than 1 itemset, the tree expansion and examination process continue. The only candidate is *abe*. The algorithm then starts the pruning process. itemset *abe* can have subsets of ab, ae, and be. They all belong to L_2. Hence, itemset *abe* passes the pruning test. In the remap process, we get ⟨abe, 0.81⟩, ⟨abe, 0.81⟩ and

⟨abe, 0.64⟩. In the reduce phase, we have the input of ⟨abe, (0.81, 0.81, 0.64)⟩. In the reduce function, we calculate the frequency by doing the following ⟨abe, (0.81, 0.81, 0.81, 0.64)⟩ → ⟨abe, 0.81+0.81+0.81+0.64⟩ → ⟨abe, 3.07⟩. As 3.07 is larger than minimum support threshold, the itemset *abe* is frequent. We have L_3={abe} and L = {a, b, e, ab, ae, be, abe}. As in L_3, there is only one itemset and so the algorithm stops.

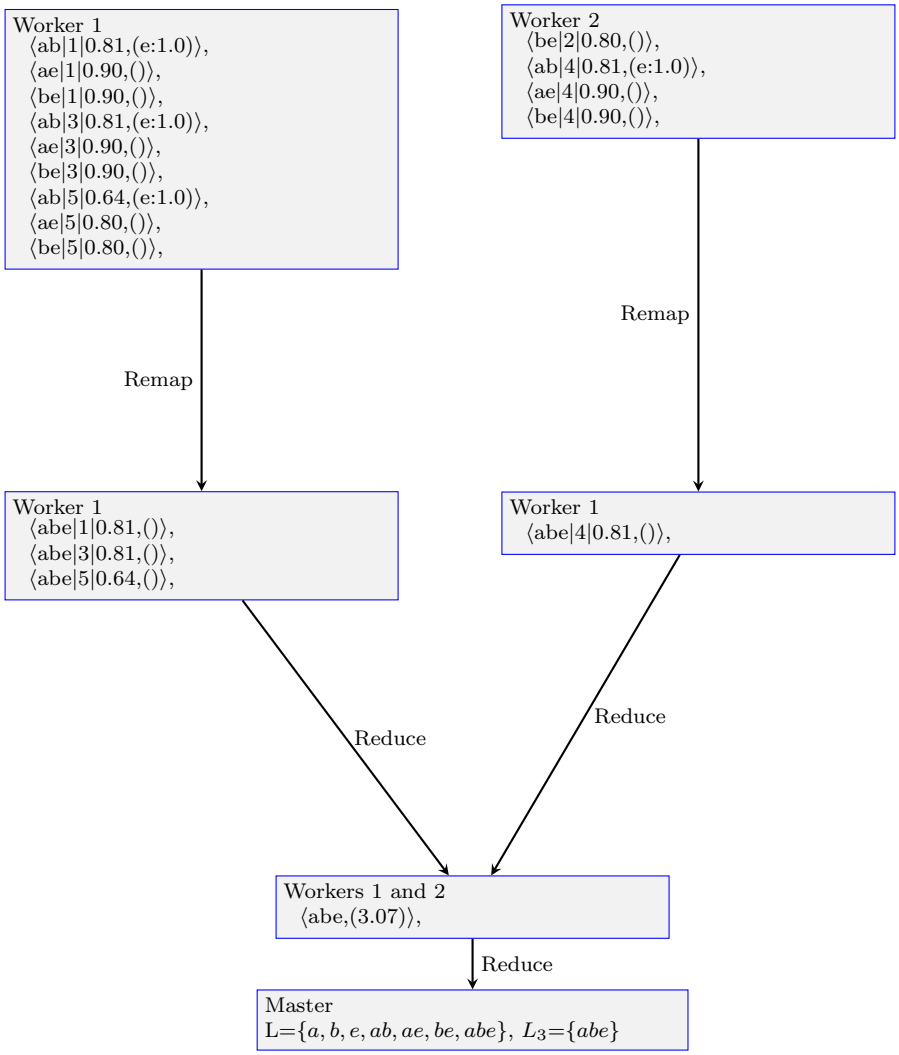


Figure 3.5: MR-UV-Eclat:Tree Expansion and Examination at level 3

3.4 Summary

In this chapter, we proposed MR-UV-Eclat algorithm. As MR-UV-Eclat is adopted from UV-Eclat, we explained logics on how to convert behaviors in UV-Eclat to MapReduce behaviors. We observed that when two vectors are intersected they become a new vector. One thing the two have vectors in common is that final key is the combination of the two old keys. Hence, by replacing the old keys with the new key for both vectors in the map phase, they can be grouped in the reduce phase. To increase the parallelism, each worker node only deal with one candidate in each transaction. Each mapped value is a projection tree. A detailed example illustrated the algorithm.

Chapter 4

MapReduce-Based Vertical Mining with MR-U-VIPER

There are two major algorithms for vertical mining. In the previous chapter, we applied spark technologies in UV-Eclat to become our proposed algorithm called MR-UV-Eclat. To make it completed, we also propose Spark version of U-VIPER called MR-U-VIPER.

4.1 Overview

In order to extend U-VIPER algorithm to adopt the big data technology Spark and to make this algorithm efficient, two puzzles are needed to be solved. First, we need to know how to apply MapReduce behaviors in a U-VIPER algorithm. Second, we need to reduce over all memory consumption in order to improve performance of the algorithm. For the next two section, we will explain how to solve these two puzzles in detail. After that, detail steps and an example are provided.

4.2 Map-Reduce Behaviors

One of the main task for MR-U-VIPER is to extend U-VIPER algorithm to adopt the big data technology Spark. We first need to create MapReduce behaviors for U-VIPER. Once the MapReduce codes are complied, all these complied resources are delivered to different servers. Codes are executed on all worker nodes independently. The master server reads input files and then maps each record to be the format of (key, value). The mapped data are sent to buffers called shuffles. All worker nodes pick up different mapped data from shuffles and store them locally. All worker nodes then group mapped data that have the same key into datasets. In this process, all servers bring their mapped data to shuffles again. They then exchange mapped data with shuffles to acquire the grouped dataset. Eventually, the reduce method implement operations (aggregation, summarization etc.) and produce final outputs. The process on converting vertical data in U-VIPER can be written as a map phase. For the map data, we need to make the key to be an item, and the value to be the vector array. In order to form new candidates, the U-VIPER algorithm requires vectors of the current level to intersect with vectors of the frequent singletons. In this case, vectors of the current level are actually the prefix dataset. Vectors of singletons are the suffix dataset. The newly formed vectors become the new prefix dataset for the next intersection operation. For example, vector $\overrightarrow{a}$ is intersected with vector $\overrightarrow{b}$ to form vector $\overrightarrow{ab}$. We observe that the key a becomes the key ab and the key b becomes the key ab. Also suppose we have all frequent singletons of {a, b, c, d}, the prefix a can be formed with {b, c, d}. Hence, keys of ab, ac, ad and ae are formed. To form vectors $\overrightarrow{ab}$, $\overrightarrow{ac}$, $\overrightarrow{ad}$ and $\overrightarrow{ae}$, vector $\overrightarrow{a}$ needs to duplicate four times and each duplicated key of a becomes one of ab, ac, ad and ae. In addition, vectors of b, c, d and e becomes vectors of ab, ac, ad and ae. In sum, the mapping phase for this example is

as follows: (a, vector1) → (ab, vector1), (ac, vector1), (ad, vector1), (ae, vector1), (b, vector2) → (ab, vector2), (c, vector3) → (ac, vector3), (d, vector4) → (ad, vector4), (e, vector5) → (ae, vector5). Thus, in reduce phase, the program can find matching pairs to form the new vectors. In reduce phase, {(ab, vector1), (ab, vector2)} → (ab, vector1 * vector2), {(ac, vector1), (ac, vector3)} → (ac, vector1 * vector3), {(ad, vector1), (ad, vector4)} → (ad, vector1 * vector4), {(ae, vector1), (ae, vector5)} → (ae, vector1 * vector5).

