ENHANCING THE PERFORMANCE OF FREQUENT PATTERN MINING ALGORITHMS USING VERTICAL DATA FORMAT

Thesis submitted to the Bharathidasan University, Tiruchirappalli in partial fulfillment of the requirements for the award of the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

Submitted By

Ms.P.SUMATHI

Ref. No. 40007/Ph.D.K3/Computer Science/Part Time/January 2017

Under the Guidance of

Dr.S.MURUGAN

Associate Professor (Retd.) & Research Supervisor

Dr.V.UMADEVI

Assistant Professor & Research Co-supervisor

POST GRADUATE AND RESEARCH DEPARTMENT OF COMPUTER SCIENCE

NEHRU MEMORIAL COLLEGE (Autonomous)

(Nationally Accredited with 'A+' Grade by NAAC)

(Affiliated to Bharathidasan University)

PUTHANAMPATTI - 621 007

TIRUCHIRAPPALLI - Dt., TAMIL NADU, INDIA

MARCH 2022

Dr.S.MURUGAN
Associate Professor (Retd.) & Research Supervisor
PG & Research Department of Computer Science
Nehru Memorial College (Autonomous)
Puthanampatti - 621 007
Tiruchirappalli - Dt., Tamil Nadu

CERTIFICATE

This is to certify that the thesis entitled "**ENHANCING THE PERFORMANCE OF FREQUENT PATTERN MINING ALGORITHMS USING VERTICAL DATA FORMAT**" submitted by **Ms.P.Sumathi**, Research Scholar, PG & Research Department of Computer Science, Nehru Memorial College (Autonomous), Puthanampatti - 621 007, for the award of the degree of **Doctor of Philosophy in Computer Science**, is a record of original work carried out by her under my supervision and guidance. The thesis has fulfilled all requirements as per the regulations of the University and in my opinion, the thesis has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

(Dr.V.UMADEVI)
Research Co-supervisor

(Dr.S.MURUGAN)
Research Supervisor

Date :

Place : Puthanampatti

Mobile: 9486631199, 8825768771, E-Mail ID:murugan_nmc@hotmail.com

P.SUMATHI
Part-Time Research Scholar
PG & Research Department of Computer Science
Nehru Memorial College (Autonomous)
Puthanampatti - 621 007
Tiruchirappalli - Dt.
Tamil Nadu

DECLARATION

I hereby declare that the work embodied in this thesis entitled "**ENHANCING THE PERFORMANCE OF FREQUENT PATTERN MINING ALGORITHMS USING VERTICAL DATA FORMAT**" is a research work done by me under the supervision and guidance of **Dr.S.MURUGAN**, Associate Professor (Retd.) and the co-supervision of **Dr.V.UMADEVI,** Assistant Professor, PG & Research Department of Computer Science, Nehru Memorial College (Autonomous), Puthanampatti - 621 007. The thesis or any part thereof has not formed the basis for the award of any Degree, Diploma, Fellowship or any other similar titles.

Date :

Place : Puthanampatti **P.SUMATHI**

PG AND RESEARCH DEPARTMENT OF COMPUTER SCIENCE
NEHRU MEMORIAL COLLEGE (AUTONOMOUS)
(*Nationally Accredited with 'A+' Grade by NAAC*)
(Affiliated to Bharathidasan University)
PUTHANAMPATTI - 621 007
TIRUCHIRAPPALLI - Dt., TAMIL NADU, INDIA

CERTIFICATE OF PLAGIARISM CHECK

Name of the Research Scholar	**Ms. P.SUMATHI**
Course of Study	Ph.D., in Computer Science
Title of the Thesis/Dissertation	**ENHANCING THE PERFORMANCE OF FREQUENT PATTERN MINING ALGORITHMS USING VERTICAL DATA FORMAT**
Name of the Research Supervisor	**Dr.S.MURUGAN** Associate Professor (Retd.)
Name of the Research Co-supervisor	**Dr.V.UMADEVI** Assistant Professor
Department/Institution/Research Centre	PG and Research Department of Computer Science Nehru Memorial College (Autonomous) Puthanampatti - 621 007
Acceptable Maximum Limit	10%
Percentage of Similarity of Content Identified	1%
Software Used	**URKUND**
Date of Verification	15-03-2022

Report on plagiarism check, the item with % of similarity is attached.

Signature of the Research Co-supervisor (Dr.V.Umadevi)	**Signature of the Research Supervisor (Dr.S.Murugan)**	**Signature of the Candidate (Ms.P.Sumathi)**

Curiginal

Document Information

Analyzed document	40007-Ph.D.K3-Computer Science-Part Time-January 2017-P.SUMATHI - Dr.S.MURUGAN.doc (D130438651)
Submitted	2022-03-15T12:11:00.0000000
Submitted by	Srinivasa ragavan S
Submitter email	bdulib@gmail.com
Similarity	1%
Analysis address	bdulib.bdu@analysis.urkund.com

Sources included in the report

W	URL: https://www.gauthmath.com/solution/Given-A-1-3-6-8-9-12-15-and-B-6-9-12-which-is-TRUE-A-B-is-the-complement-of-A-B--1703145423666181 Fetched: 2021-07-07T11:05:38.4570000	4
W	URL: https://www.doubtnut.com/pcmb-questions/the-mean-of-1-3-4-5-7-4-is-m-the-numbers-3-2-2-4-3-3-p-have-mean-m-1-and-median-q-then-p-q-a-4-b-5-c-110227 Fetched: 2021-05-04T05:30:20.2730000	8
W	URL: https://link.springer.com/chapter/10.1007/978-3-319-07821-2_2 Fetched: 2022-03-15T12:11:00.0000000	1

ACKNOWLEDGEMENT

First and foremost, I would like to place on my sincere devotion to **Lord Almighty** for his countless blessings in completing the thesis successfully without any hurdles.

The profound gratitude deep from the heart is due to my guide and research supervisor **Dr.S.Murugan**, Associate Professor(Retd.), PG & Research Department of Computer Science, Nehru Memorial College, Puthanampatti for his periodical monitoring, motivation, intellectual guidance, meticulous way of correcting the thesis, untiring effort and interest shown in doing real scientific research.

I wish to express my gratitude to my research co-supervisor **Dr.V.Umadevi**, Assistant Professor, PG & Research Department of Computer Science, Nehru Memorial College, Puthanampatti for her encouragement and support rendered for my research work.

It is my bound duty to record my sincere thanks to the benevolent management of Nehru Memorial College, President **Mr.Pon.Balasubramanian**, and Secretary **Mr.Pon.Ravichandran**, for permitting me to pursue the research work in this prestigious institution and also all the facilities rendered to carry out the research in a meticulous way.

I wish to place on record my sincere thanks to **Mr.J.Rajendra Prasad**, Correspondent of Vysya College, Salem for his constant support and encouragement for my career and research.

I acknowledge with gratitude my sincere thanks to the doctoral committee members **Dr.K.Mani**, Associate Professor, PG & Research Department of Computer Science, Nehru Memorial College, Puthanampatti and **Dr.J.G.R.SATHIASEELAN**, Associate Professor & Head, Department of Computer Science, Bishop Heber

College, Tiruchirappalli for their invaluable suggestions, guidelines and healthy discussions made during doctoral committee meetings.

I would like to express my sincere thanks to **Dr.A.R.Ponperiasamy**, Principal of Nehru Memorial College, Puthanampatti and **Dr.P.Venkatesan**, Principal of Vysya College, Salem for their precious support in carrying out the research work in an effective manner.

I am indebted to thank **Dr.M.Muralidharan**, Associate Professor & Head and all eminent **faculty members** of PG & Research Department of Computer Science for their silent support, suggestions and encouragement throughout the journey of my research.

I wish to express my gratitude to all my **fellow researchers** and my dear **friends** for their timely help, suggestion and encouragement.

I am grateful to my beloved parents **Mr.C.Parasuraman** and **Mrs.P.Mahalakshmi** for their unconditional love, support, dedication and many efforts made for my life.

I would like to record my joyful thanks to my siblings **Mr.P.Mohan**, **M.Pharm., (Ph.D.)**, Associate Professor, Faculty of Pharmacy, Dr.M.G.R Educational and Research Institute, Chennai and **Dr.P.Kalpana**, Associate Professor, PG & Research Department of Computer Science, Nehru Memorial College, Puthanampatti for their guidance, constant encouragement and co-operation in all walks of my life.

Finally, I thank all the good hearts who helped me directly as well as indirectly during the journey of my research.

P.SUMATHI

ABSTRACT

Frequent patterns are patterns/itemsets, subsequences, or substructures that appear frequently in a dataset with not less than a user-specified threshold. Researchers realized that Frequent Pattern Mining (FPM) is vital in mining associations, correlations and other relationships among data. In the modern digital world, online shopping/e-shopping has become popular and mandatory in human lives. E-stores like Amazon show up the "*Frequently Bought Together*" and "*Customers who bought this item also bought"* for their customers to promote their sales and thereby obtains profits considerably. Many transactional data were collected every day, and finding frequent itemsets from the massive dataset is an issue for the researchers because it requires more processing time and memory. However, there are more efficient and scalable FPM algorithms found in the literature and also FPM has a wide range of applications there is always a need for better algorithms to minimize the issues. Thus, the research work focuses on developing efficient algorithms for FPM.

The research work aims to create time and memory-efficient models for discovering frequent patterns from transactional databases. For that, a framework named "SUMsFPM" has been proposed comprising of four research models viz., RISOTTO, JAB-VDF, TP-NPF-VDF and GNVDF. The RISOTTO has been proposed to reduce the runtime and JAB-VDF to minimize memory usage in finding the frequent patterns from large databases. The models namely TP-NPF-VDF and GNVDF have been contributed to reducing both time and memory.

The RISOTTO algorithm improves the performance of Apriori by combining both prefixed-itemset based storage structure and Vertical Data Format (VDF) and it is abbreviated by taking the uppercase letters from the phrase "pRefixed ItemSet

stOrage verTical daTa fOrmat". The method first finds the candidate 1-itemsets(C_1) as in classical Apriori and transforms them into VDF, then the frequent 1-itemset (L_1) is constructed from C_1 by removing the items whose $SC < \delta$ (user-specified threshold). After that, the L_1 is stored in the prefixed-itemset storage as *prefix-key* and *values*. It is noted that, in RISOTTO, the *values* with a single item is not stored in prefixed-itemset storage as it does not generate successive candidates. During the successive iterations, the items in *values* are used for joining and items that satisfy the Apriori property are combined with the *prefix-keys* for generating the candidate (i+1)-itemsets, followed by frequent(i+1)-itemsets and the process is repeated until no more candidate itemsets found. As this method uses VDF, the *SC* for the (i+1)-candidate itemsets were determined using the set intersection method which avoids repeated database scans.

The VDF format avoids repeated scans of transactional databases for determining the *SC* and limits the database scan to one but it requires huge memory for storing *TID*s of each item. To minimize the memory, the JAB-VDF model has been introduced. It uses a jagged array structure for storing the *TID*s, which allocates memory space exactly needed for the itemsets than the 2-D array.

The TP-NPF-VDF algorithm has been introduced as an enhancement version to VDF by incorporating a novel pattern generation method with multithreads. It also uses the jagged array for storing itemsets. It mainly consists of four phases. The first and second phase converts the transactional database into VDF and determines the frequent 1-itemset as in Apriori. The third phase rearranges the frequent 1-itemset in ascending order based on *SC*. The fourth phase creates n-1 threads one for each itemset in a frequent 1-itemset except for the last one. Each thread runs in parallel and determines from frequent 2-itemsets to k-itemsets until it is non-empty, for each itemset in frequent 1-itemset, where $k \geq 2$ with a novel way of generating patterns.

Though the multithreads reduces the runtime, it is well-known that the GPU acceleration will enable the execution speed with multiple cores. By considering this, GNVDF, a GPU-accelerated novel algorithm for finding frequent patterns using the VDF approach with a jagged array has been introduced. Finding frequent 1-itemset remain the same as TP-NPF-VDF but it removes the null transactions initially. The common transactions in L_1 (C_{TID_list}) are identified, removed from L_1 and updated the new *min_sup* as $\delta_{new} = \delta - n$. The frequent 1-itemsets are split into two logical buckets LB_1 and LB_2 based on δ_{new}. The candidate 2-itemsets patterns are generated by combining each item I_x in LB_1 with each item I_y in LB_2 and each item I_z with I_{z+1} until the last item in LB_2. The itemset combination that ends with the last item in LB_2 will be placed in C_{2_2} and the rest in C_{2_1}. From C_{2_1} and $C_{2\_2,}$ L_{2_1} and L_{2_2} were generated based on *SC*. For generating candidate 3-itemset, each itemset I_x in L_{2_1} is combined with the next item I_y in LB_2 after the last item in I_x and placed in C_{3_2} and C_{3_1} as previous. The L_{3_1} and L_{3_2} were formed by removing the infrequent itemsets in C_{3_1} and C_{3_2}. The process is repeated until no more candidates in L_{n_1}.

All the proposed algorithms were implemented using Python and tested with both real-time and synthetic types obtained from the FIMI repository and an open-source data mining library and measured the runtime and memory usage. It is proved from the experiments that the proposed models will reduce the runtime and memory usage significantly than the existing ones.

LIST OF PUBLICATIONS

International Journals

1. **P.Sumathi**, Dr.S.Murugan, Dr.V.Umadevi, "A Multithread, Novel Pattern Based Algorithm for Finding Frequent Patterns With Jagged Array and Vertical Data Format", Indian Journal of Computer Science and Engineering (IJCSE), e-ISSN : 0976-5166, p-ISSN : 2231-3850, Vol. 12, No. 5, pp.1353-1363, Sep-Oct 2021. DOI:10.21817/indjcse/2021/v12i5/211205078 (**UGC Care List - II, Scopus Indexed**).
2. **P. Sumathi**, S.Murugan, "GNVDF: A GPU-accelerated Novel Algorithm for Finding Frequent Patterns Using Vertical Data Format Approach and Jagged Array", International Journal of Modern Education and Computer Science (IJMECS), ISSN: 2075-0161 (Print), ISSN: 2075-017X (Online), Vol.13, No.4, pp.28-41, August 2021. DOI: 10.5815/ijmecs.2021.04.03 (**UGC Care List - II, Scopus Indexed**).
3. **P.Sumathi**, S.Murugan, "A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach", International Journal of Computer Sciences and Engineering, E-ISSN: 2347-2693, Vol. 6, No. 11, pp.152-157, December 2018 (**UGC Approved Journal**).
4. **P.Sumathi**, S.Murugan, "RISOTTO - A Novel Hybrid Approach for Enhancing Classical Apriori Algorithm", International Journal of Scientific Research in Computer Science Applications and Management Studies, ISSN: 2319-1953, Vol. 7, No. 5, September 2018 (**UGC Approved Journal**).

International Conference

1. **P.Sumathi**, S.Murugan, "A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach", Proceedings of International Conference on "Blooming Trends in Tech Challenges and Opportunities", National College (Autonomous), Tiruchirappalli from 27.09.2018 to 29.09.2018.

National Conference

1. **P.Sumathi**, S.Murugan, "A Survey on Mining Frequent Itemsets" in one-day National Level Conference (Multidisciplinary) on Emerging Trends in Digital Transformation - ETDT 2018, Government College for Women, Maddur, Karnataka - 571 428 on 30.07.2018. ISBN: 978-81-933447-3-6.

Seminars/Webinars/FDPs Attended

1. Completed an FDP on "**Data Mining**" conducted by NPTEL - AICTE during Feb - Apr 2021.
2. Attended an FDP on "**Applied Research in Multidisciplinary Studies**" organized by Sona College of Technology, Salem from 18.05.2020 to 19.05.2020.
3. Participated in the webinar on "**How to use Turnitin Software for your Research**" organized by Guru Nanak Institute of Management Studies, University of Mumbai held on 04.05.2020.
4. Participated in the webinar on "**International Patent Filing Process**" organized by Sri Krishna College of Technology, Coimbatore held on 01.05.2020.

5. Participated in a webinar on "**Enhancing Research Effectiveness using Scopus, ScienceDirect and Mendeley**" organized by Kurukshetra University, Kurukshetra in Collaboration with Elsevier held on 01.05.2020.
6. Participated in a webinar on "**An Effective Research Paper Writing Skills**", organized by Bhagwan Mahavir College of Commerce & Management Studies, Gujarat from 13.04.2020 to 16.04.2020.
7. Participated in a one-day International seminar on **"Research Intelligence and Database (RID-2018)**", organized by Periyar University, Salem on 14.12.2018.

CONTENTS

	Page No.
Acknowledgement……...	i
Abstract……	iii
List of Publications/Conferences/Seminars/Webinars/FDPs ……..…	vi
Table of Contents……	ix
List of Figures……..	xiii
List of Tables……...	xv
List of Algorithms and Procedures…….....	xviii
List of Abbreviations……...	xix
List of Symbols……	xxii
CHAPTER - 1 INTRODUCTION……...	**1-22**
1.1 Background……	1
1.2 Data Mining……...	1
1.2.1 Knowledge Discovery in Databases…….	2
1.3 Frequent Pattern Mining……...	3
1.3.1 Terminologies in FPM……..	4
1.3.2 Basic Definitions……..	6
1.3.3 Architecture/Layout of Storing Transactional Data…….	7
1.4 Association Rule Mining……..	8
1.4.1 Apriori Algorithm…….	10
1.4.1.1 Disadvantages of Apriori Algorithm……...	11
1.4.2 FP-Growth Algorithm……..	12
1.4.2.1 Advantages of FP-Growth Algorithm……..	12
1.4.2.2 Disadvantages of FP-Growth Algorithm…….	13
1.4.3 Eclat Algorithm……	13
1.4.3.1 Advantages & Disadvantages of Eclat Algorithm……	13
1.5 Applications of FPM……..	14
1.6 Scope of the Research Work……..	15
1.7 Aim & Objectives of the Research Work……...	16
1.8 Problem Statement……..	17
1.9 Problem Description……...	17

1.10 Description of the Datasets……... 19
1.11 Chapter Organization…… 20

CHAPTER - 2 REVIEW OF LITERATURE…….. 23-44
2.1 Background…… 23
2.2 Works Related to Apriori and FP-Growth……. 24
2.3 Works Related to Matrix-based Apriori…… 27
2.4 Works Related to Vertical Data Format…… 36
2.5 Works Related to Eclat…….. 38
2.6 Works Related to GPUs……. 39
2.7 Observations and Limitations of the Existing Literature……... 42

CHAPTER - 3 RISOTTO: A NOVEL HYBRID APPROACH FOR ENHANCING CLASSICAL APRIORI ALGORITHM……... 45-66
3.1 Background…… 45
3.2 Prefix-Itemset Storage Structure……. 46
3.3 Vertical Data Format……. 46
3.4 Proposed Methodology…….. 47
3.4.1 Illustration by an Example…… 51
3.5 Experimental Results and Discussion…… 59
3.5.1 Welch's Two Sample *t*-test……..... 63
3.6 Chapter Summary…….. 65

CHAPTER - 4 JAB-VDF: A JAGGED ARRAY BASED DATA STRUCTURE FOR VERTICAL DATA FORMAT…….. 67-82
4.1 Background…… 67
4.2 Jagged Array …… 68
4.3 Proposed Methodology…….. 69
4.3.1 Illustration by an Example…… 69
4.4 Experimental Results and Discussion…… 79
4.5 Chapter Summary…….. 80

CHAPTER - 5 TB-NPF-VDF: A MULTITHREADED, NOVEL PATTERN FORMATION FOR VERTICAL DATA FORMAT WITH JAGGED ARRAY ……..... 83-100

5.1 Background…… 83

5.2 Multithreading……... 84

5.3 Proposed Methodology…….. 86

5.3.1 Illustration by an Example…… 89

5.4 Experimental Results and Discussion…… 95

5.4.1 Welch's Two Sample *t*-test …….. 95

5.5 Chapter Summary…….. 99

CHAPTER - 6 GNVDF: A GPU-ACCELERATED NOVEL ALGORITHM USING VERTICAL DATA FORMAT AND JAGGED ARRAY……..... 101-122

6.1 Background…… 101

6.2 Graphical Processing Unit……. 102

6.2.1 Processing flow of CUDA…….. 103

6.3 Proposed Methodology…….. 104

6.3.1 Memory Requirement Calculation……. 109

6.3.2 Illustration by an Example…….. 111

6.4 Experimental Results and Discussion…… 117

6.5 Chapter Summary…….. 121

CHAPTER - 7 CONCLUSION……..……. 123-128

7.1 Summary of the Contributions…….. 123

7.2 Limitations and Future Research Directions…… 126

7.3 Endnote…….. 127

REFERENCES…….. 129-142

APPENDICES

Appendix - A

Google Scholar Image Showing the Research Scholar Publications

Papers Included in International Digital Libraries

Appendix - B

Papers Published in the International Journals

LIST OF FIGURES

Figure No.	Title	Page No.
1.1	KDD Process	4
1.2	Classification of Frequent Pattern Mining Algorithm	9
1.3	An Example for Apriori Algorithm	12
1.4	An Example of Eclat Algorithm	14
1.5	Workflow of the Research	19
3.1	Workflow of RISOTTO	50
3.2	Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for chess Dataset	61
3.3	Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for mushroom Dataset	61
3.4	Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for t25i10d10k Dataset	62
3.5	Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for c20d10k Dataset	62
4.1	Jagged Array Representations	68
4.2	Comparison of Memory Consumption (in GB) between JAB-VDF and VDF with δ=20%	80
5.1	Multithreading	85
5.2	Workflow of TB-NPF-VDF	89
5.3	Runtime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for chess Dataset	97
5.4	Runtime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for mushroom Dataset.	97
5.5	Runtime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for t25i10d10k Dataset	98
5.6	Runtime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for c20d10k Dataset	98

Figure No.	Title	Page No.
6.1	Processing Flow of CUDA	104
6.2	Workflow of GNVDF	109
6.3	Runtime Performance of GNVDF with and without GPU-acceleration for chess Dataset	119
6.4	Runtime Performance of GNVDF with and without GPU-acceleration for mushroom Dataset	119
6.5	Runtime Performance of GNVDF with and without GPU-acceleration for t25i10d10k Dataset	120
6.6	Runtime Performance of GNVDF with and without GPU-acceleration for c20d10k Dataset	120

LIST OF TABLES

Table No.	Title	Page No.
1.1	A Sample Transactional Dataset of a Grocery Store	5
1.2	Transactional Database D in HDF	8
1.3	VDF of D	8
1.4	Characteristics of Datasets	20
3.1	Prefixed-Itemset Storage Structure	46
3.2	Transactional Database D	51
3.3	Computation of C_1	52
3.4	Computation of L_1	52
3.5	Prefixed-Itemset Storage with frequent 1-itemset	52
3.6	Computation of C_2	53
3.7	Computation of L_2	54
3.8	The Original Prefixed-Itemset Storage after Appending frequent 2-itemset	54
3.9	The Prefixed-Itemset Storage after Appending frequent 2-itemset in RISOTTO	55
3.10	Computation of C_3	55
3.11	Computation of L_3	56
3.12	The Original Prefixed-Itemset Storage after Appending frequent 3-itemset	56
3.13	The Prefixed-Itemset Storage after Appending frequent 3-itemset in RISOTTO	57
3.14	Computation of C_4	57
3.15	Computation of L_4	57
3.16	The Original Prefixed-Itemset Storage after Appending frequent 4-itemset	58
3.17	The Prefixed-Itemset Storage after Appending frequent 4-itemset in RISOTTO	58
3.18	Performance Results of RISOTTO in seconds	60

Table No.	Title	Page No.
3.19	Results of *t*-test	64
4.1	Transactional Database D	70
4.2	D in VDF	70
4.3	Frequent 1-itemset in VDF	71
4.4	VDF of frequent 2-itemsets	73
4.5	VDF of frequent 3-itemsets	73
4.6	VDF of frequent 4-itemsets	73
4.7	Jagged Array Representation of frequent 1-itemset	75
4.8	Jagged Array Representation of frequent 2-itemset	76
4.9	Jagged Array Representation of frequent 3-itemset	77
4.10	Jagged Array Representation of frequent 4-itemsets	78
4.11	Comparison of Memory Consumption (in GB) between JAB-VDF and VDF with δ=20%	79
5.1	Transactional Database D	90
5.2	D in VDF	90
5.3	Candidate 1-itemset	91
5.4	Jagged Array Representation of frequent 1-itemset	91
5.5	Sorted frequent 1-itemset	92
5.6	Frequent 2-itemset for <a> by Thread-1	92
5.7	Frequent 3-itemsets for <a> by Thread-1	93
5.8	Frequent 4-itemsets for <a> by Thread-1	93
5.9	Frequent 2-itemset for <f> by Thread-2	93
5.10	Frequent 2-itemset for <i> by Thread-3	93
5.11	Frequent 3-itemset for <i> by Thread-3	93
5.12	Frequent 2-itemset for <d> by Thread-4	94
5.13	Frequent 3-itemset for <d> by Thread-4	94
5.14	Frequent 4-itemset for <d> by Thread-4	94
5.15	Frequent 2-itemset for <m> by Thread-5	94
5.16	Frequent 3-itemset for <m> by Thread-5	94
5.17	Frequent 2-itemset for <c> by Thread-6	94

Table No.	Title	Page No.
5.18	Frequent 3-itemset for <c> by Thread-6	94
5.19	Frequent 2-itemset for <e> by Thread-7	94
5.20	Details of Itemsets for D	95
5.21	Performance Results of TB-NPF-VDF in seconds	96
5.22	Results of *t*-test	99
6.1	Vertical Data Format of D	111
6.2	Candidate 1-itemset (C_1).	112
6.3	Frequent 1-itemset (L_1)	112
6.4	Common Transaction List (CTL)	113
6.5	Final frequent 1-itemset (L_1)	113
6.6	Logical Bucket-1(LB_1)	113
6.7	Logical Bucket-2 (LB_2)	113
6.8	Candidate 2-itemset - Part I	114
6.9	Candidate 2-itemset - Part II	114
6.10	Frequent 2-itemset - Part I	115
6.11	Frequent 2-itemset - Part II	115
6.12	Candidate 3-itemset - Part I	115
6.13	Candidate 3-itemset - Part II	115
6.14	Frequent 3-itemset - Part I	116
6.15	Frequent 3-itemset - Part II	116
6.16	Candidate 4-itemset - Part I	116
6.17	Candidate 4-itemset - Part II	116
6.18	Frequent 4-itemset - Part II	116
6.19	Runtime (in ms) Performance of the Proposed Algorithm without GPU	118
6.20	Runtime (in ms) Performance of the Proposed Algorithm with GPU-acceleration	118

LIST OF ALGORITHMS AND PROCEDURES

Algorithm No.	Title	Page No.
3.1	**RISOTTO:** An algorithm for finding frequent itemsets	49
5.1	**TB-NPF-VDF:** An algorithm for finding frequent itemsets	87
6.1	**GNVDF:** An algorithm for finding frequent itemsets	106
	Procedures in GNVDF	106-108
	6.1.1 eliminate_null: A procedure to eliminate the null transactions in a dataset	106
	6.1.2 one_frequent_itemset: A procedure to find the frequent 1-itemset	107
	6.1.3 find_common_TID: A procedure to find the common transaction ID's	107
	6.1.4 two_freq_itemset: A procedure to find the frequent 2-itemset	107
	6.1.5 *n*_frequent_itemset: A procedure to find the frequent *i*-itemset where $3 \leq i \leq n$	108

LIST OF ABBREVIATIONS

AA	-	Apriori Algorithm
AMA	-	Advanced Matrix Algorithm
APFMS	-	Accelerating Parallel Frequent Itemset Mining on Graphics Processors with Sorting
ARAA	-	Advanced Reverse Apriori Algorithm
ARM	-	Association Rule Mining
BSRI	-	Boolean array Setting and Retrieval by Indexes of transactions
CGMM	-	CPU & GPU based Multi-strategy Mining
CGSS	-	Cluster based Single Scan on a GPU
CMR - Apriori	-	Coding and Map/Reduce - Apriori
CPU	-	Central Processing Unit
CSS	-	Single Scan on a Cluster
CTL	-	Common Transaction List
CUDA	-	Compute Unified Device Architecture
D2P	-	Dynamic Queue and Deep Parallel
DHP	-	Direct Hashing and Pruning
DM	-	Data Mining
DS	-	Data Structure
Eclat	-	Equivalence CLAss Transformation
ESPE	-	Efficient Sequential Pattern Enumeration
FBCM	-	Fast update pruning Based on a Compression Matrix
FIM	-	Frequent Itemset Mining
FIMI	-	Frequent Itemset Mining Implementations
FIUT	-	Frequent Item Ultra metric Tree
FMA	-	Frequent Matrix Apriori
FP-Growth	-	Frequent Pattern Growth

FPM	-	Frequent Pattern Mining
FPMBM	-	Frequent Pattern Mining using a Boolean Matrix
FUP	-	Fast Update Pruning
GNVDF	-	GPU-accelerated Novel Algorithm using Vertical Data Format
GPGPU	-	General-Purpose computing on GPUs
GPU	-	Graphical Processing Unit
GSS	-	Single Scan on a GPU
HDF	-	Horizontal Data Format
HPC	-	High-Performance Computing
IMA	-	Incremental Matrix Apriori
JAB-VDF	-	Jagged Array based Vertical Data Format
JCUDA	-	Java for CUDA
JNI	-	Java Native Interface
KDD	-	Knowledge Discovery in Databases
MAPRIORI	-	Matrix-based Apriori algorithm
MATLAB	-	MATrix LABoratory
MBAT	-	Matrix Based Algorithm with Tags
MB-MFIM	-	Matrix Based Maximal Frequent Itemset Mining
MFI	-	Maximal Frequent Itemset
MFIF	-	Maximal Frequent Itemset First
MFIWDSIM	-	Mining Frequent Itemsets with Weights over a Data Stream using Inverted Matrix
MMS-FPM	-	Multiple Minimum Support - Frequent Pattern Mining
MOA	-	Matrix-Over-Apriori
MSApriori	-	Multiple Support Apriori
MTPAPRIORI	-	Matrix-based Apriori algorithm with pruning optimization and transaction reduction strategy
NPF-VDF	-	Novel Pattern Formations with Vertical Data Format

NSFI	-	N-list and Subsume-based algorithm for mining Frequent Itemset
PRFP	-	Parallel Regular Frequent Pattern
PSPM	-	Parallel Sequential Pattern Mining
RAA	-	Reverse Apriori Algorithm
RBFI	-	Rehashing Based Frequent Itemset
RISOTTO	-	pRefixed ItemSet stOrage verTical daTa fOrmat
SIM	-	Sorting Index Matrix
SIMD	-	Single Instruction, Multiple Data
SS	-	Single Scan
SUMsFPM	-	Sumathi Murugan Frequent Pattern Mining
TB-NPF-VDF	-	Thread Based, Novel Pattern Formations with Vertical Data Format
VBM	-	Vertical Boolean Mining
VDF	-	Vertical Data Format
VDSRP	-	Vertical Data Stream Regular Patterns
VFFM	-	Vertical Format Frequent Mining
YAFIM	-	Yet Another Frequent Itemset Mining