4.2.1 In-Memory Consumption

In either map or reduce phase, Hadoop stores data in hard drives. This incurs much I/O cost. To improve the performance, Spark introduces the data structure called Resilient Distributed Datasets (RDD). This data structure allows in memory reference and computation. Thus, it speeds up the MapReduce process. However, the effect of RDD fades when data cannot fit into the main memory. In this case, RDD has to push some data into hard disk for the temporary storage. When the data are needed for the program, RDD has to bring these data to memory for use. This means that the program has to either delete a portion of unused data in memory or push the unused data into hard disk storage again. Hence, the best way to improve the overall performance in Spark programming is to lower memory consumption when designing the algorithm. U-VIPER converts original database to vertical data. Unlike UV-Eclat, each vector in U-VIPER has the same size, which is the total number of transactions. Uncertain data are usually in decimal format. This requires using the data type of double or float. This means each item value is at least 32 bits (float). However, as the number of transactions dramatically increase, the size of overall dataset becomes very big. For instance, suppose the original database has

10,000 transactions and there are 1,000 items. In order to create vertical dataset for the U-VIPER algorithm, it requires at 320,000,000 bits. In many cases, users may want only 2 decimal places. For example, for value of 0.234, we can round it up to be 0.23. Hence, we do not need the precision of data types like double and float. If we multiply 100 with these values, these values become all integers. In this case, the range of these values is between 0 and 100, we can squeeze these values to be the format of byte. The data size of byte is 8 bits. Take the dataset we mentioned before, we can reduce the data size from 320,000,000 bits to be 80,000,000 bits. For MR-U-VIPER, we convert the original database to vertical data that have many byte values. Thus, the over all data size can be reduced and fit more data into the main memory.

4.3 Detailed Implementation

Our proposed MR-U-VIPER will first convert the original database into vertical data. It will apply frequency counts for each vector. Infrequent vectors are filtered in these mapped vertical data. L_1 is formed and kept in the main memory. These vertical data are then duplicated and generated to have two sets of data. One set is for prefix data and the other set is for suffix data. In order to form frequent itemsets in each level, the algorithm will do the following. (1) Form new candidate prefix data. (2) Form new candidate suffix data. (3) Union prefix and suffix data and then conduct a frequency count. In the following sections, we will detail the main steps for this algorithm. At the end, summarized steps are provided.

4.3.1 Conversion to Vertical Data

When reading the original database, if the item is present in the transaction, the value of the item will present inside the vector of the item. If the item is not present in the transaction, it will show 0 in the stated position of the vector of the item. As the result of the first map phase, vertical data are formed. These vertical data are also mapped data that have the format of (key: item, vector). This vector is an array with the total size of the number of transactions. To calculate the support value of an item, sum up each value of the vector. Only mapped records with support value larger than minimum support threshold are kept in the mapped dataset. These mapped dataset is then duplicated into two copies. One is for prefix mapped dataset. The other one is for suffix mapped dataset. The prefix dataset and suffix dataset are then changed to form the next level of candidate dataset. We called this process remap.

4.3.2 Formation of the Prefix Data

The remap phase of prefix data is to form one of the combined values of the next candidates. It accepts two parameters which are L_1 and the current frequent itemsets. The algorithm goes through each mapped record and finds the diffset. Note that L_1 is an ordered list. The algorithm only finds the last entry of the current mapped itemsets and then find the position this entry in the L_1. The diffset is between the next item of this entry until the end of L_1. Each mapped itemset will be combined with its own diffset to form new keys. To do the concatenation insert one of the diffset at the end of the current key. For example, if there are 3 items in the diffset, there will be 3 keys created. The new mapped records are formed by combining the new key with the vector to which the original mapped itemset belong to. In between,

Algorithm 8 MR-U-VIPER Convert to Vertical Data

```
1: function CONVERT_TO_VERTICAL(D, minsup, num_item, L_1)
2:     Load D in RDD
3:     for each transaction t in D do
4:         Initial itemVal[num_item]
5:         for each item i in t do
6:             itemVal[i] = val
7:         end for
8:         for each item i in itemVal do
9:             map (i, itemVal[i])
10:        end for
11:    end for
12:
13:    Reduce (key, list of value) − > (key, vector)
14:                                              ▷ This form D_v
15:
16:    for each (k, v) in D_v do
17:        sum up all value in v and assign the result to r
18:        if r < minsup then
19:            remove (k, v) in D_v
20:        else
21:            assign k to L_1
22:        end if
23:    end for
24:
25: end function
```

each newly formed will apply the pruning process with the help of referencing the current frequent itemsets.

4.3.3 Formation of the Suffix Data

The remap phase of suffix data is to ensure the suffix value joined with the prefix candidates created in the process of forming the new prefix data. For example, we have a prefix mapped record of (key:ab, vector1) and a suffix mapped record of (key:c, vector2). In order to form itemset abc, we need to make sure these mapped record of

Algorithm 9 MR-U-VIPER form Prefix Dataset

```
1: function FORM_PREFIX(DP_{k-1}, L_1, L_{k-1})
2:     load DP_{k-1} to RDD
3:
4:     for each (k,v) in DP_{k-1} do
5:         Find last entry of k and assign the position to l
6:         Assign last position of L_1 to r
7:         get a subset in L1 between position l to position r
8:         assign to subset to S
9:
10:        for each i in S do
11:            create new key that concatenate v with k (like k_v)
12:            get new key of nk
13:            map (nk, v)
14:        end for
15:    end for                                   ▷ This result in DP_k
16: end function
```

ab and *c* are joined. The only way for them to be joined in the MapReduce process is to make them have the same key. Hence, we made (key: ab, vector1) to be (key:abc, vector1) and (key:c, vector2) to be (key:abc, vector2). In the reduce phase, as these two new mapped records have the same key, they can form a new record that is (key:abc, vector1 * vector2). When trying to form L_2 to C_3 in suffix data, (key:c, vector2) may be already replaced by (key:ac, vector2) and (key:bc, vector2). In this case, we first find the diffset. First of all, we need to find the boundary. We have the right boundary to be the position of the second last entry of the current itemset. The left boundary is the first position of L_1. We acquire a new key by inserting each of diffset in front of the current itemset. For example, suppose L_1 is {a, b, c, d, e}, for the current suffix dataset of (key:ac, vector2) and (key:bc, vector2). The second last entry for the key *ac* is *a*. The position for *a* in L_1 is 0. Hence, we get an empty diffset. The record (key:ac, vector2) is discard as it can not create new level of suffix candidates. For the record (key:bc, vector2). The second last entry is *b*. The position

of b in L_1 is 1. So the diffset is {a}. We then insert a in the front of bc, so the new mapped record is (key:abc, vector2).

Algorithm 10 MR-U-VIPER form Suffix DataSet

```
1: function FORM_SUFFIX(DS_{k-1}, L_1, L_{k-1})
2:     load DS_{k-1} to RDD
3:
4:     for each (k,v) in DS_{k-1} do
5:         Find the second last entry of k and assign the position to r
6:         Find the first entry of k and assign the position to l
7:         get a subset in L_1 between position l to position r
8:         assign to subset to S
9:
10:        for each i in S do
11:            create new key that concatenate v with k (like v_k)
12:            get new key of nk
13:            map (nk, v)
14:        end for
15:    end for                                   ▷ This result in DS_k
16: end function
```

4.3.4 Union and Reduce Phase

Once formations of prefix dataset and suffix dataset are done, there will be a matching pair for each of keys in prefix and suffix. In order to continue the reduce operation, we can union both the prefix and suffix datasets into one dataset. We can then conduct the reduce operation. In a reduce operation, the algorithm groups two mapped record in a single mapped record of (key, vector1 * vector2). The multiplication of two vectors is done by multiplying each value of the same position of each vector. As a result, we have outputs in the format of (key, vector3). In order to compute frequency, the algorithm goes through each of these records again and then calculates the support value. This will create the other dataset with keys and their support value. Only itemsets with their support values are more than the minimum

support value are kept and they become frequent itemsets L. The reduced dataset becomes the prefix data set for the next level candidate formations.

4.3.5 Illustrative Example

There is still a main function to control all of the steps described above. This main function controls all flows. The algorithm receives the input file and then loads the file to RDD. It then does the vertical data conversion. This dataset is then duplicated to two datasets. One is for prefix data and the other is for suffix data. The prefix dataset and the suffix dataset are then remapped for the next level candidate datasets. They are then union to one set of data. This becomes the new prefix dataset. Through the reduce step, each duplicated key becomes a unique key with its calculated vector. Frequency verifications are then applied. Only frequent data are mapped to the final prefix data. L_k is generated. These steps of prefix formation, suffix formation and union and reduce step are repeated until the size of L_k is only 1 or 0.