LIST OF SYMBOLS

H_0, H_1	-	Null and Alternate hypothesis
{*in-frequent*$_i$}	-	Set of *in-frequent* items in candidate *i*-itemset
{*itemset*$_i$}	-	Set of frequent *i*-itemset
$\|D\|$	-	Number of transactions in a dataset
$\|I\|$	-	Number of items in a dataset
$\cap$	-	Set intersection
$\cup$	-	Set union
$\bowtie$	-	Natural join
μ_1,μ_2	-	Means of two groups
A, B, X, Y	-	Items in transactional database *D*
s_1^2 and s_2^2	-	Variances of the two groups
arr	-	Jagged array
c	-	Number of columns
C_i	-	Candidate *i*-itemset
C_{i_1}	-	Part I of candidate *i*-itemset
C_{i_2}	-	Part II of candidate *i*-itemset
CM	-	Memory space required for C_{TID}
C_{TID}	-	Common *TID*'s
C_{TID_list}	-	Common Transaction List
D	-	Transactional Database
I	-	Itemset
$I_1,I_2,\ldots,I_m$	-	List of items in Itemset *I*
i-itemset	-	i^{th} itemset
$item_{i1}$	-	First item in the candidate *i*-itemset
$itemset_{i\_1}$	-	Number of items in the first part of frequent *i*-itemsets

$itemset_{i\_2}$ - Number of items in the second part of frequent *i*-itemsets

I_x, I_y - Each item in LB_1 and LB_2

I_z, I_{z+1} - Items in LB_2

k-itemset - k^{th} itemset

L - List of frequent itemsets

LB_1 - Logical Bucket - 1

LB_2 - Logical Bucket - 2

L_i - Frequent *i*-itemset

L_{i_1} - Part I of frequent *i*-itemsets

L_{i_2} - Part II of frequent *i*-itemsets

LK_k - Prefix keys in prefixed itemset storage

LV_k - Values in the prefixed itemset storage

m,p,x - Lengths of each array in a jagged array

M_i - Memory required for frequent *i*-itemset

min_conf - Minimum confidence

min_sup - Minimum support

MS - Memory saved for the entire dataset *D*

n - Total number of transactions in *D*

n_1 *and* n_2 - Sizes of two groups

Ø - Null set

PIDS - Prefixed-itemset storage

r - Number of items in the grocery shop

$rbytes_i$ - Number of bytes of memory removed from the candidate *i*-itemset as *in-frequent*

rr_1 - Number of rows to be removed as *in-frequent*

SC - Support Count

SC_{item} - Support Count of the item *item*

T - Transaction

tc_i	-	Possible *i*-item combinations
TID/tid	-	Transaction ID
$TID\text{-}List_1$, $TID\text{-}List_2$, $TID\text{-}List_3, \ldots,$ $TID\text{-}List_n$	-	List of TIDs
TID-set	-	Set of Transaction ID's
TM	-	Total memory required
TM_{final}	-	Total memory required for the frequent itemsets in GNVDF
TM_i	-	Memory required for candidate *i*-itemset
TNT	-	Total Number of Transactions
t_x	-	Thread *x*
X	-	Itemset
X_1 and X_2	-	Means of X_1 and X_2
δ	-	Minimum support
δ_{new}	-	New support threshold

Chapter - 1

INTRODUCTION

CHAPTER - 1

INTRODUCTION

A journey of thousand miles begins with a single step

--LAO-TZU

1.1 Background

In recent days, the quantity of data generated or collected from various sources has been increasing enormously. Data Mining (DM) is an interdisciplinary field, has been widely used to analyze those data. Frequent Pattern Mining (FPM) plays a core role in DM, and it enables us to find relationships among the items in transactional databases [AH,14]. Thus, the research incorporates various novel FPM algorithms to mine frequent patterns efficiently with less time and memory usage.

This chapter provides background information necessary for understanding the contributions made in this research. In particular, section 1.2 outlines the basics of DM, section 1.3 deals with FPM, the importance of Association Rule Mining (ARM) is discussed in section 1.4, section 1.5 mentions the various applications of FPM, the scope, aim & objectives of the research work were presented in sections 1.6 and 1.7 respectively. Similarly, the statement of the problem and its description were discussed in sections 1.8 and 1.9 respectively. The elaborate description of the datasets was illustrated in section 1.10 and finally, the chapter organization of the thesis is covered in section 1.11.

1.2 Data Mining

DM is the most commonly used process for exploring and analyzing a large quantity of data to acquire novel, valid, potentially valuable and intelligent patterns hidden in the database [VD,19],[FPS,96]. Databases, data warehouses, the Web,

other information repositories, and streaming data are examples of data sources. The significant tasks of DM are:

i. **Anomaly detection** – unusual items or events in the unlabeled datasets are identified for further analysis.

ii. **Association rule mining** – identifies the relationships between variables. Using this task, the supermarket can find out the products that are purchased together frequently by the customers and use this information for marketing purposes.

iii. **Clustering** – determining the similar groups and structures in the data without using the known structures.

iv. **Classification** – accurately predict the target class for the new data from the model.

v. **Regression** – predict a range of numeric values for the given dataset.

vi. **Summarization** – presenting a more compact representation of the dataset e.g. visualization and report generation.

1.2.1 Knowledge Discovery in Databases

Knowledge Discovery in Databases (KDD) is a repetitive and interactive process of discovering useful knowledge from a collection of data in the context of large databases. Knowledge Discovery and DM are distinct terms. It consists of the following steps:

i. **Data cleaning** – removing noise and inconsistent data.

ii. **Data integration** – combining numerous data sources.

iii. **Data selection** – retrieving relevant data from the database for the analysis task.

iv. **Data transformation** – transforming and consolidating the data into forms that are appropriate for mining by performing summary or aggregation operations.

v. **Data mining** – extracting data patterns by applying intelligent methods.

vi. **Pattern evaluation** – identifying the interesting patterns representing knowledge based on interestingness measures.

vii. **Knowledge presentation** – presenting the mined knowledge to the users by using visualization and knowledge representation techniques.

Steps 1 through 4 are the data pre-processing techniques, which makes the data for mining ready. The DM step may interact with the user or a knowledge base. The interesting patterns are presented to the user and they may be stored as new knowledge in the knowledge base by the last two steps [HPK, 12]. The diagrammatic representation of the KDD process is shown in Figure 1.1.

1.3 Frequent Pattern Mining

FPM is an essential task and plays a vital role in DM tasks such as various kinds of ARM, sequential pattern mining, associative classification and frequent pattern-based clustering. It is widely used in mining associations, correlations, and many other relationships among the data. Mining frequent patterns from large scale databases have become a significant research problem in DM and knowledge discovery community.

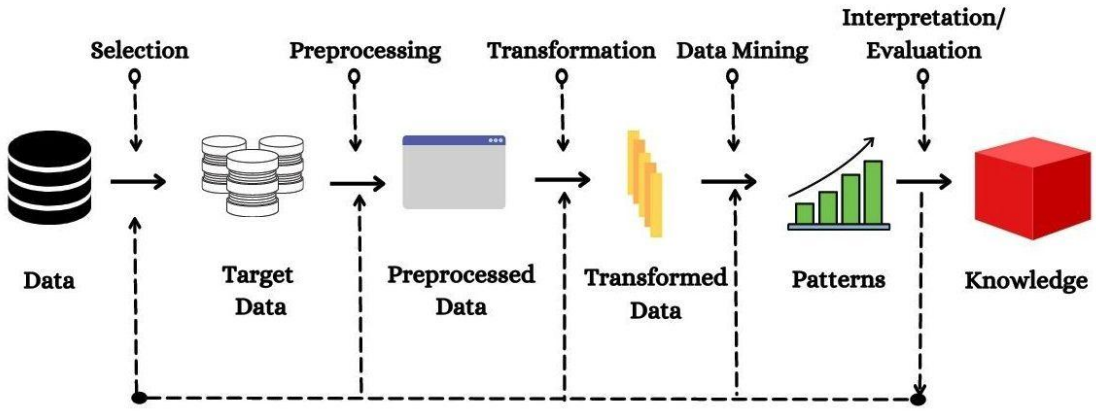


Figure 1.1 KDD Process

1.3.1 Terminologies in FPM

k-Itemset

An itemset or set of items that contain *k* unique items is called *k*-itemset. For example, a set of a desktop computer, a printer that occur frequently in a dataset is called a 2-itemset [HPK,12].

Subsequence

It contains a set of items purchased in sequential order i.e. buying a personal computer first, then a wireless keyboard, and then a wireless mouse that occurs frequently in a shopping history database [HPK,12].

Substructure

It can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently with not less than a user-specified threshold is called structured patterns [HPK,12].

Frequent patterns

These are the patterns/itemsets/subsequences that appear frequently together in transactional datasets or supermarket datasets.

Minimum support count threshold

Users or decision-makers are interested to identify the occurrence of *k*-items with specified numbers of times, which is known as the minimum support count threshold.

Transactional/Supermarket dataset

A transactional dataset is a dataset that contains the items are purchased in each transaction. A transaction *T* in *D* is represented as a pair defined as *T* = <*TID, list of items*>, where *TID* is the unique identification number for the *list of items* purchased by each transaction. Table 1.1 showed below is an example transactional dataset for a grocery store.

Table 1.1 A Sample Transactional Dataset of a Grocery Store

TID	List of items
1	Milk, Butter, Bread
2	Milk, Dry grapes
3	Bread, Butter, Rusk
4	Rusk, Butter
5	Dry grapes, Bread, Butter
6	Ghee, Bread, Dry grapes
7	Milk, Bread, Butter
8	Yummy apple, Grapes
9	Yummy apple, Milk, Ghee
10	Budget milk, Butter, Dry grapes

The discovery of frequent patterns plays an essential role in DM. A commonly used application is the market basket analysis, where the frequently purchased items

are discovered from the transactional entries of a grocery store for making business decisions. Many efficient and scalable algorithms have been developed for FPM, from which the association and/or correlation rules can be derived, which helps in making business decisions and predictions. These algorithms are categorized into three major groups [HPK,12].

i. Apriori-like algorithms

ii. Frequent pattern growth-based algorithms such as FP-growth

iii. Algorithms that use the Vertical Data Format (VDF)

1.3.2 Basic Definitions

Let $I=\{I_1, I_2,\ldots, I_m\}$ be an itemset, and D is a transactional database containing a set of transactions T and is a non-empty itemset such that $T \subseteq I$ and each transaction T is holding a unique identifier *TID*. Let A be a set of items. A transaction T is said to contain in A if $A \subseteq T$. The format of the association rule is $A \rightarrow B$, where $A \subset I$, $B \subset I$, $A \neq \emptyset$, $B \neq \emptyset$, and $A \cap B = \emptyset$ [HH,16]. Association rule $A \rightarrow B$ that holds in D with support (s) and confidence (c) [HD,16].

Support(s): The support of an association rule $A \rightarrow B$ is defined as the percentage of records that contain $A \cup B$ to the total number of records in the database [ST,16]. It is noted that the support count is increased when an item is present in numerous transactions in the database D [ST,16].

$$\text{support}(A => B) = p(AUB) \qquad \ldots \text{Equation (1.1)}$$

Confidence(c): The confidence of a rule $A \rightarrow B$ is defined as $s(A \rightarrow B)/s(A)$. It is the ratio of the number of transactions that contain all items in the consequent (B), as well

as the antecedent (*A*) to the number of transactions that include all items in the antecedent (*A*) [PP,15].

$$\text{confidence}(A => B) = \frac{\text{Support\_count}(AUB)}{\text{Support\_count}(A)} \quad \text{… Equation (1.2)}$$

The minimum support threshold is used to obtain the frequent itemsets from the databases. In contrast, the minimum confidence constraint is applied to those frequent itemsets found previously in determining the best rules.

1.3.3 Architecture/Layout of Storing Transactional Data

There are two formats in which a transactional database can be represented. i) Horizontal Data Format(HDF) ii) Vertical Data Format(VDF).

i) Horizontal Data Format

This representation consists of two columns namely *TID* and List of Item *ID*s, where *TID* is a transaction *ID* and List of item *ID*s specifies the items bought by the customer for the *TID*. Both the Apriori and FP-growth algorithms mine the frequent patterns in HDF and it is shown in Table 1.2.

ii) Vertical Data Format

In VDF, the data can be expressed in {*itemset*:*TID_set*} format where the *itemset* is the name of the item and *TID_set* is the transaction set that contains the itemset. The VDF is used in the Eclat algorithm that minimizes the database scan and it uses a set intersection of *TID*s for finding the Support Count (*SC*) for *k*-itemsets where *k*=2,3,...,*n*. The VDF of *D* is shown in Table 1.3.

Table 1.2 Transactional Database D in HDF

TID	List of item IDs
T1	A,B,E
T2	B,D
T3	B,C
T4	A,B,D
T5	A,C
T6	B,C
T7	A,C
T8	A,B,C
T9	A,B,C,E

Table 1.3 VDF of D

itemset	TID_set
A	T1,T4,T5,T7,T8,T9
B	T1,T2,T3,T4,T6,T8,T9
C	T3,T5,T6,T7,T8,T9
D	T2,T4
E	T1,T9

1.4 Association Rule Mining

ARM is a process for finding interesting associations and relationships between data items in datasets. It is a successful technique for extracting knowledge from databases. It discovers the frequent if-then rules called association rules and it is used for analyzing and predicting customer behaviour. They are essential in customer analytics, product clustering, market basket analysis, catalogue design and store layout. Every association rule has two parts: i) an antecedent (if) and ii) a consequent (then). An antecedent refers to the item found within the data whereas the consequent is an item found in combination with the antecedent. It uses the criteria namely

support and confidence to identify the most important relationships. Support indicates how frequently the items appear in the data. Confidence indicates the number of times in the if-then statements is found true [HPK,12].

The discovery of association rules involves two major steps. They are:

i. Finding frequent patterns/itemsets

ii. Generating reliable and strong association rules from the frequent itemsets [HPK,12]

Step 1 of ARM is a challenging task [SD,15] and plays a vital role in mining associations and correlations [DS,16]. This research work focuses on FPM algorithms. In general, the FPM can be categorized into three main groups viz., Join-Based, Tree-Based, and Pattern Growth [ABH,14] as shown in Figure 1.2.

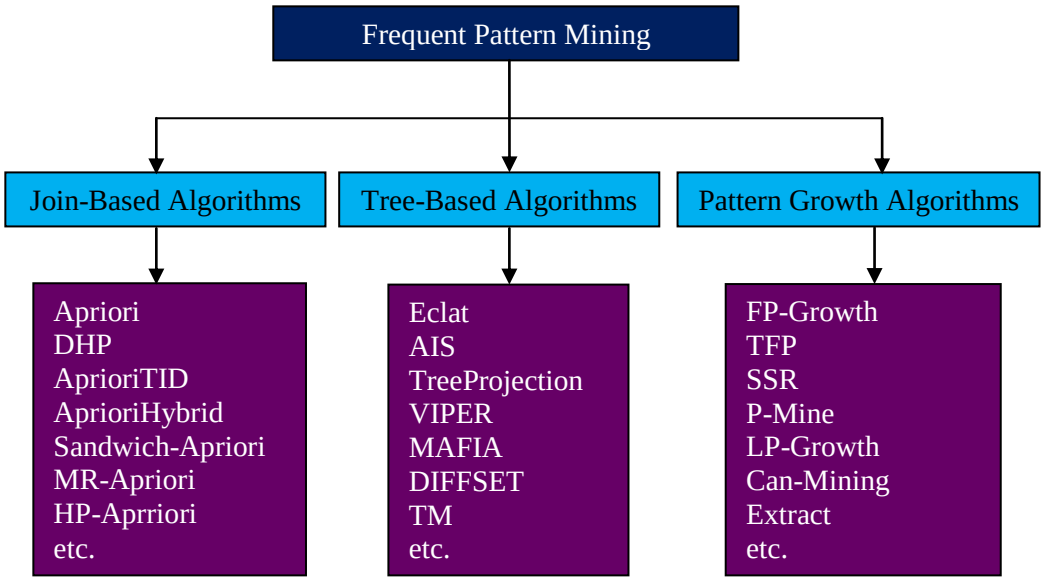


Figure 1.2 Classification of Frequent Pattern Mining Algorithm

The Join-Based algorithms use a bottom-up approach to discover frequent patterns in a dataset and find the larger itemsets as long as their itemsets appear more than a prescribed threshold defined by the user in a database. The Tree-Based algorithms use set-enumeration concepts by constructing a lexicographic tree that

enables the items to be mined with either breadth-first or depth-first order. Finally, the Pattern Growth algorithms implement a divide-and-conquer approach to partition and project databases depending on the presently identified frequent patterns and expand them into longer ones in the projected databases.

Apriori Algorithm, FP-Growth and Eclat (Equivalence CLAss Transformation) are the popular static DM techniques for finding frequent patterns [Sin,16] using the above strategies.

1.4.1 Apriori Algorithm

It is one of the most popular algorithms and it is the first algorithm proposed by R.Agrawal and R.Srikant in 1994 in the field of DM and it is a classical algorithm of ARM. It generates frequent itemsets for the Boolean association rule. Since the algorithm uses the prior knowledge of the frequent itemset properties it is named Apriori. It uses an iterative approach called level-wise search, where k^{th} itemset is used to explore $(k+1)^{th}$ - itemsets. There are two steps involved in each iteration and it is repeated when no candidate itemsets can be found. They are:

i. Generation of candidate itemsets

ii. Finding the occurrence of each candidate itemset in a database and pruning all disqualified candidate itemsets based on support count(threshold) and closure property i.e. if a set of items is frequent, then all of its proper subsets are also frequent [HPK,12]

After finding the frequent itemsets, the association rules are generated from those large itemsets with the constraints of minimal confidence (*min_conf*)

and minimum support (*min_sup*) thresholds. Figure 1.3 shows an illustration of the Apriori algorithm [CJAH$^+$,19].

In this example, the transactional database *D* contains four transactions and the items sold are A, B, C, D and E. Let the *min_sup* be 2. Initially, the *D* is scanned once to create candidate 1-itemset C_1. From Figure 1.3, it is identified that the *SC* of {D} is less than the *min_sup* and it is removed. The L_1 contains the items A, B, C, and E. After finding L_1, $L_1 \bowtie L_1$ is performed and to find the *SC* for C_2, *D* is scanned again. In this case, the itemset combinations {A,B} and {A,E} doesn't satisfy the *min_sup* and they are removed. The item combinations after removing the items viz., {A,C},{B,C},{B,E} and {C,E} forms L_2. This process is iterated until no more candidate and/or frequent itemsets are found.

1.4.1.1 Disadvantages of Apriori Algorithm

The classical Apriori algorithm is inefficient because

i. It is not suitable for large databases
ii. It defines the presence and absence of an item
iii. It allows uniform *min_sup* threshold
iv. More scanning of the transactional database is needed for generating frequent itemsets
v. More I/O cost is required
vi. Generation of candidate itemsets and support counting is expensive and also memory consuming [CJAH$^+$,19]

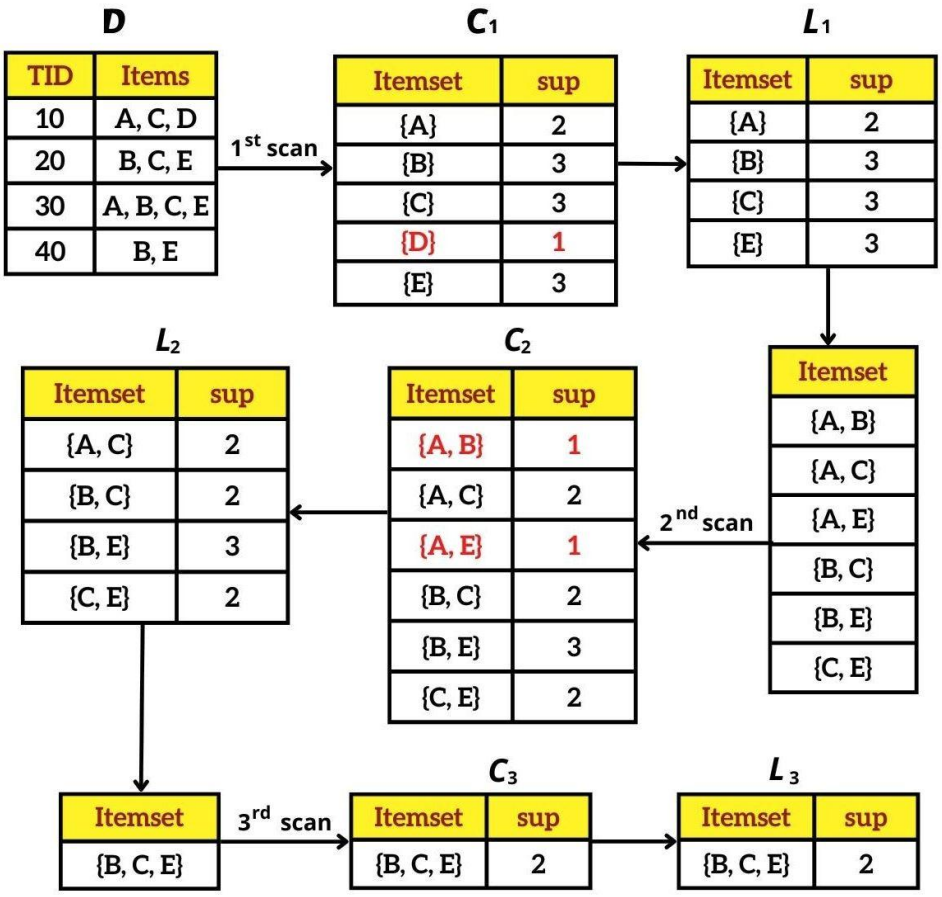


Figure 1.3 An Example for Apriori Algorithm

1.4.2 FP-Growth Algorithm

FP-growth depends on a prefix-tree configuration which stores the database into a compact form known as FP-tree. It follows the divide-and-conquer approach. It first compresses the database representing frequent items into an FP-tree, which keeps the association information of the itemsets. It then divides the FP-tree into sub-trees called conditional FP-trees using the dataset called conditional pattern base. [SBE,21],[CJAH^{+},19],[HPK,12].

1.4.2.1 Advantages of FP-Growth Algorithm

i. Faster than Apriori algorithm

ii. No candidates are generated

iii. Only two passes over the dataset

1.4.2.2 Disadvantages of FP-Growth Algorithm

i. FP tree may not fit in memory

ii. FP tree is expensive to build

1.4.3 Eclat Algorithm

It uses VDF and finds all frequent itemsets by intersecting the *TID-list*. It first scans the database and determines the *TID*s in which the item occurs for each item. The (*k*+1)-itemsets were generated from *k*-itemset using Apriori property and depth-first search computation. The *TID*s of (*k*+1)-itemsets are generated by intersecting the *TID-sets* of frequent *k*-Itemset. This process continues until no more candidate itemsets are found. An example of the Eclat algorithm is shown in Figure 1.4 [CJAH^{+},19].

1.4.3.1 Advantages & Disadvantages of Eclat Algorithm

i. It does not require repeated scanning of the database to find the support of *k*+1 itemsets and it is obtained using the set intersection method from *k*-itemsets

ii. It is faster than the Apriori algorithm as it uses depth-first search

iii. Though it requires less memory consumption than Apriori, the usage of array storage structure requires huge memory and computational time for intersecting the sets when there are many transactions

Thus, to eradicate the said disadvantages, a vast amount of research has been contributed to FPM and many remarkable algorithms have been proposed in the last

two decades. Further, the research contributions proposed in this thesis provides modifications to the standard and/or existing algorithms to reduce the execution time/runtime and memory space in finding the frequent patterns.

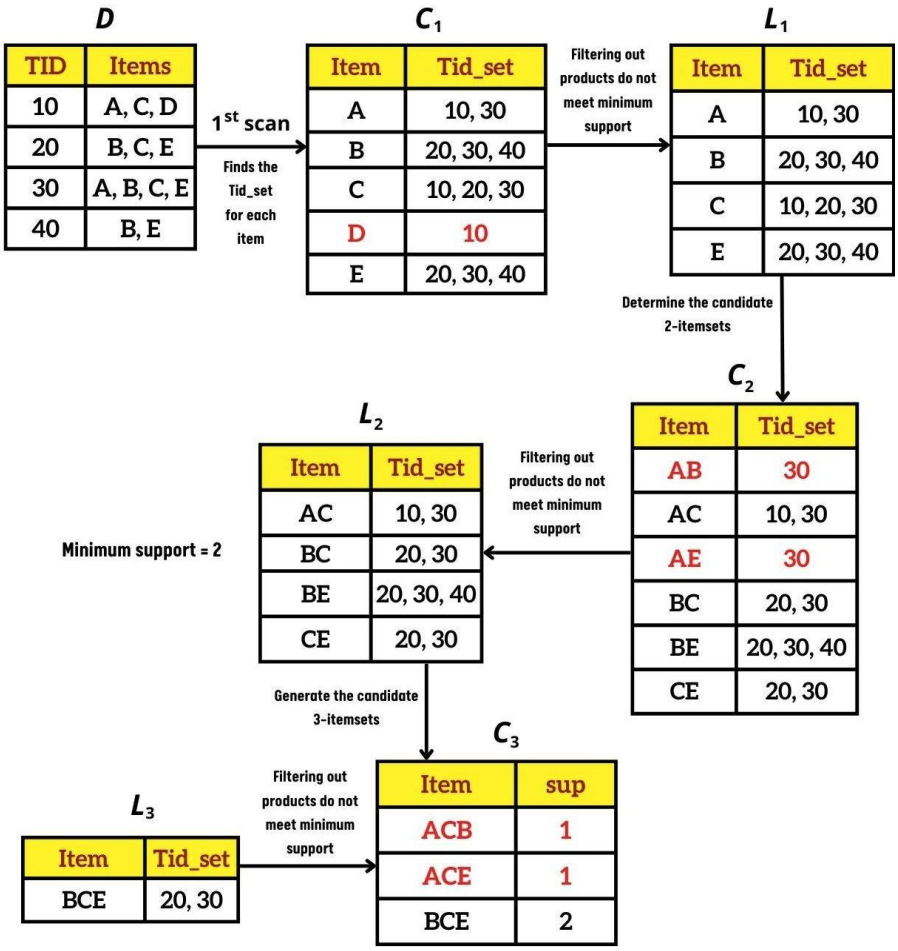


Figure 1.4 An Example of Eclat Algorithm

1.5 Applications of FPM

FPM has been used in a variety of real-world applications to improve decision-making and management.

In the business world, mining frequent patterns assist the business people in designing promotion schemes, providing discounts, organizing self and store layout, posting special advertisements, storage management and forecasting potential markets [CGGK,00].

In the medical domain, the frequent patterns enable the doctors to make treatment decisions and uncover the gene actions [OKSI,00],[Wet,02].

In education, mining frequent patterns enable the teachers in modifying the teaching methods to improve their teaching quality, to facilitate students to be trained better and select the contents of teaching based on the student's calibre [MLWY^{+},00].

In disaster prevention, mining frequent patterns assist in weather forecasting by analyzing different environmental factors and help to prevent impending [ZWH,04].

Similar to the previous it can be used in many other fields like police department, engineering design, software bug detection and recommendation systems [ABH,14]. Thus, mining frequent patterns plays a hot topic of research for the past twenty decades.

1.6 Scope of the Research Work

FPM has been a purposeful research area in DM for the past two decades. Many researchers contributed numerous competent and scalable techniques for determining the frequent itemsets from transactional databases. Nowadays, online shopping become a mandatory mode of purchase in human lives and amazon like e-stores display the items which are "*frequently bought together*" to their customers and provides offers based on that. In this way, the e-stores increase their sales and profit considerably. Also, they display "*Customers who bought this item also bought*" in their web portal along with the product description and reviews. For displaying this information, FPM is an essential task and though there are scalable algorithms exists, the prolonged processing time and more memory consumptions are the major issues

in mining frequently bought items. So, there is always a need for developing better algorithms with reduced runtime requirements and memory usage.

Thus, this research work focuses on developing efficient FPM methods in finding frequent patterns in such a way that the runtime and usage of memory to be reduced than the existing algorithms.

1.7 Aim & Objectives of the Research Work

Even though an enormous amount of remarkable research works have been contributed by many researchers for FPM to efficiently mine the frequent patterns from transactional datasets, the requirement of prolonged processing time and a large amount of memory space are still the two major issues that the FPM faces, especially when the amount of data is large.

To solve the above said issues, the research work aims to devise novel FPM algorithms to determine the frequent patterns from the static datasets to achieve the following objectives:

i. To develop FPM algorithms that efficiently mine the frequent patterns with a minimum runtime

ii. To formulate the FPM algorithms to consume less memory in mining frequent patterns

Thus, the research work focuses on developing robust FPM algorithms for reducing the runtime and consumption of memory in mining the frequent itemsets from transactional datasets. To evaluate the proposed FPM algorithms, they were compared with some existing algorithms to prove that the proposed algorithms will detect the frequent patterns faster with less memory.

1.8 Problem Statement

To accomplish the said objectives, four research models have been proposed in this research work as a research framework called SUMsFPM. They are:

i. **RISOTTO** – A Novel Hybrid Approach for Enhancing Classical Apriori Algorithm

ii. **JAB-VDF** – A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach

iii. **TB-NPF-VDF** – A Multithread, Novel Pattern based Algorithm for Finding Frequent Patterns with Jagged Array and Vertical Data Format

iv. **GNVDF** – A GPU-accelerated Novel Algorithm for Finding Frequent Patterns Using Vertical Data Format Approach and Jagged Array

1.9 Problem Description

The RISOTTO algorithm has been developed by combining both Prefixed-itemset based storage structure and VDF approach to reduce runtime needed to find the frequent patterns from the transactional datasets. The Prefixed-itemset based storage structure utilized in this research work generates a fewer number of candidate itemset in each iteration of the algorithm. Similarly, the usage of VDF restricts the number of database scans to one rather than $(2^{|I|} - 1)$ times where $|I|$ is the number of items in a dataset.

The array storage structure utilized in the VDF normally requires more storage space as there are enormous numbers of *TID*s for each item in the transactional database. So to reduce the memory space, a Jagged Array Based - Vertical Data Format (JAB-VDF) has been proposed in this research.

Further, to minimize runtime and memory requirements, TB-NPF-VDF and GNVDF have been developed. The TB-NPF-VDF method generates frequent patterns by adopting a novel pattern generation method with multiple threads. Usage of multiple threads reduces the runtime required in generating frequent patterns and also utilizes the CPU effectively. Further, it uses the jagged array storage representation to minimize the memory requirement in preserving the frequent patterns.

To reduce runtime and memory space further, a GPU-accelerated method for finding frequent patterns with novel pattern generation using VDF with jagged array has been proposed. The adaptation of the novel pattern generation method in this research contribution generates lesser candidate itemsets than TB-NPF-VDF which reduces the runtime requirement. To reduce memory space further when compared to JAB-VDF, GNVDF adopts a data structure called Common Transaction List (*CTL*), which preserves the common *TID*s of all items in frequent 1-itemset and they were removed from it. The removal of the items in *CTL* from frequent 1-itemset reduces memory space significantly. Thus, all the methods proposed in the research work reduces the runtime and memory space.

The workflow of the proposed research is shown in Figure 1.5. The proposed framework is called **SUMsFPM** which is coined by taking the first two characters from my name Ms.P.**SU**MATHI, the first character from my research supervisor name Dr.S.**M**URUGAN and the first characters from the phrase "**F**REQUENT **P**ATTERN **M**INING".