Example 4.3.1. In order to provide a clear view for the algorithm, we present an example to illustrate the steps and the detail for this algorithm. Suppose we use a master-worker server structure. There are two worker nodes, which are Workers 1 and 2. The master server executes operations in the main function. Worker servers execute other sub-routines such as prefix formation, suffix formations and so on. Observed from Figure 4.1, the master receives the input and also minsup (in this case it is 3). Through the map phase, each item with its value become ⟨key:item, (tid,value *100)⟩. Items that are not in the transaction are mapped to ⟨key:item, (tid, 0)⟩. In the reduce phase, the list of values with the same key become ⟨key:item, vector⟩. This vector is an ordered array with each entry of value by the order to transaction ID. The first row of the table contains all keys, and each column is the

Algorithm 11 MR-U-VIPER

```
1: function MAIN(D,minsup)
2:     k = 1
3:     Initial L, L_1, L_k
4:     Get total number of items from D and assign it to num_item
5:     D_v = CONVERT_TO_VERTICAL(D, minsup, num_item, L_1)
6:     duplicate D_v to DP_1 and DS_1
7:     L = L ∪ L_1
8:     L_k = L_1
9:     while size of L_k > 1 do
10:        k++
11:        DP_k = FORM_PREFIX(DP_{k-1}, L_1, L_{k-1})
12:        DS_k = FORM_SUFFIX(DS_{k-1}, L_1, L_{k-1})
13:        DP_k = DP_k ∪ DS_k
14:        function REDUCE_TO_VECTOR(DP_k, minsup)
15:            for each (key, list of vectors) in DP_k do
16:                initial vector3
17:                initial freq = 0, i = 0
18:                for each v1, v2 in list of vectors do
19:                    vector3[i] = v1*v2
20:                    freq += vector3[i]
21:                    i++
22:                end for
23:                if freq >= minsup then
24:                    map (key,vector3)
25:                end if
26:            end for
27:        end function
28:
29:        L_k = frequency count of DP_k
30:        L = L ∪ L_k
31:    end while
32:
33: end function
```

vector with corresponded key. Through MapReduce steps, all data are distributed to each workers. In this case, Worker 1 contains vectors of a, c and e and Worker 2 contains vectors of b and d. Each worker conducts the frequency count operation. For example, to calculate the frequency count of c, we can do (0+80+0+70+70+70)/100

= 2. We get that the frequency of c is less than minsup of 3. The vector of c is removed from Worker 1. As the result of the frequency count operation, we delete vectors of c and d. We get L_1 = {a,b,e}, L_k={a,b,e} and L={a,b,e}. This vertical data are duplicated to two datasets, prefix and suffix. They are also stored in Workers 1 and 2.

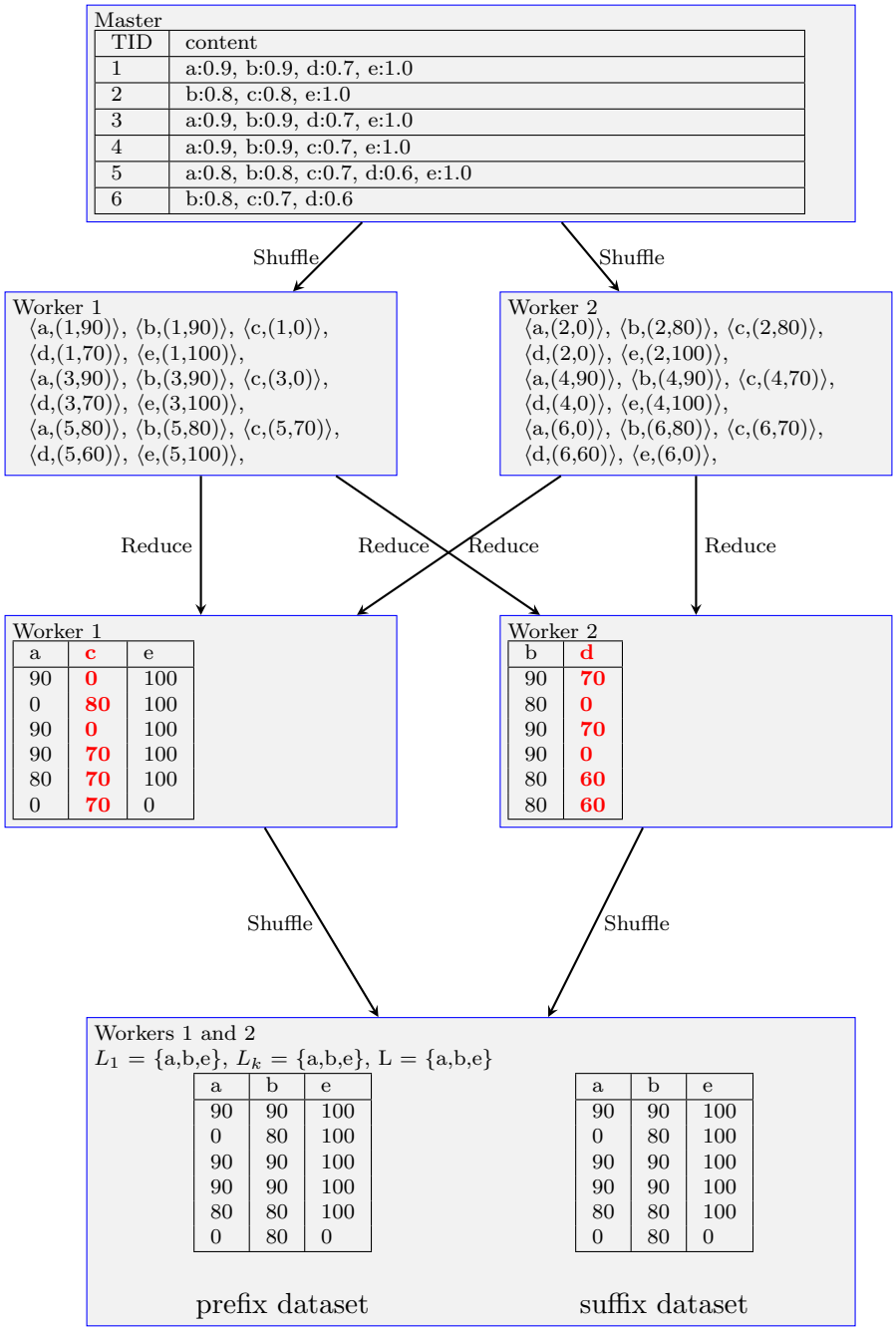


Figure 4.1: MR-U-VIPER:Convert to Vertical Data

Prefix data are stored partially in both Workers 1 and 2. In order to conduct the remap phase, these two servers are required to exchange data through shuffle buffers.

Each worker also receives L_1 and L_k. Each worker utilizes L_1 and each vector to form mapped data. For example, for mapped data $\langle$a, (90, 0, 90, 90, 80, 0)$\rangle$, we know the position of a in L_1 is 0. In this case, we need subset of {b, e} to form new keys. As the result, we have new mapped data of ab and ae. The new mapped data become $\langle$ab, (90, 0, 90, 90, 80, 0)$\rangle$ and $\langle$ae, (90, 0, 90, 90, 80, 0)$\rangle$. In the process of prefix formation, we also apply the pruning process by using L_k.