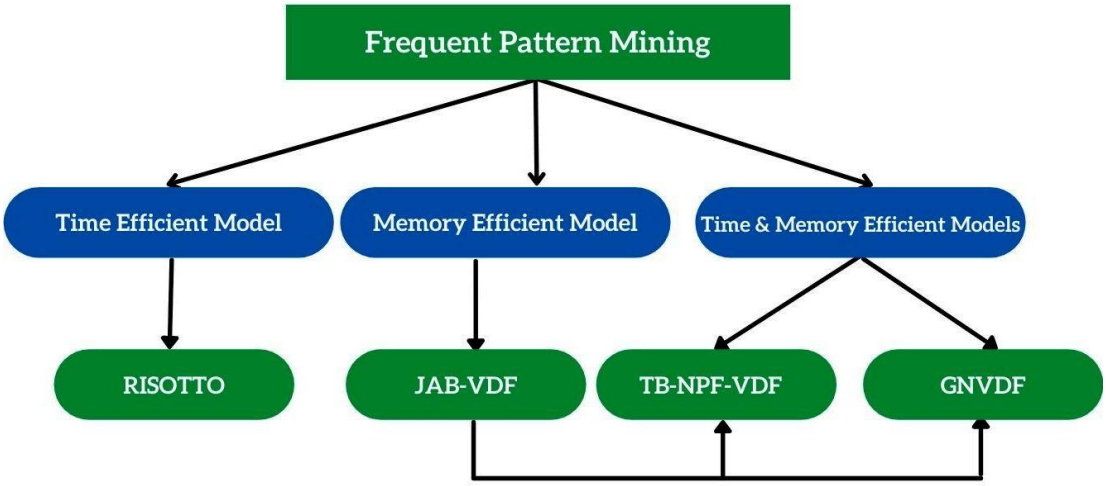


Figure 1.5 Workflow of the Research

1.10 Description of the Datasets

All the proposed algorithms were implemented using Python programming language (version 3.8.2), and GNVDF was implemented with CUDA Toolkit with NVIDIA GPU. To do a uniform and fair comparison, the experiments of all algorithms were conducted using the same software and hardware configurations. The experiments were performed using 8.00GB RAM, Intel Core i7 with 2.40GHz 64-bit processor and Windows 8.1. To evaluate the effectiveness of the proposed methods, an empirical study was conducted with four datasets viz., chess, mushroom, t25i10d10k and c20d10k. Out of the four datasets, chess and mushroom are the real-time datasets, t25i10d10k and c20d10k are the synthetic datasets. The synthetic datasets were normally generated through computer algorithms as an alternative to real-time datasets i.e. they are spawned digitally and not collected in the real world. All the datasets were obtained from the FIMI repository (http://fimi.ua.ac.be) and an open-source Data Mining Library (http://www.philippe-fournier-viger.com/spmf). The characteristics of the datasets were illustrated in Table 1.4.

Table 1.4 Characteristics of Datasets

Datasets/ Databases	Number of transactions #|*D*|	Number of items #|*I*|	Average item count per transaction	Maximum length	Density %
chess	3196	75	37.00	37	49.33%
mushroom	8416	119	23.00	23	19.33%
t25i10d10k	9976	929	24.77	63	2.66%
c20d10k	10000	192	20.00	27	10.42%

The reason for choosing those datasets is that many researchers used them as bench-mark datasets for Frequent Itemset Mining and ARM-based research.

1.11 Chapter Organization

The organization of the thesis is given below.

In Chapter 2, a thorough investigation of the review of literature is made about finding frequent itemsets with their limitations since 2003. The investigation paves way for the proposed methodologies.

Chapter 3 presents a hybrid model called RISOTTO proposed in this thesis for finding frequent itemsets with an illustrative example. It also describes the basics of Prefix-itemset storage structure and VDF. Further, it analyzes the results of RISOTTO by comparing it with Prefixed-Itemset Storage and VDF.

Chapter 4 presents a memory-efficient data structure called jagged array for the VDF approach in finding frequent itemsets. It describes how the jagged array reduces the memory requirements mathematically along with an illustrative example. Further, it discusses the memory comparison in GB between JAB-VDF and VDF with δ=20%.

An FPM algorithm using the multithreaded concept with a novel way of pattern generation and jagged array using VDF called TB-NPF-VDF has been presented in chapter 5 along with an appropriate illustration. It also describes the importance of multithreading with its advantages. Further, it discusses the results of the comparison with Matrix-Apriori, VDF and NPF-VDF.

Chapter 6 illustrates the background of the Graphical Processing Unit (GPU) and the processing flow of CUDA. It explains the proposed methodology, GNVDF: a GPU-accelerated novel algorithm for finding frequent patterns using the VDF approach and jagged array with an appropriate illustration. It also describes memory usage required using the mathematical equations and how much amount of memory is saved in comparison with the JAB-VDF. Further, it discusses how the GPU enables the execution speed when compared with the same method without the usage of GPU.

The last chapter, chapter 7 is devoted to the summary of the key contributions along with possible future extensions.

By implementing the proposed FPM algorithm, the business users can make better decision making and increase the profit of their organizations by identifying the significant frequent patterns with minimum runtime and memory consumption.

Chapter - 2

REVIEW OF LITERATURE

CHAPTER - 2

REVIEW OF LITERATURE

You need to understand things in order to invent beyond them

--Bill Gates

2.1 Background

In general Data Mining (DM) tasks are classified into two categories. They are i) Descriptive Mining and ii) Predictive Mining. Descriptive mining is the process of generating patterns from the existing data and is used for creating meaningful subgraphs, whereas predictive mining is to forecast the explicit values based on the patterns determined from the known results. Association Rule Mining (ARM) is a descriptive mining technique of DM. It is the process of discovering items, which tend to occur together in transactions i.e. which items are most frequently purchased by the customers. Association rules will help the retailer to develop marketing strategies and inventory management to increase the sale of their organization.

Finding association rules can be decomposed into the following two subtasks.

i. Discovering all itemsets whose support is greater than the user-specified minimum support is called FPM.
ii. Generating the desired rules from the frequent itemsets with at least the specified minimum confidence.

FPM is a vital part of ARM which investigates the frequent patterns from the transactional databases. As the data are to be mined is large, a huge amount of time and memory is needed for accessing data and to store the frequent patterns respectively. Though there are two decades of research in FPM, research in reducing the time and minimizing the memory requirement is a quite common issue in finding frequent patterns in FPM because there are huge data generated every day from

various sources. Several FPM algorithms have been proposed in the literature and this chapter presents a brief overview of the relevant research works and which provides a stronger lead to the proposed research models.

2.2 Works Related to Apriori and FP-Growth

In [THY,09], the authors have introduced a novel method for mining frequent itemsets called FIUT (Frequent Item Ultrametric Tree). In that, the authors have used a special UT for enhancing the efficiency in obtaining frequent itemsets. Based on the comparison with the FP-growth algorithm, it was proved by them that the FIUT outperforms FP-growth by reducing I/O overhead and search space. The FIUT generates the frequent itemsets only by checking the leaves of the FIU tree without traversing the tree recursively and also using compressed storage.

An improved version based on Coding and Map/Reduce (CMR-Apriori) has been proposed in [GR,13]. They compared the traditional Apriori, Apriori algorithm with parallel processing and CMR-Apriori and proved that the CMR-Apriori algorithm outperforms others with twice Map/Reduce processes.

A new algorithm called enhanced Apriori algorithms has been introduced in [LVSM,14], which takes less scanning time and reduces the I/O spending time by cutting down the unwanted transaction records in the database. A new algorithm called semi-Apriori using a binary-based data structure for mining frequent itemsets as well as association rule has been proposed in [FAB,14] and proved that this technique outperforms Apriori in terms of execution time.

An improved Apriori has been designed in [SNM,15]. In this method, the transaction *ID*s along with the support count is maintained in the frequent itemsets and they generated the k+1 itemset by set intersection and proved that the number of database scans is reduced than the classical Apriori algorithm.

In [BGD,15], the authors have proposed an improved version of Apriori for reducing the time for searching the database and the memory space by partitioning. A novel Apriori algorithm has been proposed in [JS,15] to overcome the limitations of the classical Apriori algorithm based on local and global power set and observed that the novel algorithm requires only two scans instead of many scans as in the classical Apriori algorithm. In [PD,16], the authors surveyed the improved approaches of Apriori from 2012 to 2015.

In [LS, 16], the authors have introduced a Modified Apriori algorithm using the greedy and vectorization method. They compared the execution time of traditional Apriori and Modified Apriori by varying the number of transactions and proved that the Modified Apriori requires less time than the Apriori. They also proved that the proposed method reduces the number of rules generated than the original Apriori.

The authors in [BDH, 16] have developed a new recursive algorithm based on Apriori called Meta-Apriori. In that, they partitioned the whole database into smaller ones using the divide and conquer approach. After partitioning, they applied Meta-Apriori if the partition is huge or Apriori if it is of reasonable size. Finally, they merged the achieved results to get the result for the whole database and proved that Meta-Apriori requires less time than the Apriori.

In [DZZC,16], the authors have proposed a modified Apriori called DC_Apriori. In this, the authors have restructured the storage structure of the database and they generated *k*-frequent itemsets by joining the 1-frequent itemsets with (*k*-1)-frequent itemsets. It prevents generating invalid candidate itemsets, reduces the database scans and also enhances the itemset generation.

A modified Apriori has been proposed in [KSG,16] using the transposition technique and proved that it is less complex than the classical Apriori. An improved Apriori algorithm has been presented in [RS_1,16] and made a comparison between conventional Apriori and Improved Apriori algorithms. It was proved that the improved Apriori provides better performance than the classical Apriori algorithm**.**

A prefixed-itemset based data structure for candidate itemset generation has been proposed in [YZ,16]. It requires smaller memory space and carried out the connection and pruning operations much faster than Apriori. It was analyzed that the proposed structure improved the efficiency of the classical Apriori algorithm.

The authors in [VLC+,16] proposed a new algorithm for mining frequent itemsets based on the idea of N-lists, an improved version of PrePost called NSFI algorithm which uses a hash table. The empirical results showed that NSFI outperforms PrePost and Eclat.

A method called Advanced Reverse Apriori Algorithm (ARAA) has been proposed in [BPG,17], which is opposite to Apriori. In that, the authors have generated the k^{th} itemset first and moved on to the lower level sets i.e. k-1,k-2,…,1. They compared Apriori Algorithm (AA), Reverse Apriori Algorithm (RAA) and ARAA and proved that the number of scans in ARAA is less than the AA but greater than RAA and is equal to the number of transactions in the database. Also, proved that the ARAA is more suitable for all types of datasets but RAA is applicable for higher datasets because it drastically reduced the multiple scans, execution time and also increased throughput.

2.3 Works Related to Matrix-based Apriori

In [EZ,03], the authors have introduced a new disk-based ARM algorithm called Inverted Matrix. In this method, the transactional data is first converted into a new database layout called Inverted Matrix to avoid multiple scanning of the database. Using this, the frequent pattern could be found in less than a full scan with random access. They have also built a small independent tree by summarizing the co-occurrences for each frequent item and finally, a non-recursive mining process could be applied to reduce the memory requirements with minimum candidate generation. From the experimental studies, they have revealed that the Inverted Matrix approach outperformed the FP-Tree algorithm, especially in mining very large transactional databases.

The authors in [YH,05] have proposed a new matrix algorithm for generating a large frequent candidate itemset efficiently. It generates a matrix and the frequent candidate sets were obtained from that matrix. Numerical experiments and comparisons were performed using the Apriori algorithm for small, medium, and large size datasets. The experimental result confirms that the proposed algorithm outperforms the Apriori algorithm.

The authors in [PVG,06], have introduced a novel method called Matrix Apriori, which utilizes simple data structures viz., matrices, and vectors to generate frequent patterns. They have found that the algorithm minimizes the number of candidate itemsets generated, thereby efficient computation is achieved than Apriori and FP-growth algorithms.

In [HYW,08], the authors have developed a novel method called Efficient Sequential Pattern Enumeration (ESPE) based on a 2-sequence matrix to

mine sequential patterns without setting minimum support in advance. This approach finds frequent sequences from all 2-sequences by scanning the sequence database only once. It uses simple mathematical equations and an efficient storage structure for computing the index of all 2-sequences. Further, it supports the incremental addition of new items and sequences. They have proved that the performance of ESPE is better than the AprioriAll and PrefixSpan for various datasets.

In [ZLZ,08], the authors have initiated a novel algorithm based on the Boolean matrix. It finds outs the maximum frequent itemsets in a short time and scans the database once through the vector and matrix operations. Further, it does not produce any candidate itemsets. The authors in [YE,10] compared the novel matrix Apriori and FP-growth algorithms and revealed that both the algorithms are better alternatives to the Apriori algorithm in terms of database scan and candidate generation. The FP-growth is better than Apriori when the minimum support value is decreased. Matrix Apriori algorithm was proposed as a faster and simpler alternative by combining both Apriori and FP-growth.

The authors in [Jin,10] have presented a new mining algorithm for discovering Maximal Frequent Itemset (MFI). It eliminates and plotting blocks to the matrix by simply counting the value of rows and columns and the experimental result showed that the proposed algorithm provides an efficient result. In [ZWX,10], the authors have presented a method called MaxMatrix which does MFI checking by using the pseudo-projection matrix of the MFS matrix. It uses only logical operation for MFI checking which saves system resources significantly because it does not allocate new memory space for the pseudo-projection matrix. Further, they proved that the method reduces the MFI generation time and the number of subsets used for ARM.

The authors in [YWWJ,11] have developed an innovative method called Boolean matrix. In that, they used the Boolean matrix array to replace the transaction database and removed the non-frequent itemsets from the matrix. To generate the *k*-frequent itemsets, the vector operation "AND" and the random access characteristics of an array are used in the Hadoop Platform and proved that it exponentially increases the efficiency of the algorithm. Y.S.He and P.Du [HD,11] have built a new algorithm based on compressed matrices which improve the efficiency of creating *k*-frequent itemsets, by scanning the database once, and thereby mining association rules is also improved. The newly created algorithm reduces I/O load and also improves the speed of discovering frequent itemsets, especially in large itemsets.

To overcome the disadvantages of the Apriori algorithm, the authors in [WS,11] have initiated a Boolean matrix, and the transaction data is converted to Boolean values and stored in place. It generates frequent itemsets directly from the Boolean matrix and also saves a lot of memory space. This approach requires only one database scan and reduces the number of candidate sets and system costs.

The authors in [MDA,11] have proposed an Advanced Matrix Algorithm (AMA) for finding out frequent itemsets from the transactional database using the Boolean matrix by scanning the database only once. The proposed algorithm is more efficient and effective in generating frequent itemsets and removed the most significant issue of ARM, especially on computational complexity which handles huge transactional databases.

The authors in [Wan,11] have proposed an improved algorithm for ARM-based on a relation matrix. The transaction database is scanned and stored in the matrix with entries either one or zero. The frequent itemsets are generated from

the relation matrix and then the association rules are derived from the frequent itemsets. They have shown that the proposed algorithm is efficient both in theoretical and experimental analysis.

The authors have developed a matrix algorithm [DD,12], which transforms the database into a matrix database. In this, the frequent *k*-itemset is obtained from the matrix which avoids the repeated database scan and proved that it greatly reduced the number of candidate itemsets and improved the efficiency of computing.

In [OE,12], the authors have focused on the solution to an incremental update problem by proposing the Incremental Matrix Apriori (IMA) algorithm. It scans only new transactions, allows the change of minimum support, and handles new items in increments. The matrix Apriori works without candidate generation and scans the database twice. The experimental results showed that the IMA provides speed-up between 41% and 92% while increment size is varied between 5% and 100%.

T.N.Mujawar et al. [MSB,12] have presented an approach for mining association rules from XML data using XQuery and Apriori algorithms without any pre and post-processing. In this research article, a Matrix-based Apriori algorithm (MAPRIORI) and an improved matrix-based Apriori algorithm with pruning optimization and transaction reduction strategy (MTPAPRIORI) were implemented. The result showed that the database is scanned only once in both of these algorithms. Further, it is observed that the number of frequent itemsets generated and the running time by the MTPAPRIORI algorithm is less than the MAPRIORI with different support levels. Also, it reduces the scale of the transaction database to be scanned and provides overall efficiency.

A Matrix-based multidimensional sequential pattern mining algorithm has been introduced in [QL,12]. It does not need the repeated scan of the database to

generate a 1_Large sequence *k*-Itemset. During the first scan, 1_Large itemsets are obtained. The frequent patterns are obtained from the Boolean sequenced matrix using set and matrix theory in the second database scan. It occupies less memory, improves mining efficiency, and runs faster than other algorithms. H. Singh and R. Dhir [SD,13] have presented a new Matrix Based Algorithm with Tags called MBAT. It is based on transactional matrix and transaction reduction to find the frequent itemsets and proved that the MBAT is more efficient than the classical Apriori algorithm in ARM.

A.R.H.Alwa and B.A.V.Patil [AP,13] have launched a novel approach to improve the Apriori algorithm using Matrix-File. This approach extracts particular rows and columns and performs a function on that rather than scanning the entire database. It outperforms the classical Apriori algorithm because the pruning process is applied to those columns whose item count is less than the minimum support. It also saves time and speed by reducing the redundant scanning of the database.

To solve the problem in Apriori the authors in [YXHJ^{+},13] have proposed an improved frequent itemset mining algorithm based on Sorting Index Matrix (SIM). It generates frequent 2-itemset from 1-itemset vector and the corresponding matrix multiplication sequentially. From the frequent 3-itemset, it creates a simple SIM for frequent *k*-itemsets. The entire process simply scans the database only once and does not produce candidate itemsets. From the experimental outcomes, they have shown that the SIM improves the efficiency of mining frequent itemsets than the existing methods.

Using dynamic matrix Apriori and Multiple Support Apriori (MSApriori), the authors in [Cha,14] have built a methodology to mine association rules over

dynamic databases. From the experiments, it was found that a remarkable improvement has been achieved in terms of time, and the number of frequent items and generated rules. A Matrix Apriori with an incremental approach for ARM has been proposed in [BML,14] which were based on Apriori and FP-growth algorithms. It uses simple data structures namely matrix and vector, generates frequent patterns, and minimizes the number of itemsets. It improved the speed of the mining process and also increased efficiency than the previous algorithms.

A new method named Dynamic Matrix Apriori has been proposed by R.Chaudhary et al. [CSS,15] using the dynamic matrix technique, which is much faster when compared to traditional Apriori in the generation of candidate itemsets. They also have proposed a new framework that uses the Map Reduce programming model. From the experiments on a large set of databases, they have achieved an improved result in terms of runtime, the number of generated frequent itemsets and rules. In [VP,15], the authors have proposed a method based on transaction reduction techniques for mining frequent patterns from large databases. In this, the data is compressed in the form of a bit array matrix and the whole database is scanned only once. To achieve efficiency, the frequent patterns are mined from this matrix by using the count-based transaction reduction and support count method.

The authors in [AH,15] have introduced a novel method to find frequent itemset using probability and matrix in two steps. In the first step, a preliminary matrix is generated for the dataset. The regular itemsets are directly generated from the probability matrix in the second step. The improved algorithm reduces the number of comparisons and scans.

In [TG,15], the authors have introduced a vertical format approach for finding frequent itemsets using the Boolean matrix. The presence of an item for the *TID*s is

represented as 1 and 0 otherwise. It uses logical AND operation for finding the *SC* from frequent 2-itemset to frequent *n*-itemsets until it is not empty. It also uses the additional information in the Boolean matrix namely "number of iterations" to control the number of iterations for candidate generation. Finally, they have demonstrated that the FPMBM is more efficient and scalable than the existing ones.

In [MR,16], the authors have created an algorithm called Matrix-Over-Apriori (MOA) by using elementary matrix and AND operation. They compared MOA with all other existing techniques for ARM and proved that MOA is scalable, precise, simple, clear, easy to implement, and also reduces the memory and time requirements than the existing ones. A new method for Mining Frequent Itemsets with Weights over a Data Stream using Inverted Matrix called MFIWDSIM has been proposed by L.N.Hung and T.N.T.Thu [HT,16]. In this, the data stream is converted into an inverted matrix and saved in the computer disks and mines them many times with different support thresholds and alternative minimum weights. With the analysis and evaluation, they proved that the MFIWDSIM is better than WSWFP-stream.

In [NJGC^{+},17], the authors have proposed a modified Apriori algorithm named Frequent Matrix Apriori (FMA), for reducing the time complexity. In that, the database information is stored in the frequent matrix by scanning it only once and then the matrix is discretized using minimum support parameters and the most frequent itemsets are found recursively by scanning the discretized dataset. By the theoretical and experimental way, the authors have proved that FMA is more efficient than the original AA in terms of time.

In [KK,17], the authors have presented a new top-down approach called MB-MFIM by using a transaction Boolean matrix. In this method, the maximal frequent itemsets are directly generated without the help of a subset based on the

compressed matrix. The proposed algorithm provides a better result than the Maximal Frequent Itemset First (MFIF) algorithm with datasets of different sizes and thresholds.

An improved Apriori algorithm based on relational algebra theory has been proposed in [ZZ,17]. The relationship matrix and correlation operations are obtained by Optimization Relation Association Rule. The database is scanned only once with a relation matrix which reduces the running time of the algorithm to mine frequent itemsets. The simulation results showed that the improved algorithm works more efficiently than the existing one.

Judith Pavón et al. [PVG,06] have introduced a method called Matrix-Apriori to increase the speed of finding frequent itemsets. It creates a Boolean matrix MFI by scanning the transactional database which contains the frequent 1-itemset. The vector STE maintains the *SC* of the candidate itemset. To accelerate the search of frequent patterns, the first row of MFI writes the indexes. For producing frequent patterns, a conditional pattern generation method was used in this method and proved that it outperforms Apriori and FP-Growth algorithms.

In [Lan,18], the author has introduced an improved matrix pruning and weight analysis Apriori algorithm by using matrix compression and weight analysis algorithms as reference. This algorithm constructs the Boolean transaction matrix and removes infrequent itemset and generates a new candidate itemset. Then it calculates the item's weight, transaction's weight, and weight support. With the experimental results, the author has proved that the improved Apriori algorithm not only reduces the number of repeated scans of the database but also improves the efficiency of data correlation mining.

A new incremental ARM algorithm called FBCM has been proposed in [ZOKL^{+},19] by combining the Fast Update Pruning (FUP) algorithm with a compressed Boolean matrix to suit the dynamically changing data. It requires only a single scan of the database and provides support for incremental databases. While scanning, it obtains two compressible Boolean matrices and applies ARM to those matrices. When compared with existing algorithms, it improved the computational efficiency of incremental ARM and proved that it is suitable for knowledge discovery in the edge nodes of cloud systems.

The authors in [XJW,19], have introduced a modified Apriori algorithm based on the Boolean matrix and weight function. In this algorithm, they have trimmed the duplicate transactions by adding weight rows to the matrix and also compressed the matrix to reduce storage space. Self-join and intersection operations were used to obtain *k*-frequent itemsets. They have paralleled it using Hadoop and each map activity finds the frequent itemset for the subset of the large matrix which shortens the processing time in the big data environment.

Research has been contributed by Sun et al. by applying the prefixed-itemset storage and the compression matrix to optimize the connection, pruning, support counting steps, and transaction storage mode of the Apriori algorithm. It uses an intersection strategy for determining *SC*. The optimized Apriori is based on the MapReduce technique for massive data and they have proved that the optimized Apriori outperforms others [SL,20].

The authors in [SS,20], have presented a novel algorithm for generating frequent patterns from a large dataset. Initially, they transformed the transactional dataset into a Boolean matrix to generate a 1-frequent itemset matrix, and then it is

divided into multiple loads based on the available nodes in the system. To discover all frequent itemsets, they have used AND operation on individual load and proved from the experiment that the computational time and consumption of memory reduced.

The authors have initiated an algorithm for finding frequent itemsets based on the transaction matrix, itemset matrix, and item index list in [SJ,20]. It reduces the number of database scans to one and avoids frequent I/O operation by compressing the matrix and then performing bitwise AND operation on the compressed matrix. The frequent itemsets were generated using the itemset count and index list. The main advantage of this method is that no candidate itemsets are generated and outperforms the existing method.

2.4 Works Related to Vertical Data Format

A novel VDF representation called Diffset has been developed by the authors in [ZG,03], which keep track of the differences in the *TID*s of a candidate pattern and from which it generates frequent patterns. The method cut down the size of memory required to store intermediate results and also increased performance significantly.

Y. M. Guo et al. [GW,10] have initiated a new algorithm for mining frequent itemsets with VDF. It only needs a single scan of the entire database and uses the AND operation for finding the frequent itemsets. Furthermore, it was demonstrated that the algorithm requires less storage and enhances mining efficiency.

In [KSK,12], the authors have presented a VDSRP method to generate a complete set of regular patterns over a data stream at a user given regularity threshold using a sliding window and VDF. It has been proved that the proposed method outperforms both in execution and memory consumption.

The authors in [VV,13] have introduced a Parallel Regular Frequent Pattern (PRFP) method to find out the regular-frequent patterns from large databases using VDF format and proved from the experiments that the algorithm reduced the number of database scans, I/O cost and inter-process communication.

In [AR,14], a new Rehashing Based Frequent Itemset (RBFI) generation algorithm of the VDF for the transactional database has been proposed. Rehashing has been introduced to avoid hash collision and secondary clustering problems in hashing. It was proved that RBFI provides better performance than Apriori and Hash-based algorithms.

In [IMA,15], a method called Vertical Boolean Mining (VBM) has been introduced to eliminate the pitfalls of vertical mining by compressing the bit vectors of frequent itemsets. It intersects two compressed bit vectors without requiring a time-consuming decompression step. They found that the VBM is superior to both Apriori and classical vertical ARM in terms of time and memory usage.

Jen, T. Y., et al. have created a novel vertical format based parallel method for finding frequent patterns called Apriori_V with MapReduce platform. They proved that it provides a significant improvement in reducing the number of operations and decreasing computational complexity [JMG,16].

A Vertical Format Frequent Mining (VFFM) algorithm has been proposed in [GSG,16] to find frequent items from the database. It first transforms the database into VDF, as <*item*, {*transaction-id*}> and finds the candidate itemsets after the first scan of the transactional database. The *SC* of each (*k*+1)-candidate itemsets is counted by the intersection of every pair of frequent single items instead of the database scan. It was proved by them that the VFFM is efficient when compared with AA, FUP and sampling method.

In [TC,16], a tokenization based approach for optimizing enhancing the Apriori algorithm has been proposed. Ravikiran, D., et. al, have proposed a new model called RCP to mine regular sort of crimes in crime databases using VDF which requires only one database scan. From the experimental results, they proved that RCP is more efficient than the existing RPtree [RS_2,16]. In [Sin,16], the authors have focused on the various FPM techniques, their challenges in static and stream data environments.

Subashini et al. [SK,19] have studied ARM methods in HDF and VDF approaches viz., Apriori, APRIORITID, APRIORI_RARE and APRIORIRARE_TID. They analyzed the pros and cons of each technique.

2.5 Works Related to Eclat

In [AR,14], the authors have built the enhanced versions of Apriori and Eclat algorithms. In these enhanced versions, the authors have used individual thresholds for each itemset and proved that the enhanced-AA performs best when compared with the Enhanced-Eclat Algorithm.

In [MYZL,16], the authors have presented an improved version of Eclat called the Eclat-growth algorithm using an increased search strategy. For reducing the runtime in generating an intersection of two itemsets and support degree calculation, a BSRI (Boolean array Setting and Retrieval by Indexes of transactions) method has been introduced. It has been proved by them that the Eclat-growth outperforms Eclat, Eclat-diffsets, Eclat-opt and hEclat in mining association rules.

An enhanced Apriori and Eclat have been introduced in [SV,17], in which different thresholds are maintained for each itemset. They compared different sizes of dataset and items and proved from the experiment that the enhanced-Apriori

algorithm is better than the Enhanced-Eclat algorithm in terms of the number of frequent items and rules.

2.6 Works Related to GPUs

W. Fang et al. [FLXH+,09] have introduced two implementations for Apriori using GPUs with Single Instruction, Multiple Data (SIMD) architectures. Both methods use a bitmap data structure. To prevent the data transformation between the GPU and CPU memory, the first one was executed using GPU. The second one uses both the CPU and GPU for processing with trie structure. They proved that both implementations speed up the processing than the classical Apriori algorithm.

The authors J. Zhou et al. have designed [ZYW,10] a GPU-based Apriori algorithm with OpenGL to accelerate ARM and proved that it is better than the traditional ones.

S. M. Fakhrahmad et al. [FD,11] have developed different parallel versions of a novel sequential mining algorithm for finding frequent itemsets. The approaches are: i) allocating a processor to each partition, ii) allocating a processor to each column, and iii) allocating the k^{th} processor to mine the [FD,11] k^{th}-itemsets.

A compressed bit matrix-based parallel algorithm for exploring frequent itemsets has been introduced by Zong-Yu et al., which uses both bottom-up and top-down approaches for efficient pruning [ZY,12]. It also uses OpenMP's parallel multithreaded, dynamic scheduling approach to extract frequent itemsets. Finally, they demonstrated that this approach reduced memory space, I/O overhead with a single database scan compared to the Apriori algorithm.

Authors in [HYZH+,13] have suggested a novel algorithm, namely Accelerating Parallel Frequent Itemset Mining on Graphics Processors with

Sorting (APFMS). This parallel frequent itemset mining employs GPUs in the process of mining. GPUs speed up the process using the OpenCL platform and proved that the APFMS outperforms the previous computation time-based methods.

William Albert et al. [AFB[a],14], and it is based on the parallel processing nature of GPU. In the proposed method, a bitset representation was used for parallel processing and proved that the HSApriori is faster than traditional HorgeltAprirori.

M. Tiwary et al. [TSM,14] developed a parallel Apriori Map Reduce model by employing high-performance GPU to address the issues of Apriori. In this, every node in a Hadoop cluster has a GPU attached to it. They also employed NVIDIA's GPU, as well as JCUDA and JNI, to complete the integration. From the results, they finalized that the proposed method requires less run time. The disadvantage of the algorithm is that an additional hardware cost is linked with the GPUs in each node in the Hadoop cluster.

In [QGYH,14], the authors have designed a Spark-based parallel Apriori algorithm called YAFIM (Yet Another Frequent Itemset Mining) and revealed that the YAFIM is faster than the Apriori's MapReduce implementation by 18 times.

To remove the limitations in the traditional cluster-based map-reduce, J. Li et al. [LSHW,15] have designed a multi-GPU based parallel Apriori algorithm to accelerate the calculation process of Apriori. It has been initiated especially to mine association rules in medical data. The analytical results have proved that the proposed method significantly improves the execution speed with a lower cost for medical data.

A novel method called CGMM to suit both sparse and dense datasets has been introduced by L. Vu et al. [VA,15]. To enhance the speediness of the FPM process, the CPU is combined with GPU. In this method, the CPU uses the FP-tree data structure to perform mining, and the GPU converts the data to bit vectors.

They demonstrated that the performance of CGMM is faster when compared with the existing sequential FPM and GPApriori by testing with AMD CPUs and NVIDIA GPU.