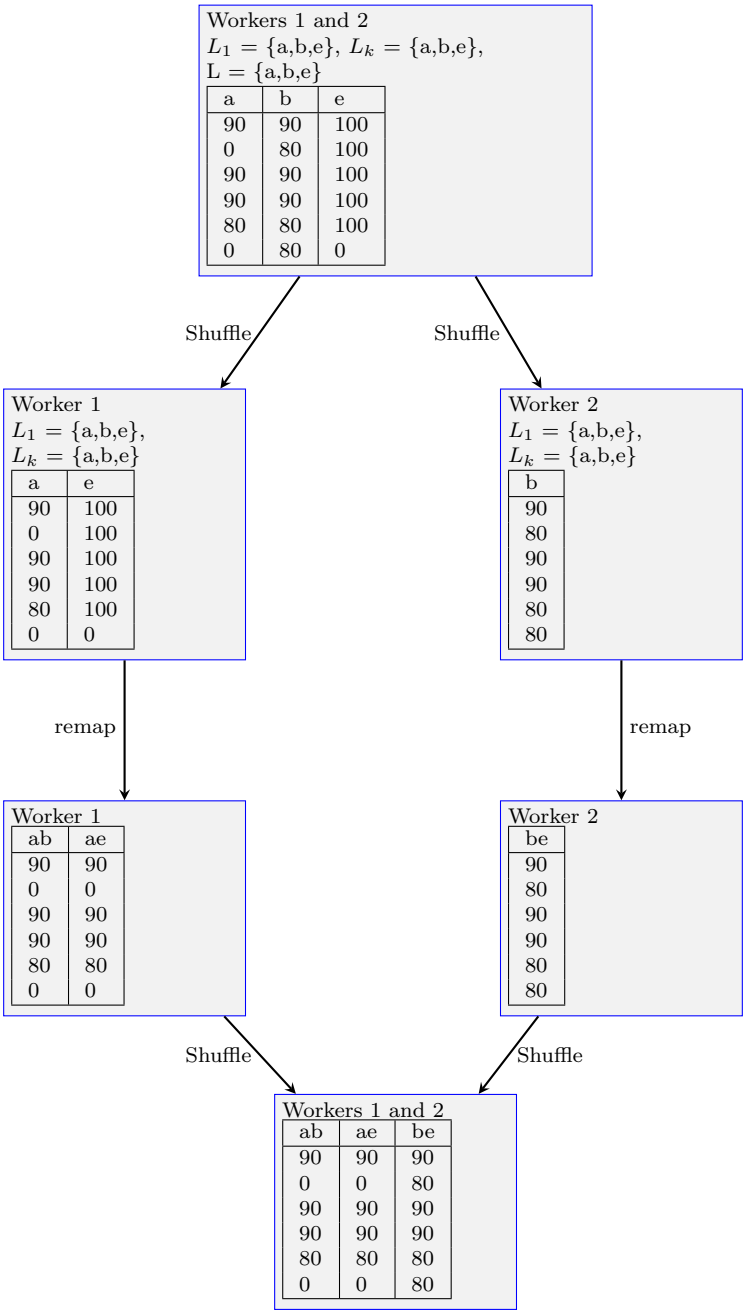


Figure 4.2: MR-U-VIPER:Prefix Formation at Level 2

Suffix data are also store partially in both Workers 1 and 2 nodes. The process is similar to the process of prefix formation but the way of forming the new key is different. In prefix formation, we know that the key a can generate new keys of ab

and *ae* as *a* is the prefix of *ab* and *ae*. The record of *e* is removed in the server as *e* is no any prefix of level 2 candidate keys. However, in suffix formation, *a* is removed from the dataset as *a* is not any suffix of level 2 candidate keys. The key *e* generates two new keys of *ae* and *be*. As the result, ⟨e, (100, 100, 100, 100, 100, 0)⟩ becomes ⟨ae, (100, 100, 100, 100, 100, 0)⟩ and ⟨be, (100, 100, 100, 100, 100, 0)⟩.

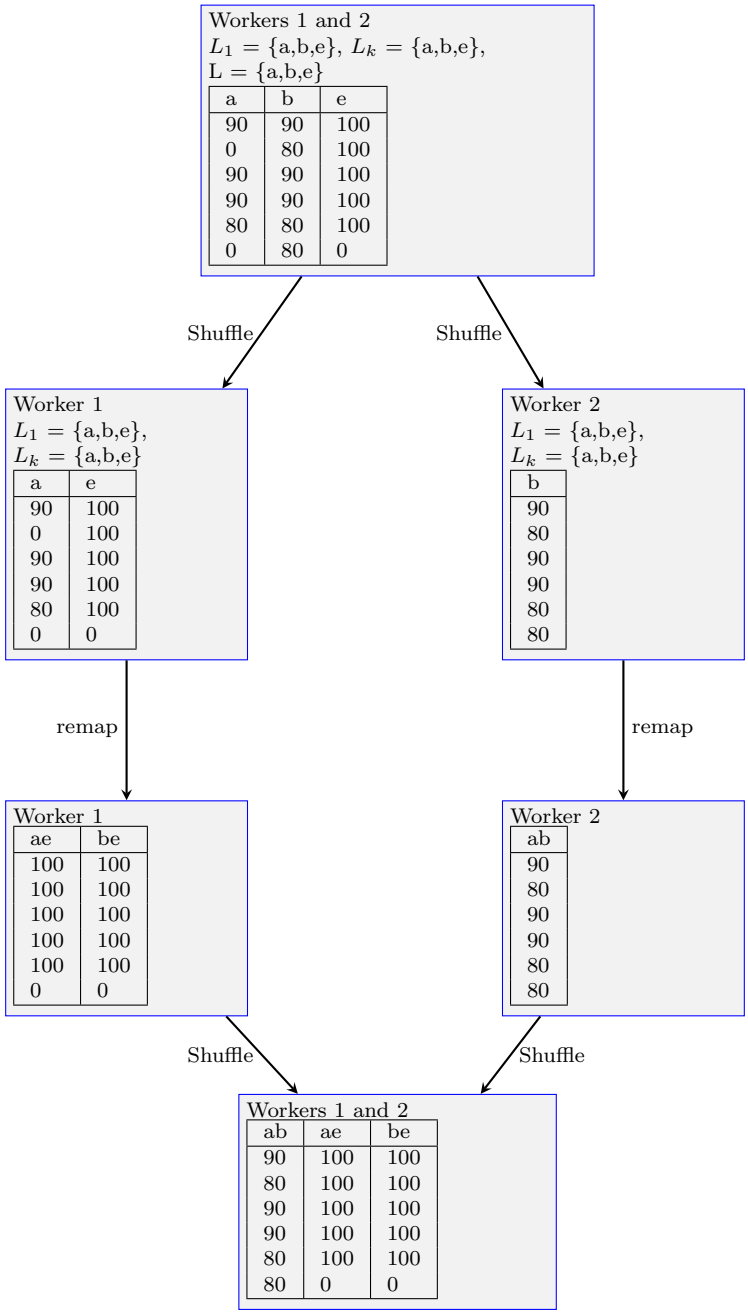


Figure 4.3: MR-U-VIPER:Suffix Formation at Level 2

Now we have new candidates of the prefix dataset and suffix dataset that we need to combine to form the next candidate dataset. This is through a union operation. The reduce operation is to make a unique dataset. For example, for the key *ae*, we

have two mapped records. They are ⟨ae, (90, 0, 90, 90, 80, 0)⟩ and ⟨ae, (100, 100, 100, 100, 100, 0)⟩. As in the input of reduce phase, these become ⟨ae, [(90, 0, 90, 90, 80, 0), (100, 100, 100, 100, 100, 0)]⟩. To calculate the real vector for the key *ae*, it basically calculates the product of the two vectors. Hence, it becomes ⟨ae, (90*100*0.01, 0*100*0.01, 90*100*0.01, 90*100*0.01, 80*100*0.01, 0*0*0.01)⟩ = ⟨ae, (90, 0, 90, 90, 80, 0)⟩. The frequency count operation leads to the result L_k={ab, ae, be}. The result of union and reduce operations becomes the prefix dataset. This algorithm continue until the level 3, in which case we get only one itemset, *abe*.

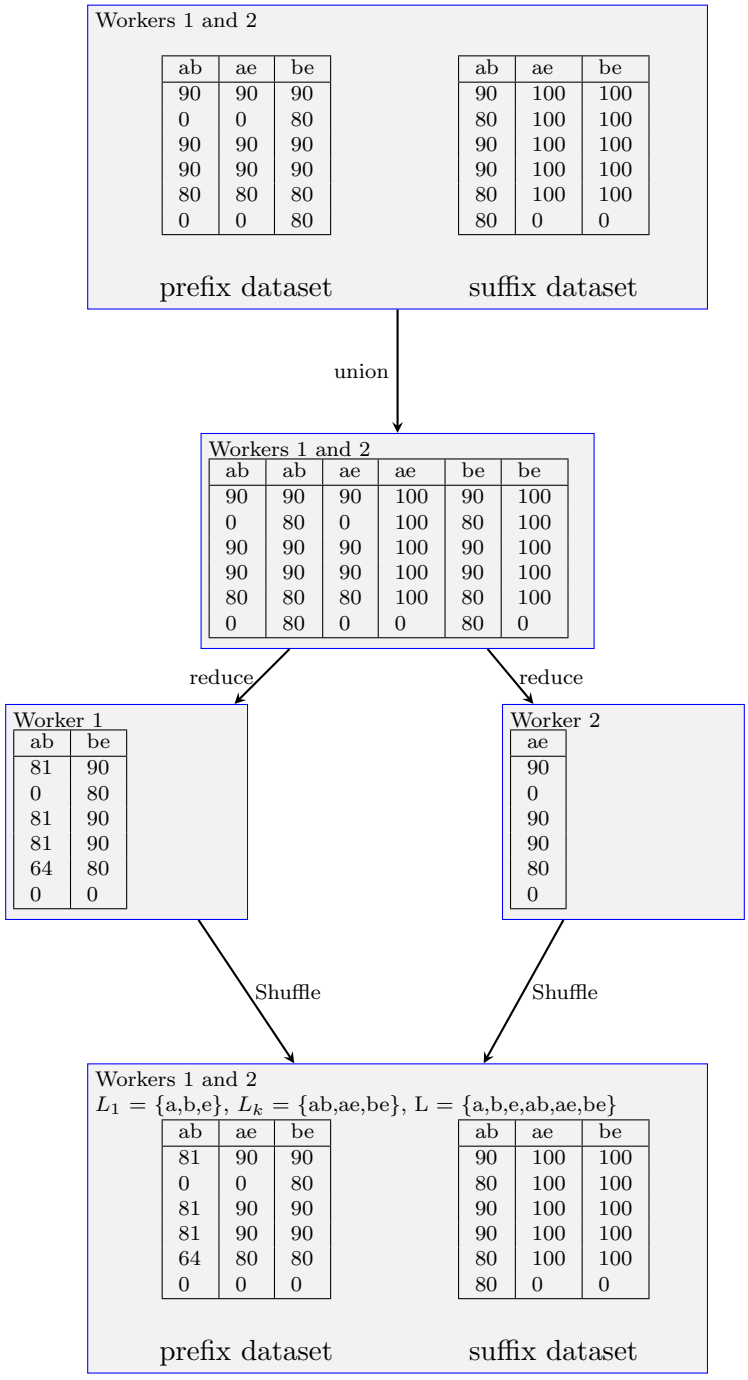


Figure 4.4: MR-U-VIPER: Union and Reduce phases at Level 2

4.3.6 Summary

In this chapter, we proposed the MR-U-VIPER algorithm that is adopted from U-VIPER. To better explain the algorithm, we identified behaviors that can use MapReduce approaches using U-VIPER. We observed that the bit-vector operations can be converted to MapReduce behaviors. With in-memory saving technique, all supported values are converted to 8-bit integers. The actual algorithm creates prefix mappings and suffix mappings. It then takes the union of the two mapping datasets to form one set of mapped data. In reduce phase, mapped data with the same keys are grouped and summed up the value in each position to form a new bit-vector. In the other reduce phase, each vector is summed up the value in each position. The final value becomes the support. A detailed example is illustrated the algorithm.

Chapter 5

Evaluation

In this chapter, we evaluate our proposed algorithms analytically and experimentally. For analytical evaluation, we analyzed the memory usage for all algorithms we proposed and two more algorithms. We compared two other map-reduce Apriori-based algorithms. These two algorithms are MR-U-Apriori-org and MR-U-Apriori-impr. They are both the map-reduce version of U-Apriori. These two algorithms are briefly explained in next few sections.

For experimental evaluation, we compare these four algorithms for run time with different sized of data and a different minimum threshold. In addition, we also compare performances of our proposed algorithms with U-VIPER and UV-Eclat.

5.1 MR-U-Apriori-org and MR-U-Apriori-impr

In order to compare our proposed algorithms, we show U-Apriori converted to Map-reduce versions. For evaluations, we show two map-reduce versions of U-Apriori. They are MR-U-Apriori-org and MR-U-Apriori-impr.

5.1.1 MR-U-Apriori-org

The original U-Apriori utilize a candidates generate and test approach to find frequent itemsets from uncertain data. It bases on the current level of frequent itemsets and generates the next level of candidates. It then reads through each transaction and generate the local set of candidates. Only candidates that match the global sets of candidates are kept from the testing stage. It then reads the database again and return only frequent candidates. For converting this algorithm to map-reduce algorithm, each slave server grasps a piece of the original data. The master data generates the global candidates and then deliver to each slave server. Each worker node generates the local candidates from the transactions and then finds matches to global candidates. The matched candidates are mapped into a format of ⟨itemset, value⟩. In the reduce phase, mapped records with the same keys are collected and then added up to one value.

5.1.2 MR-U-Apriori-impr

MR-U-Apriori-org utilizes the current level of frequent itemsets to generate next level of candidates. This is a very expensive process as not only it is very time consuming to generate global candidates but also it takes much memory to store and pass to each worker server. It is simple enough to generate local candidates when reading each transaction and then have the additional check by having the pruning step to filter itemsets that are not frequent in the current level.

5.2 Analytical Evaluation

In this M.Sc. thesis, four algorithms are compared for evaluation. They are MR-U-Apriori-org, MR-U-Apriori-improv, MR-U-VIPER and MR-UV-Eclat. As all these algorithms execute in Spark, we need to explain some important aspects in Spark. Spark mainly handles big data and enable in-memory computation. This explains why Spark is 10 times faster than Hadoop. Most of in-memory computations happen in map and reduce processes. Each worker server fetches data either from its local hard disk or from shuffles and then builds RDD into memory. If the RDD cannot fit into memory, it will store them in temporary hard disk storage and bring them to main memory again when room is available in main memory. In this memory usage evaluation, we need to look at the process on each algorithm that consumes the most memory usage. Specifically, we count how many nodes each algorithm creates for its peak time (the time that the algorithm consumes the most memory). We find out that the peak time for each algorithm is the mapping process on generating candidates. For these four algorithms, they generates candidates in lexicographic tree order. For this evaluation, we use the worst-case scenario approach to exam memory usage. We assume all transactions contain the total number of items and these items are all frequent. We specify t as the total number of items, l as the current level of candidates. Let us look at MR-U-Apriori-org. The algorithm needs to generate global candidates and local candidates. In our case, we assume each transaction has all items. Thus, there will be two sets of global candidates. If we want to generate candidates on l level, MR-U-Apriori-org reads each transaction and then generates the lexicographic tree until the l level. For instance, when reading transaction $\{a, b, c, d\}$ and we want to generate level 3 (l=3) candidates. In this case, we first need to generate level 2 candidates and then generate level 3 candidates based on level 2 candidates. Suppose

that t is the total number of frequent singletons, and l is the level of candidates to be generated. Total nodes MR-U-Apriori-org for mapping phase is:

$$\sum_{j=1}^{l}\sum_{i=1}^{t}(t-j+1-(i-1))\times 2 \tag{5.1}$$

For MR-U-Apriori-improv, it only generates local candidates, and thus total nodes MR-U-Apriori-improv is:

$$\sum_{j=1}^{l}\sum_{i=1}^{t}(t-j+1-(i-1)) \tag{5.2}$$

For MR-U-VIPER, it is based on the current level of itemsets to generate the next level itemset. In addition, each vector contains the number of total transactions, and we let this number be totalTrans. Hence, the total nodes of MR-U-VIPER is:

$$\sum_{i=1}^{t}(t-l+1-(i-1))\times totalTrans \tag{5.3}$$

For MR-UV-Eclat, it bases on the current level of itemsets to generate the next level candidates. Hence, the total nodes of MR-UV-Eclat is:

$$\sum_{i=1}^{t}(t-l+1-(i-1)) \tag{5.4}$$

From equations (5.1), (5.2), (5.3) and (5.4), MR-U-VIPER is observed to be very sensitive to the size of total transactions. As the data size increases, the amount of nodes the algorithm carried dramatically increases. MR-UV-Eclat has the least amount of nodes carried. In the average case, the number of local combinations is less than global combinations as the number of items is usually more than the global number of items. Hence, the memory consumption is even much less in MR-UV-Eclat

than MR-U-VIPER.

5.3 Experimental Evaluation

In this section, we discuss the experimental evaluation of our proposed algorithms. Still, we compare our proposed algorithms MR-U-VIPER and MR-UV-Eclat with the other two algorithms, which are MR-U-Apriori-org and MR-U-Ariori-impr. Also there are two other experiments. One is to compare performance between MR-U-VIPER and U-VIPER. The other is to compare performance between UV-Eclat. All mining algorithms will have the same mining results.

We use IBM synthetic dataset [AS94] to test these algorithms. Specifically, we test in different sizes of datasets and in different minimum support threshold. There are two kind of datasets: (80_90) and (10_100). These numbers represent the range of existential probability in the datasets. For instance, (80_90) all itemsets will have support value of 0.8 to 0.9. Similarly, (10_100) all itemsets will have support value of 0.1 to 1.0. The size of datasets is between 1000 transactions to 100k transactions. In addition, when doing experiments of comparing performances of MR-U-VIPER, U-VIPER, MR-UV-Eclat and U-Eclat, use mushroom data that have 8124 transactions and other set of real life data which have 1000556 transactions.