A new multi-core based parallel mining algorithm for finding frequent itemsets has been presented in [HL,15] using LINQ queries. It decomposes the transactional database into smaller datasets known as conditional patterns. Many threads ran concurrently on a multi-core computing system, one for each conditional pattern. They proved that the algorithm is faster by 2*x* and 4*x* times than the fast Eclat and FP-growth algorithms, respectively.

Y. Li et al. [LXYC,17] have developed a GPU-based algorithm called Multi-level Vertical Closed FIM. It uses a multi-layer vertical data structure to reduce memory usage. The implementation is being accelerated with GPU to achieve high-speed computation, mainly for large sparse datasets.

A Dynamic Queue and Deep Parallel (D2P) Apriori algorithm were generated by Y. Wang et al. in [WXXS,18]. They parallelized the candidate generation task with a dynamic bitmap queue and Graph-join. It also uses a vertical bitmap structure with low-latency memory on GPU. They found that the D2P-Apriori is faster by 23 times than modern CPU methods.

A fast GPU-based frequent itemset mining algorithm for massive datasets called GMiner has been introduced in [CHK,18] to overcome the limitations of various parallelism methods viz., multi-core CPU, multiple machines and many-core GPU, particularly the workload skewness. It extracts the patterns fastly from the enumeration tree by using the computational power of GPU. From the experimentation, they showed that the GMiner is better than the existing sequential and parallel methods.

The authors Y. Djenouri et al. [DDBC,19] have created three High-Performance Computing (HPC)-based versions of Single Scan (SS) for FIM viz., GSS, CSS, and CGSS. The GSS, CSS, and CGSS have been implemented by SS with GPU, cluster architecture, and GPU with multiple cluster nodes. They also proposed three methods for reducing GPU thread divergence and cluster load balancing. Experiments have shown that the CGSS outperforms the SS, GSS, and CSS in terms of speed.

In [GLFC^{+},19], the authors have reviewed the works related to Parallel Sequential Pattern Mining (PSPM), viz., partition-based, Apriori-based, pattern growth-based, and hybridized algorithms for PSPM. They also reviewed the open-source software utilized in PSPM. Further, they summarized the issues and uses of PSPM on big data.

In [HTDV,19], the authors have proposed an FPM algorithm with a multi-core processor and Multiple Minimum Support called MMS-FPM. It quickly generated frequent patterns. It has been designed mainly to solve rare item problems. They have proved that the MMS-FPM is superior to MSApriori and also scalable.

2.7 Observations and Limitations of the Existing Literature

From the existing literature, the following observations were identified which paves the way for the researcher to select the research problem.

i. Some of the existing methods generate more candidate itemset and requires much disk access
ii. Though the VDF approach restricts the database scan to one, the memory required for storing *TID*s for each item is huge

iii. Some of the data structures utilized in the existing literature may generate a reduced set of candidate itemsets but requires more memory

iv. Some of the pattern generation methods may need more execution time in generating the frequent patterns for the transactional databases

v. There are still issues related to data size and scalability

From the above observations, it has been identified that there is always a need for speedy algorithms for frequent pattern generation with a minimum amount of time and memory usage. Thus, this research work focuses on developing novel FPM algorithms with the compact data structure called jagged array by creating novel pattern generation approaches using multithreading and GPU usage with the VDF approach.

Chapter - 3

RISOTTO: A NOVEL HYBRID APPROACH FOR ENHANCING CLASSICAL APRIORI ALGORITHM

CHAPTER - 3

RISOTTO: A NOVEL HYBRID APPROACH FOR ENHANCING CLASSICAL APRIORI ALGORITHM

The POSITIVE THINKER sees the INVISIBLE, feels the INTANGIBLE,
and achieves the IMPOSSIBLE

--Winston Churchill

3.1 Background

Association Rule Mining (ARM) is a successful technique for finding relations between data items in databases. Finding frequent itemsets is one of the computationally crucial steps in the task of mining association rules. The Apriori is one of the most important algorithms for finding frequent itemsets. The main challenge in classical Apriori is that the mining often needs to generate a huge number of candidate itemsets and require more database scans, increasing time and decreasing efficiency. It also increases the I/O cost and requires more memory. To eradicate these issues, a lot of improvements to Apriori have been proposed in the literature.

Research in improving the Apriori is a common issue and is an ongoing research topic these days. A refinement to the Apriori, which uses a Data Structure (DS) called prefixed-itemset for candidate itemset generation and Vertical Data Format (VDF) approaches, has been proposed in the literature. Prefixed-itemset storage shortens the time for generating candidate itemsets but still needs more database scans as in Apriori, and VDF scans the database only once. RISOTTO, a novel hybrid approach for generating frequent patterns has been contributed to this research by considering these advantages. It combines both the prefixed-itemset storage structure and VDF.

The proposed work minimizes the number of database scans to one and reduces the time needed for candidate itemset generation.

3.2 Prefixed-itemset Storage Structure

It is a new way of storing itemsets [YZ,16] that uses <Prefix-key, Values> pair for each itemset. The Prefix-key column stores the (k-1)-items in the k^{th} itemset, and the last item in k^{th} itemset are stored in the Values column. If there is no prefix for an itemset, *NULL* is stored in the prefix-key. Suppose if the 1-itemset contains {A, B, C, D, E} and 2-itemsets contains {AB, AC, AE, BC, BD, BE}, then the prefixed-itemset based storage structure for the same is illustrated in Table 3.1.

Table 3.1 Prefixed-Itemset Storage Structure

Itemset	Prefix-key	Values
1-itemset	NULL	{A,B,C,D,E}
2-itemset	A	{B,C,E}
	B	{C,D,E}

After the k-itemsets are stored in the prefixed-itemset storage, in the joining step, the (k+1)-itemset are generated by first joining or connecting the values of k-itemset, and then the key values are prefixed with each (k+1)-itemset which forms C_{k+1}. In the pruning step, the (k+1)-itemset which does not satisfy the Apriori property is removed from C_{k+1}.

3.3 Vertical Data Format

In general, there are two ways in which a transactional database can be represented in frequent pattern mining algorithms. They are Horizontal Data Format (HDF) and VDF. In VDF the data can be expressed in {*item* - *TID_set*} notation where the *item* is the name of the item in the database and *TID_set* is the set of transactions that the *item* belongs to.

This method first transforms the HDF dataset into VDF by scanning the dataset once, which forms candidate 1-itemset. Among them, the itemset that satisfies the minimum support (δ) will be considered as a frequent 1-itemset. It is noted that the support count for an itemset is the length of the *TID_set*. Starting with k=2, the frequent k-itemsets can be used to construct the candidate (k+1) itemsets based on the Apriori property. The *TID_set* for the candidate (k+1) is computed by intersecting the *TID_sets* of the corresponding item in k-itemsets. This process is repeated by incrementing k by one until no frequent itemsets or candidate itemsets can be found.

The main advantage of VDF is that there is no database scan is required for finding the support of (k+1)-itemsets because the *TID_set* of k-itemset holds the complete information for finding such support. The disadvantage is that if the *TID_set* is long, it will take substantial memory space and more computation time to intersect the long sets.

3.4 Proposed Methodology

It combines both prefixed-itemset based storage structure [YZ,16] and the VDF approach [SNM,15] for enhancing the performance of the classical Apriori algorithm in terms of time and the number of database scans. It progresses as follows:

In the first step, the algorithm finds the candidate 1-itemset (C_1) from the transactional database by scanning it once as in classical Apriori, and it is transformed into VDF, i.e. it maintains the *TID_set* in which the frequent 1-itemset occurs along with the Support Count (*SC*) or Total Number of Transactions (*TNT*). The frequent 1-itemset (L_1) is constructed from C_1 by removing the items whose *SC* is less than δ. After finding L_1, the information regarding this is stored in the new DS called

prefixed-itemset based storage, as in Table 3.1. The prefix for frequent 1-itemset is always *NULL*, and the values are the items in L_1. In general, the frequent *k*-itemset where k=2,3,…,n contains (*k-1*)-items as prefix-key (LK_k) and the last item as the value (LV_k).

In the second step, the values in frequent 1-itemset in the prefixed-itemset based storage LV_1 is joined by itself ($LV_1 \bowtie LV_1$) instead $L_1 \bowtie L_1$ and the items which do not satisfy the Apriori property is removed, and then they are combined with the prefix-key which forms C_2. To improve the efficiency by reducing the search space by considering the Apriori property, i.e. all nonempty subsets of a frequent itemset must also be frequent. The *SC* for the items in C_2 is calculated just by performing the intersection of the *TID_set* in L_1 instead of scanning the database as in classical Apriori, which minimizes the database scans. From C_2, L_2 is formed by removing those elements from C_2 whose $SC<\delta$. Similar to the previous step, the frequent 2-itemsets are stored in the prefixed-itemset based storage with the appropriate prefix-key and values. The second step is repeated with k=3,4,5,…,n until there are no more candidate itemsets found.

The proposed approach is named RISOTTO[1], which is abbreviated by taking the boldface uppercase letters from the phrase "p**R**efixed **I**tem**S**et st**O**rage ver**T**ical da**T**a f**O**rmat". The algorithm for RISOTTO is shown below. The workflow of RISOTTO is illustrated in Figure 3.1.

[1]**P.Sumathi**, S.Murugan, "RISOTTO - A Novel Hybrid Approach for Enhancing Classical Apriori Algorithm", International Journal of Scientific Research in Computer Science Applications and Management Studies, ISSN: 2319 – 1953, Vol. 7, No. 5, September 2018 (**UGC Approved Journal**).

Algorithm 3.1: RISOTTO - An algorithm for finding frequent itemsets

Input:

- A dataset D with n transactions;
- δ - minimum support threshold.

Output:

- Frequent itemsets (L) in D.

Method:

(1) $L \leftarrow \emptyset$;

(2) $C_1 \leftarrow$ scan D and generate candidate 1-itemsets;

(3) $L_1 \leftarrow$ generate frequent 1-itemsets based on δ;

(4) $L \leftarrow L \cup L_1$;

(5) $PIDS \leftarrow$ create a prefixed-itemset storage DS;

(6) $PIDS(LK_1) \leftarrow NULL$;

(7) $PIDS(LV_1) \leftarrow$ items in L_1;

(8) **for** (k=2; $L_{k-1} \neq \emptyset$; k++) **do**

begin

$C_{k\_init} \leftarrow PIDS(LV_{k-1}) \bowtie PIDS(LV_{k-1})$;

$C_{k\_init} \leftarrow$ prune C_{k_init};

$C_k \leftarrow PIDS(LK_{k-1}) \bowtie C_{k\_init}$;

$L_k \leftarrow \{C_k \mid SC(C_k) \geq \delta\}$;

$PIDS(LK_k) \leftarrow (k\text{-}1)$-items in L_k;

$PIDS(LV_k) \leftarrow k^{th}$ item in L_k;

$L \leftarrow L \cup L_k$

endfor

(9) return L;

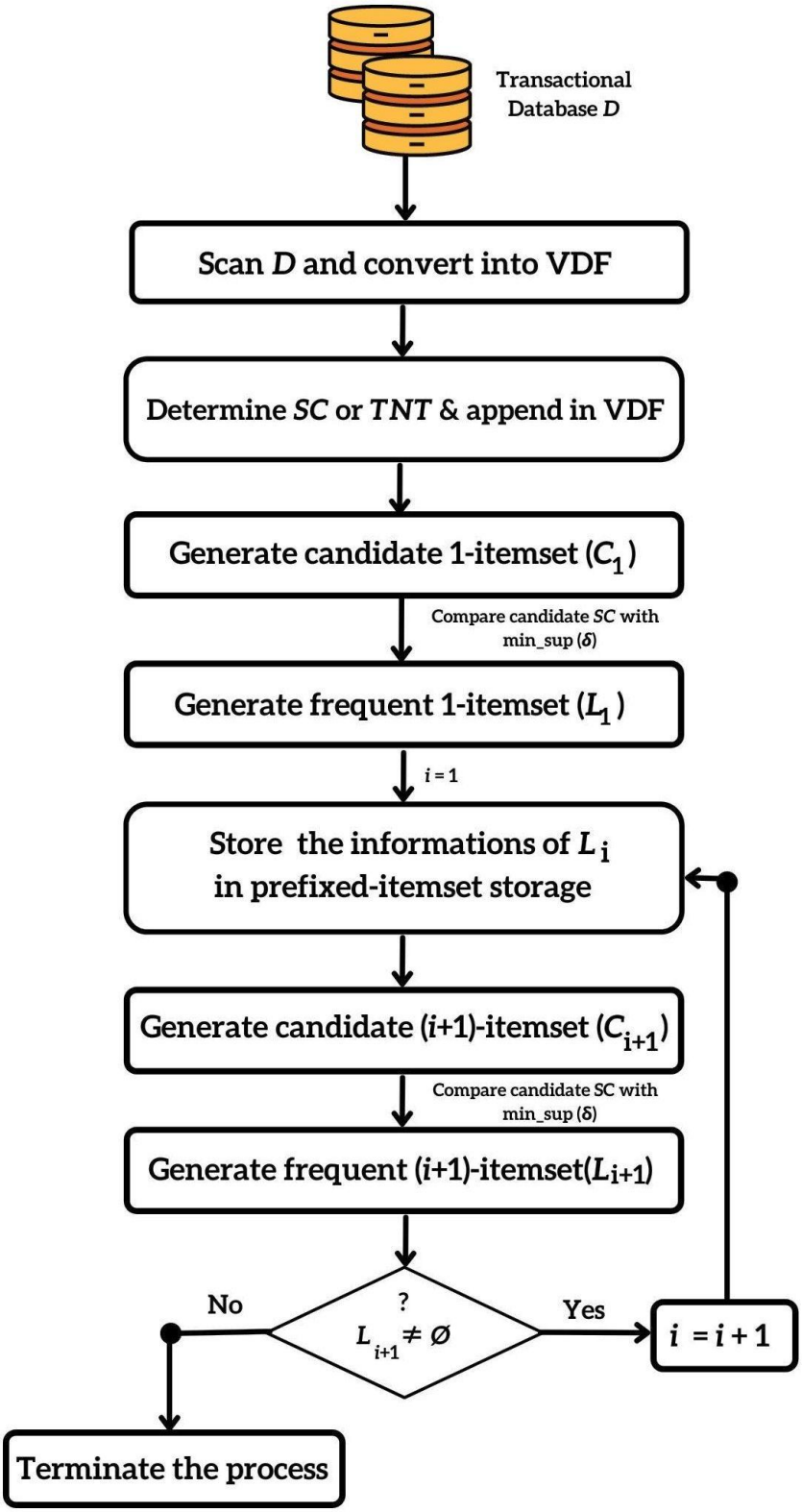


Figure 3.1 Workflow of RISOTTO

The main advantage of this hybrid approach is that it restricts the database scan to one because for finding the *SC* for frequent *k*-itemsets where *k*=2,3,4,…,*n* the database need not be scanned, and it is found by set intersection method from the *TID_sets* of L_{k-1} which in turn minimizes the I/O cost. Using the prefixed-itemset storage, the number of candidate *k*-itemsets generated is reduced when compared with the classical Apriori algorithm because it uses the values of the prefix-key items stored for joining rather than the values in L_k.

3.4.1 Illustration by an Example

A sample transactional database *D* shown in Table 3.2 has been taken for illustrating the proposed methodology. It consists of ten transactions. Each transaction comprises Transaction ID (*TID*) and items bought from the business enterprise a, b, c, d, e, f, g, h, i, k, p, and m. Let the $\delta = 6$. The frequent 1-itemset is computed as in the classical Apriori but the L_1 in the proposed method contains *TID_set* and *TNT* or *SC*. The computation of C_1 and L_1 are shown in Table 3.3 and 3.4, respectively.