All of these experiments were run on Xeon 6 core 2.1 GHz machine with 20 GB of ram. All algorithms are executed in Linux virtual machines on a master-slave structure environment, with 1 master node and 2 worker nodes. The master node has 2 cores and 6 GB of rams. Each worker node has one core and 4 GB of ram. To test scalability, the setting of Spark is an in-memory computation. This means that if data structure cannot fit into main memory, the execution will be crashed.

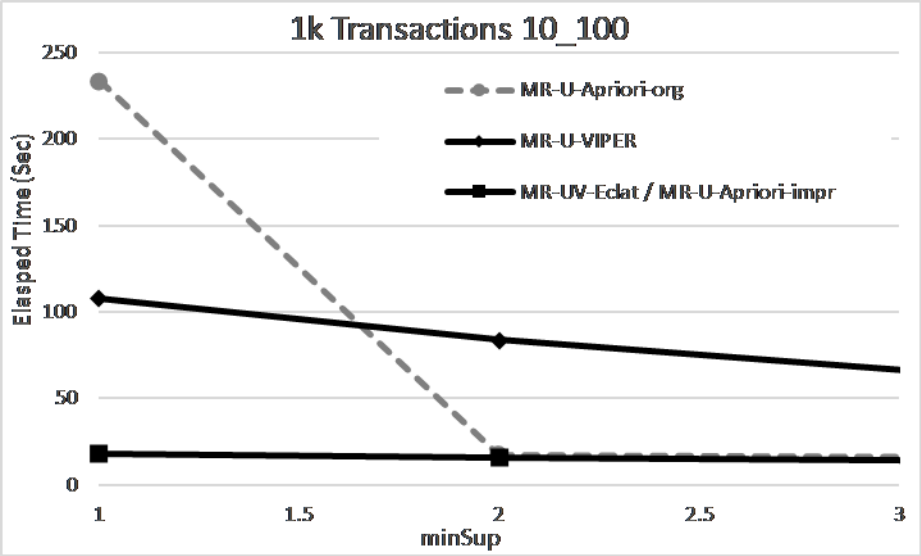


Figure 5.1: Experimental result on 1000 Transactions on 10_100 (Experiment 5.3.1)

Experiment 5.3.1. From this experiment, the test data is 1000 transactions and (10_100). The experiment result is shown in Figure 5.1. MR-U-VIPER is observed to have the better performance than MR-U-Apriori but has worse performance than MR-U-Apriori and MR-UV-Eclat when the minimum support threshold is really small. Since the data size is small, the amount of nodes MR-U-VIPER goes through are less than MR-U-Apriori-org. The amounts of nodes MR-U-Apriori and MR-UV-Eclat still are less than MR-U-VIPER and MR-U-Apriori-org. As the minimum support threshold increases, all algorithms carry fewer nodes. However, the performance of MR-U-VIPER is still dragged as it is affected by the number of transactions.

Experiment 5.3.2. For this experiment, MR-U-VIPER fails on the first few low minimum thresholds, hence, Figure 5.2 does not show the result for MR-U-VIPER. The data size is 10000 and is on (80_90). This data set is still sparse data. Only when the minimum support threshold is set to extremely low, the data set is considered to be dense. All these algorithms carry a significant amount of nodes when the data

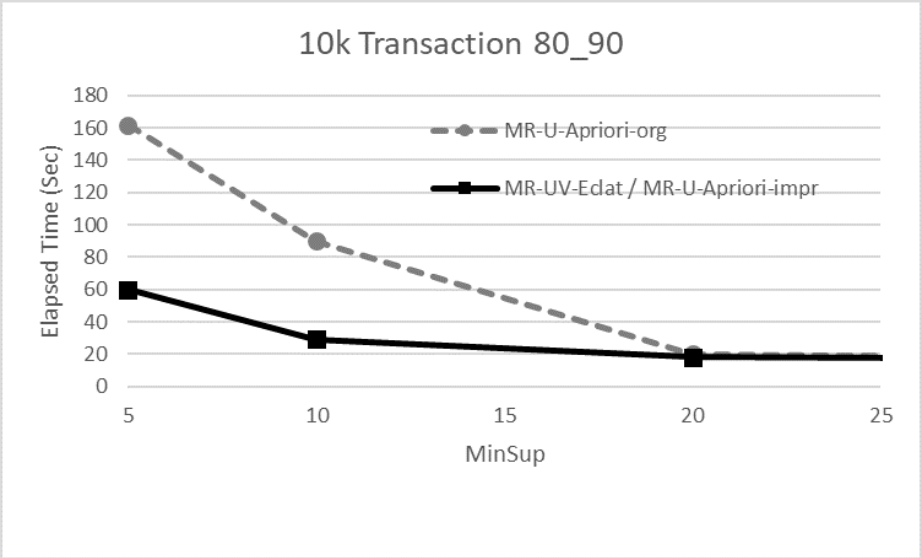


Figure 5.2: Experimental result on 10000 Transactions on 80_90 (Experiment 5.3.2)

set is dense. In this case, when the minimum support becomes 5, MR-UV-Eclat is observed to have the best performance. MR-U-Apriori has the worst when the minimum support threshold becomes really low. As the minimum support threshold increases, the difference in performance between all algorithms becomes less.

Experiment 5.3.3. As the data size becomes 100000, even though it is on (10_100). MR-U-Apriori-org still fails because data cannot fit into memory. Observed from Figure 5.3, the strength of MR-UV-Eclat becomes more obvious. If the minimum support threshold becomes less than 25, MR-UV-Eclat is observed to have much a better performance than MR-U-Apriori. This indicates that MR-UV-Eclat is fast and scalable when data sets are large in size and dense.

Experiment 5.3.4. This experiment is for testing the performance on each algorithm when running on data sets of different sizes. All data sets are on (80_90). The data size range is between 1000 and 10000. The minimum support threshold is set to 1%. Observed form Figure 5.4, MR-U-VIPER is sensitive to data size. As the data size

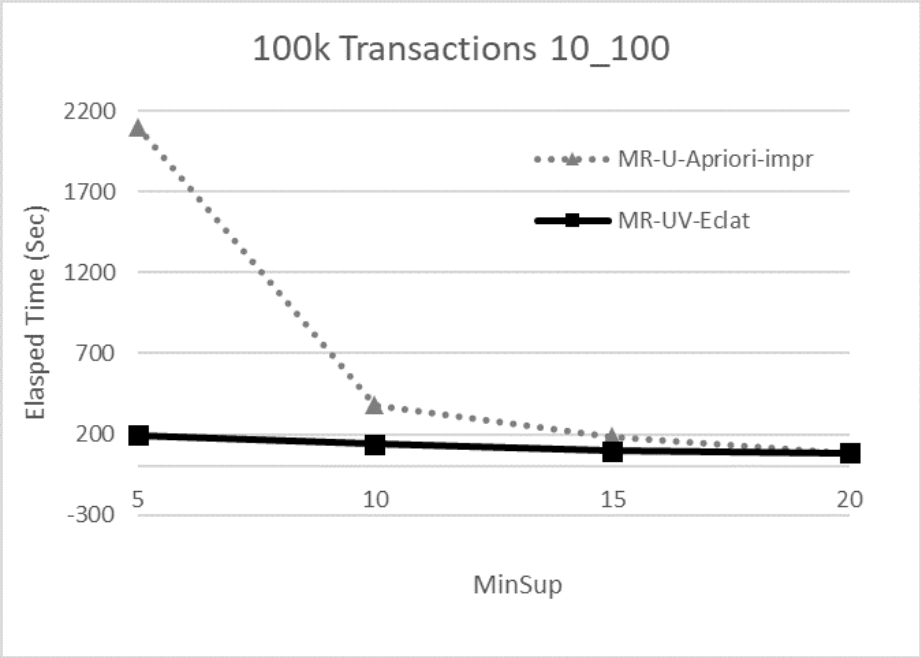


Figure 5.3: Experimental result on 100000 Transactions on 10_100 (Experiment 5.3.3)

goes up, performance of MR-U-VIPER decreases.

Experiment 5.3.5. The test data for this experiment are mushroom data (50_60). The data size is 8124 transactions. MR-U-VIPER outperforms U-VIPER two times most of cases. This is because U-VIPER is executed only by one machine and MR-U-VIPER is executed by 3 servers; (1 master server and 2 worker servers). As minSup increases, the difference of elapsed time is not two times. In MR-U-VIPER, data are needed to deliver to all worker servers for executions independently. This incurs communication time. When the data size is large, the communication time is less significant. However, when the data size is small, the communication time dominates the major execution time. Thus, the effect of running in multiple machines fade as data size becomes small. The execution time in U-VIPER is not as twice as the execution time in MR-U-VIPER.

Experiment 5.3.6. Kosarak data have the largest data size among all our test data.

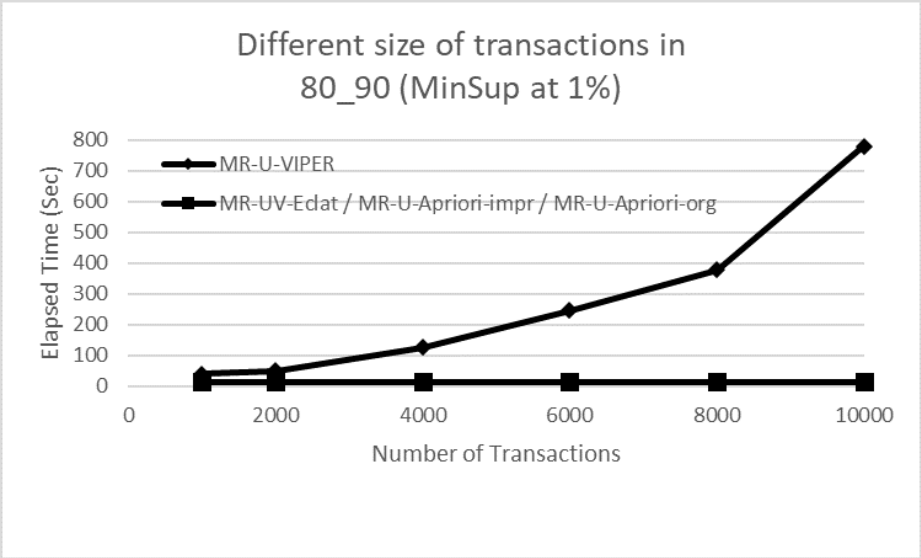


Figure 5.4: Experimental result on Different Size of Data on 80_90 (Experiment 5.3.4)

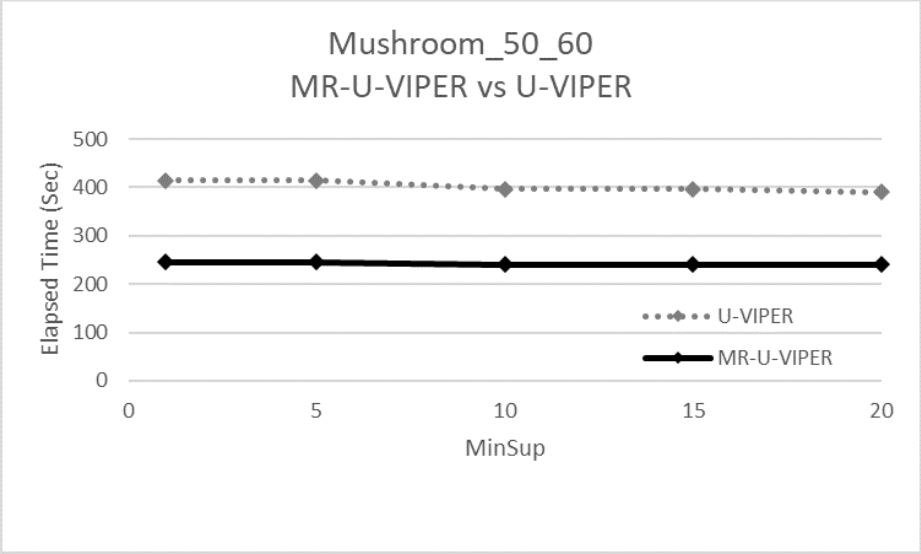


Figure 5.5: Experimental result on Mushroom Data in VIPER Algorithms (Experiment 5.3.5)

Compared with the last experiment, MR-UV-Eclat outperforms UV-Eclat in all cases. The performance of UV-Eclat is slow as the data size is very large, thus can no longer be efficient if the algorithm is executed by one machine.

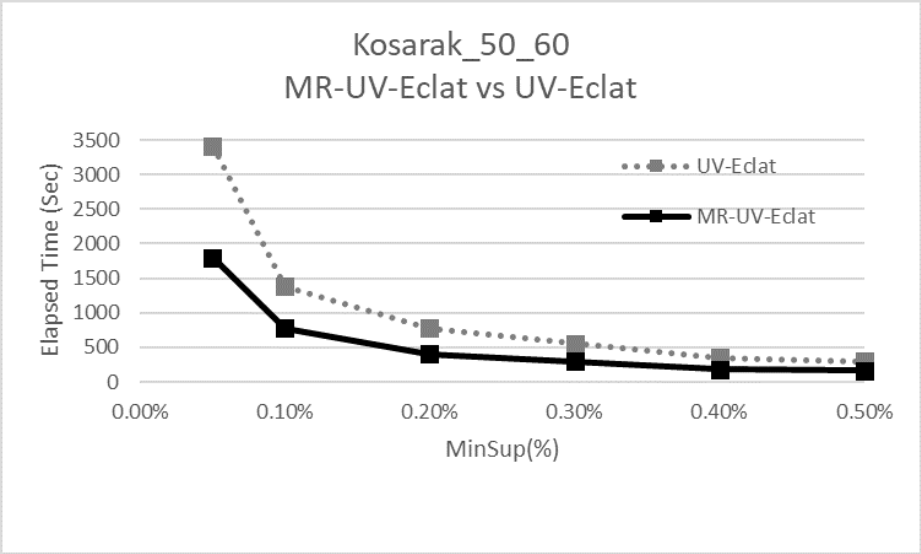


Figure 5.6: Experimental result on kosarak Data in Eclat Algorithms (Experiment 5.3.6)

For the experimental evaluation, we compared the performance of our proposed algorithms MR-U-VIPER and MR-UV-Eclat with two other map-reduce algorithms: MR-U-Apriori-org and MR-U-Apriori-impr. MR-U-VIPER is not scalable and fast as it has the most visits of nodes. MR-UV-Eclat is the fastest and most scalable as it consumes the least amount of memory. This is in line with our analytic evaluation.

5.4 Summary

In this chapter, we evaluated our proposed algorithms, analytically and experimentally. We compared them with two other algorithms; MR-Apriori-org and MR-Apriori-impr. In analytical evaluations, we evaluated memory consumption. We observed that the MR-U-VIPER algorithm is very sensitive to data size. As the size of data increased, the memory consumption on MR-U-VIPER drastically increased. MR-UV-Eclat consumed the least amount of memory among all algorithms. In ex-

perimental evaluation, we experimented all four algorithms. MR-U-VIPER, again, has the worst performance over all in the sense it took the longest runtime. The reason is large amount of memory it consumed. We also observed that MR-UV-Eclat has the best performance (i.e., took the shortest runtime) especially when the data were dense. Hence, we concluded that MR-UV-Eclat is a fast and scalable association mining algorithm that mines big uncertain data.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

As we are living in a world full of data but lacking of data relationships. By utilizing frequent pattern mining we can provide an important mean to find knowledge through data. Still, frequent pattern mining is a challenging task. First of all, many existing algorithms on frequent pattern mining can only handle precise data. Second of all, traditional technologies are not efficient in processing big data. Third of all, there still lacks for big uncertain data mining algorithms that are scalable and consume low memory. These motivate us to search big uncertain data mining algorithms that are fast and scalable.

In my M.Sc. thesis, we proposed MR-UV-Eclat algorithm. As MR-UV-Eclat is adopted from UV-Eclat, we still need to explain the logic of how to convert behaviors in UV-Eclat to MapReduce behaviors. We observed that two vectors can be intersected to become a new vector. One thing the two vectors have in common is that the final key is the combination of the two old keys. Hence, by replacing the olds key with the new key for both vectors in the map phase, they can be grouped in

the reduce phase. To increase the parallelism, each worker node only deals with one candidate in each transaction. Each mapped value is a projection tree.

We also adopted Spark technologies in U-VIPER to make MR-U-VIPER. We identified behaviors that can use MapReduce approaches in U-VIPER. We observed that the bit-vector operations can be converted to MapReduce behaviors. On memory saving technique, all support values are converted to 8-bit integers. In the actual algorithm, it creates prefix mappings and suffix mappings. Two mapping data points are union to become one mapping data. In reduce phase, each mapped data with the same keys are grouped and summed up each position to form a new bit-vector. In the other reduce phase, each vector is summed up the value in each position. The final value becomes the support.

We evaluated our proposed algorithms analytically and experimentally. We compared them with two other algorithms which are MR-Apriori-org and MR-Apriori-impr. In analytical evaluations, we evaluate memory consumption. We observed that the MR-U-VIPER algorithm is very sensitive to data size. As the size of data increases, the memory consumption on MR-U-VIPER drastically increases. MR-UV-Eclat consumes the least memory of all algorithms. In experimental evaluations, we experimented with all four algorithms in run time. MR-U-VIPER, again, has the worst performance over all. The reason is the large amount of memory it consumes. We also observed that MR-UV-Eclat has the best performance especially when the data is dense. Hence, MR-UV-Eclat is a fast and scalable association mining algorithm for mining big uncertain data.

Recall from Section 1.2, we asked four questions about uncertain data mining in big data. In this thesis, we provided answers to all of them:

1. *Are there any more scalable data structures that can efficiently store uncertain*

data? In MR-UV-Eclat, we propose using tree projection structure to store a transaction. It is very flexible to go through each level of candidates.

2. *Can we provide mining algorithms that can save memory?* In MR-UV-Eclat, each worker node only deal with one part of a tree projection structure, and the memory consumption is very low.

3. *Can we provide mining algorithms that efficiently handle big uncertain data?* As MR-UV-Eclat has low memory consumptions and high parallelism, it can efficiently handle big uncertain data.

4. *Can we provide mining algorithms that improve overall performances when dealing with big data?* Compared with MR-Apriori-org, MR-Apriori-impr and MR-U-VIPER, MR-UV-Eclat improves the overall performance.

In conclusion, we successfully developed the algorithm of MR-UV-Eclat. From both analytic evaluations and experimental evaluations, MR-UV-Eclat shows low memory consumptions and excellent run time. It can handle big uncertain data really well. It is a fast and scalable mining algorithm.

6.2 Future Work

For MR-UV-Eclat, the current solution reaches one level at a time from projection-tree structure. If we can find the maximum number of items in the transactions and compute the upper bound of the final level of frequent itemsets, we can compute the upper bound all these candidates at a level until the last level can be fit into memory. For example, we find out that the maximum number of items in a transaction for a dataset is 6. We can assume that the maximum frequent level is 6. If the current

level is 3, we can level the process to generate candidates of levels 4, 5 and 6. In this case, we can save 3 MapReduce cycles.

There are some improvements needed in MR-U-VIPER. First of all, in order to reduce the memory consumption pressure, we suggest sub-dividing the vector. For example, if the number of transactions is 10000, we can sub-divide the vector into two parts that with each having a size of 5000. In this case, these two parts can be executed in two different servers simultaneously. Second of all, we also find the mapped keys can consume large sections of memory. For example, if we have a mapped key that has 10 items in it, the key will require 10 integers in size. To further reduce the memory, we can assign a global ID for each mapped key. In this case, it only requires an integer.

Bibliography

[AAP01] Ramesh C Agarwal, Charu C Aggarwal, and VVV Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceeding of the ACM SIGMOD 1993*, pages 207–216. ACM, 1993.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th Int. Conf. Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[BCJ+17] Peter Braun, Alfredo Cuzzocrea, Fan Jiang, Carson Kai-Sang Leung, and Adam GM Pazdor. Mapreduce-based complex big data analytics over uncertain and imprecise social networks. In *Proceedings of the International Conference on Big Data Analytics and Knowledge Discovery 2017*, pages 130–145. Springer, 2017.

[BCL12] Bhavek P Budhia, Alfredo Cuzzocrea, and Carson Kai-Sang Leung. Vertical frequent pattern mining from uncertain data. In *Proceedings of the KES 2012*, pages 1273–1282, 2012.

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

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

[CL17] Deepankar Choudhery and Carson K Leung. Social media mining: prediction of box office revenue. In *Proceedings of the 21st International Database Engineering & Applications Symposium*, pages 20–29. ACM, 2017.

[FLS+18] Simon Fong, Jiaxue Li, Wei Song, Yifei Tian, Raymond K Wong, and Nilanjan Dey. Predicting unusual energy consumption events from smart home sensor network by data stream mining with misclassified recall. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–25, 2018.

[FPS96] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37, 1996.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceeding of the ACM SIGMOD 2000*, pages 1–12. ACM, 2000.

[KD13] Yun Sing Koh and Gillian Dobbie. Efficient single pass ordered incremental pattern mining. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems VIII*, pages 137–156. Springer, 2013.

[KMR+94] Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, Hannu Toivonen, and A Inkeri Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 401–407. ACM, 1994.

[LHM98] Bing Liu, Wynne Hsu, and Yiming Ma. Integrating classification and association rule mining. In *Proceeding of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD 1998)*, pages 80–86, New York, NY, August 27-31, 1998.

[LJPC18] Carson K Leung, Fan Jiang, Tik Wai Poon, and Paul-Émile Crevier. Big data analytics of social network data: who cares most about you on facebook? In *Highlighting the Importance of Big Data Management and Analysis for Various Applications*, pages 1–15. Springer, 2018.

[LLH12] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, page 76. ACM, 2012.

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

[LMJ14] Carson Kai-Sang Leung, Richard Kyle MacKinnon, and Fan Jiang. Reducing the search space for big data mining for interesting patterns from uncertain data. In *Proceedings of the 2014 IEEE International Congress on Big Data (BigData Congress)*, pages 315–322. IEEE, 2014.

[LMJ16] Carson Kai-Sang Leung, Richard Kyle MacKinnon, and Fan Jiang. Finding efficiencies in frequent pattern mining from big uncertain data. *World Wide Web Journal*, pages 1–24, 2016.

[LTBZ12] Carson Kai-Sang Leung, Syed K Tanbeer, Bhavek P Budhia, and Lauren C Zacharias. Mining probabilistic datasets vertically. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, pages 199–204. ACM, 2012.

[LZHS12] Ning Li, Li Zeng, Qing He, and Zhongzhi Shi. Parallel implementation of apriori algorithm based on mapreduce. In *Proceedings of the 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD 2012)*, pages 236–241. IEEE, 2012.

[NLHP98] Raymond T Ng, Laks VS Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the ACM SIGMOD 1998*, pages 13–24. ACM, 1998.

[PHL+01] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings of IEEE International Conference on Data Mining (ICDM 2001)*, pages 441–448. IEEE, 2001.

[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)*, pages 1664–1671. IEEE, 2014.

[Ros14] David Rossell. Big data and statistics: A statisticians perspective. *Mètode Science Studies Journal-Annual Review*, 5, 2014.

[SCCC10] Liwen Sun, Reynold Cheng, David W Cheung, and Jiefeng Cheng. Mining uncertain data with probabilistic guarantees. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 273–282. ACM, 2010.

[SHS+00] Pradeep Shenoy, Jayant R Haritsa, S Sudarshan, Gaurav Bhalotia, Mayank Bawa, and Devavrat Shah. Turbo-charging vertical mining of large databases. In *Proceedings of the ACM SIGMOD 2000*, pages 22–33. ACM, 2000.

[SSA+16] Md Badi-Uz-Zaman Shajib, Md Samiullah, Chowdhury Farhan Ahmed, Carson K Leung, and Adam GM Pazdor. An efficient approach for mining frequent patterns over uncertain data streams. In *Proceedings of the IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI 2016)*, pages 980–984. IEEE, 2016.

[SYZ+18] Sagar Samtani, Shuo Yu, Hongyi Zhu, Mark Patton, John Matherly, and HsinChun Chen. Identifying supervisory control and data acquisition (SCADA) devices and their vulnerabilities on the internet of things (IoT): A text mining approach. *IEEE Intelligent Systems*, 2018.

[YLF10] Xin Yue Yang, Zhen Liu, and Yan Fu. Mapreduce as a programming model for association rules algorithm on hadoop. In *Proceedings of the 3rd International Conference on Information Sciences and Interaction Sciences (ICIS 2010)*, pages 99–102. IEEE, 2010.

[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 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[ZED+12] Paul C Zikopoulos, Chris Eaton, Dirk DeRoos, Thomas Deutsch, and George Lapis. Understanding big data. *New York: McGraw-Hill*, 5(8), 2012.

[ZPOL97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In *Proceeding of the KDD 1997*, pages 283–286, 1997.