Table 3.2 Transactional Database D

TID	Items Purchased
0	c, d, e, g, h, i, k, p, m
1	b, e, f, g, h, i, p, m
2	c, e, m
3	a, b, c, d, e, f, g, i, p
4	a, b, c, d, e, p
5	a ,b ,c, d, f, h, p
6	b, e, f, h, i, p, m
7	a, c, d, e, k, p, m
8	a, c , d, e, f, i, p, m
9	a, c, d, e, f, h, i, p, m

After computing L_1 with one database scan, it is stored in prefixed-itemset storage with the values viz., 1-itemset in Itemset column, *NULL* in Prefix-key column and the frequent 1-itemset, i.e. {a, c, d, e, f, i, m, p} in Values column as shown in

Table 3.5. Next {a, c, d, e, f, i, m, p} ⋈ {a, c, d, e, f, i, m, p} is performed and it is {ac, ad, ae, af, ai, am, ap, cd, ce, cf, ci, cm, cp, de, df, di, dm, dp, ef, ei, em, ep, fi, fm, fp, im, ip, mp} and all satisfies the Apriori property and forms C_2.

Table 3.3 Computation of C_1

Item	TID_set	TNT or SC
a	{3, 4, 5, 7, 8, 9}	6
b	{1, 3, 4, 5, 6}	5
c	{0, 2, 3, 4, 5, 7, 8, 9}	8
d	{0, 3, 4, 5, 7, 8, 9}	7
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}	9
f	{1, 3, 5, 6, 8, 9}	6
g	{0, 1, 3}	3
h	{0, 1, 5, 6, 9}	5
i	{0, 1, 3, 6, 8, 9}	6
k	{0, 7}	2
m	{0, 1, 2, 6, 7, 8, 9}	7
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}	9

Table 3.4 Computation of L_1

Item	TID_set	TNT or SC
a	{3, 4, 5, 7, 8, 9}	6
c	{0, 2, 3, 4, 5, 7, 8, 9}	8
d	{0, 3, 4, 5, 7, 8, 9}	7
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}	9
f	{1, 3, 5, 6, 8, 9}	6
i	{0, 1, 3, 6, 8, 9}	6
m	{0, 1, 2, 6, 7, 8, 9}	7
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}	9

Table 3.5 Prefixed-Itemset Storage with frequent 1-itemset

Itemset	Prefix-key	Values
1-itemset	*NULL*	{a, c, d, e , f , i, m, p}

The *TID_set* of an item say ac is calculated by intersecting the *TID_sets* of the items a and c respectively.

TID_set of {ac}={3,4,5,7,8,9}∩{0,2,3,4,5,7,8,9}

={3,4,5,7,8,9}

The *SC* for each item is determined by counting the number of items in *TID_set*.

SC of {ac} = length({3,4,5,7,8,9})=6

Similarly, the *SC* for other items in C_2 is computed, and it is shown in Table 3.6. Out of these items, only the items ac, ad, ap, cd, ce, cp, de, dp, ei, em, ep, fp, ip and mp satisfy δ hence forms L_2, and it is shown in Table 3.7.

Table 3.6 Computation of C_2

Itemset	TID_set (by set intersection)	SC	Itemset	TID_set (by set intersection)	SC
{ac}	{3,4,5,7,8,9}	6	{df}	{3,5,8,9}	4
{ad}	{3,4,5,7,8,9}	6	{di}	{0,3,8,9}	4
{ae}	{3,4,7,8,9}	5	{dm}	{0,7,8,9}	4
{af}	{3,5,8,9}	4	{dp}	{0,3,4,5,7,8,9}	7
{ai}	{3,8,9}	3	{ef}	{1,3,6,8,9}	5
{am}	{7,8,9}	3	{ei}	{0,1,3,6,8,9}	6
{ap}	{3,4,5,7,8,9}	6	{em}	{0,1,2,6,7,8,9}	7
{cd}	{0,3,4,5,7,8,9}	7	{ep}	{0,1,3,4,6,7,8,9}	8
{ce}	{0,2,3,4,7,8,9}	7	{fi}	{1,3,6,8,9}	5
{cf}	{3,5,8,9}	4	{fm}	{1,6,8,9}	4
{ci}	{0,3,8,9}	4	{fp}	{1,3,5,6,8,9}	6
{cm}	{0,2,7,8,9}	5	{im}	{0,1,6,8,9}	5
{cp}	{0,3,4,5,7,8,9}	7	{ip}	{0,1,3,6,8,9}	6
{de}	{0,3,4,7,8,9}	6	{mp}	{0,1,6,7,8,9}	6

Similar to frequent 1-itemset, the frequent 2-itemsets are appended to prefixed-itemset storage. In L_2, the items ac, ad and ap have the common prefix a and values are {c,d,p}. Similarly, the items cd, ce and cp have the common prefix c and values are {d,e,p}, the items {de,dp} has the common prefix d and values are {e,p}, the items {ei,em,ep} has the common prefix e and values are {i,m,p}, the items fp has the prefix f and value is p, the items ip has the prefix i and value is p and the item mp has the prefix m and value is p. The original prefixed-itemset storage after appending frequent 2-itemset is shown in Table 3.8.

Table 3.7 Computation of L_2

Itemset	TID_set (by set intersection)	SC
{ac}	{3,4,5,7,8,9}	6
{ad}	{3,4,5,7,8,9}	6
{ap}	{3,4,5,7,8,9}	6
{cd}	{0,3,4,5,7,8,9}	7
{ce}	{0,2,3,4,7,8,9}	7
{cp}	{0,3,4,5,7,8,9}	7
{de}	{0,3,4,7,8,9}	6
{dp}	{0,3,4,5,7,8,9}	7
{ei}	{0,1,3,6,8,9}	6
{em}	{0,1,2,6,7,8,9}	7
{ep}	{0,1,3,4,6,7,8,9}	8
{fp}	{1,3,5,6,8,9}	6
{ip}	{0,1,3,6,8,9}	6
{mp}	{0,1,6,7,8,9}	6

Table 3.8 The Original Prefixed-Itemset Storage after Appending frequent 2-itemset

Itemset	Prefix-key	Values
1-itemset	*NULL*	{a, c, d, e , f , i, m, p}
2-itemset	a	{c, d, p}
	c	{d, e, p}
	d	{e, p}
	e	{i, m, p}
	f	{p}
	i	{p}
	m	{p}

But for the prefix-keys f, i, and m the values column contains only one value. With one value, there is no possibility of generating a frequent 3-itemset. So, they are not stored in the prefixed-itemset storage of the RISOTTO algorithm which further helps to reduce the time and storage. The prefixed-itemset storage after appending frequent 2-itemset in RISOTTO is shown in Table 3.9. To find candidate 3-itemset, the values of frequent 2-itemset in prefixed-itemset storage is considered.

Table 3.9 The Prefixed-Itemset Storage after Appending frequent 2-itemset in RISOTTO

Itemset	Prefix-key	Values
1-itemset	*NULL*	{a, c, d, e, f, i, m, p}
2-itemset	a	{c, d, p}
	c	{d, e, p}
	d	{e, p}
	e	{i, m, p}

From Table 3.9, first {c,d,p} ⋈ {c,d,p} is calculated and it is {cd,cp,dp} and all the item satisfies the Apriori property so each item is prefixed with the prefix-key a which gives {acd,acp,adp}. Next {d,e, p} ⋈ {d,e,p} is calculated and it is {de,dp,ep} and each item is prefixed with the prefix-key c which gives {cde,cdp,cep} because the items {de,dp,ep} satisfies Apriori property. Similarly, for the values {e,p} and {i,m,p} the combinations were generated and forms {dep} and {eim, eip, emp} as candidate 3-itemset. After determining the candidate 3-itemset, the transactions in which the combination occurs and *SC* is calculated as

TID_set of {acd} = {3,4,5,7,8,9}∩{0,3,4,5,7,8,9} = {3,4,5,7,8,9}

SC of {acd} = length({3,4,5,7,8,9}) = 6

Likewise, it is calculated for the remaining candidate 3-itemset, and it is shown in Table 3.10.

Table 3.10 Computation of C_3

Itemset	TID_set (by set intersection)	SC
{acd}	{3,4,5,7,8,9}	6
{acp}	{3,4,5,7,8,9}	6
{adp}	{3,4,5,7,8,9}	6
{cde}	{0,3,4,7,8,9}	6
{cdp}	{0,3,4,5,7,8,9}	7
{cep}	{0,3,4,7,8,9}	6
{dep}	{0,3,4,7,8,9}	6
{eim}	{0,1,6,8,9}	5
{eip}	{0,1,3,6,8,9}	6
{emp}	{0,1,6,7,8,9}	6

From Table 3.10, the item {eim} does not satisfy δ so it is removed from candidate 3-itemset and L_3 shown in Table 3.11 is formed. Like frequent 1- and 2-itemsets, the frequent 3-itemsets are also appended into prefixed-itemset storage by separating them into prefix-key and values, as shown in Table 3.12. The prefix-keys in Table 3.12 such as ad, ce, de, ei and em contains only one item in the values column. So as in the 2-itemset, the entries for those prefix-keys will not be saved in the prefixed-itemset storage of RISOTTO.

Table 3.11 Computation of L_3

Itemset	TID_set (by set intersection)	SC
{acd}	{3,4,5,7,8,9}	6
{acp}	{3,4,5,7,8,9}	6
{adp}	{3,4,5,7,8,9}	6
{cde}	{0,3,4,7,8,9}	6
{cdp}	{0,3,4,5,7,8,9}	7
{cep}	{0,3,4,7,8,9}	6
{dep}	{0,3,4,7,8,9}	6
{eip}	{0,1,3,6,8,9}	6
{emp}	{0,1,6,7,8,9}	6

Table 3.12 The Original Prefixed-Itemset Storage after Appending frequent 3-itemset

Itemset	Prefix-key	Values
1-itemset	*NULL*	{a, c, d, e , f , i, m, p}
2-itemset	a	{c, d, p}
	c	{d, e, p}
	d	{e, p}
	e	{i, m, p}
	f	{p}
	i	{p}
	m	{p}
3-itemset	ac	{d, p}
	ad	{p}
	cd	{e, p}
	ce	{p}
	de	{p}
	ei	{p}
	em	{p}

Table 3.13 shows the prefixed-itemset storage after appending frequent 3-itemsets in RISOTTO.

Table 3.13 The Prefixed-Itemset Storage after Appending frequent 3-itemset in RISOTTO

Itemset	Prefix-key	Values
1-itemset	*NULL*	{a, c, d, e , f , i, m, p}
2-itemset	a	{c, d, p}
	c	{d, e, p}
	d	{e, p}
	e	{i, m, p}
3-itemset	ac	{d, p}
	cd	{e, p}

From the above table, {d,p}⋈{d,p} is performed and it gives {dp} which is prefixed with the prefix-key {ac} which forms {acdp} as the first candidate 4-itemset. Likewise, it is performed for other values for the frequent 3-itemset in prefixed-itemset storage. The candidate 4-itemset C_4 for the sample example is shown in Table 3.14. All the candidate 4-itemsets in Table 3.14 satisfies the minimum support and forms L_4, as shown in Table 3.15.

Table 3.14 Computation of C_4

Itemset	TID_set (by set intersection)	SC
{acdp}	{3,4,5,7,8,9}	6
{cdep}	{0,3,4,7,8,9}	6

Table 3.15 Computation of L_4

Itemset	TID_set (by set intersection)	SC
{acdp}	{3,4,5,7,8,9}	6
{cdep}	{0,3,4,7,8,9}	6

Similarly, the frequent 4-itemset is also appended in the original prefixed-itemset storage with appropriate prefix-key and values, and it is shown in Table 3.16.

Table 3.16 The Original Prefixed-Itemset Storage after Appending frequent 4-itemset

Itemset	Prefix-key	Values
1-itemset	*NULL*	{a, c, d, e, f, i, m, p}
2-itemset	a	{c, d, p}
	c	{d, e, p}
	d	{e, p}
	e	{i, m, p}
	f	{p}
	i	{p}
	m	{p}
3-itemset	ac	{d,p}
	ad	{p}
	cd	{e,p}
	ce	{p}
	de	{p}
	ei	{p}
	em	{p}
4-itemset	acd	{p}
	cde	{p}

Table 3.17 shows the prefixed-itemset storage after appending frequent 4-itemset in the RISOTTO algorithm. All the values in the frequent 4-itemset of the original prefixed-itemset storage contain only one value. They will not be stored in the RISOTTO algorithm as it is impossible to form any candidate 5-itemset. Therefore, the candidate 5-itemset is *Ø*, and the algorithm terminates.

Table 3.17 The Prefixed-Itemset Storage after Appending frequent 4-itemset in RISOTTO

Itemset	Prefix-key	Values
1-itemset	*NULL*	{a, c, d, e , f , i, m, p}
2-itemset	a	{c, d, p}
	c	{d, e, p}
	d	{e, p}
	e	{i, m, p}
3-itemset	ac	{d, p}
	cd	{e, p}

It is noted that the prefixed-itemset storage after appending frequent 4-itemset remains the same as the prefixed-itemset storage after appending frequent 3-itemset and the RISOTTO algorithm terminates.

3.5 Experimental Results and Discussion

To analyze the effectiveness of the proposed method, an empirical study has been performed using the datasets shown in Table 1.4. The algorithms were implemented in Python. The runtime performance of RISOTTO is compared with prefixed-itemset storage and VDF for the four datasets with different δ is carried out, and it is tabulated in Table 3.18. The δ varied from 20% to 70%. Figures 3.2 to 3.5 show the graphical representation of the runtime comparison between the algorithms viz., prefixed-itemset storage, VDF, and the proposed RISOTTO algorithm for the datasets, namely chess, mushroom, t25i10d10k and c20d10k, respectively.

From Table 3.18 and figures 3.2 through 3.5, it is observed that the RISOTTO outperforms the existing algorithms, namely prefixed-itemset storage and VDF, i.e. the runtime required is reduced from 22.0163 to 13.5594 seconds on an average. The reason is that the number of candidate itemsets produced in RISOTTO is less when compared to VDF. Also, it uses the values in the prefixed-itemset storage for creating candidate itemsets whose length is greater than one at any point of time. It minimizes the database scan to one compared with the prefixed-itemset storage method because RISOTTO uses VDF, which maintains the transaction in which frequent itemset occurs.

Table 3.18 Performance Results of RISOTTO in seconds

min_sup (δ) in %	Runtime in Sec.		
	Prefixed-Itemset Storage	VDF	RISOTTO
chess			
20	21.0054	16.8578	11.5625
30	20.9810	16.0452	11.0023
40	18.0054	14.0750	9.0531
50	17.5612	13.3017	8.9234
60	16.2378	12.7943	7.3456
70	15.9301	11.9825	6.8421
Average	**18.2868**	**14.1761**	**9.1215**
mushroom			
20	24.1790	21.1215	16.6217
30	23.6723	20.0462	15.7312
40	22.5724	19.7083	14.4581
50	22.0245	18.2058	13.9210
60	20.8256	17.7898	12.8521
70	19.9310	15.9575	10.6719
Average	**22.2008**	**18.8049**	**14.0427**
t25i10d10k			
20	26.6373	23.3254	19.2415
30	25.6037	21.4578	17.5689
40	24.9612	20.0025	15.9121
50	22.5817	18.7621	13.7321
60	21.7630	18.0056	13.0012
70	19.0175	16.0527	11.9801
Average	**23.4274**	**19.6010**	**15.2393**
c20d10k			
20	27.9152	24.4253	19.6142
30	25.2081	22.6752	17.5127
40	24.3574	21.9546	16.3382
50	23.6490	19.4316	14.9102
60	22.7518	19.0012	14.0045
70	21.0186	17.5242	12.6251
Average	**24.1500**	**20.8354**	**15.8342**
Overall Average (All Datasets)	**22.0163**	**18.3543**	**13.5594**

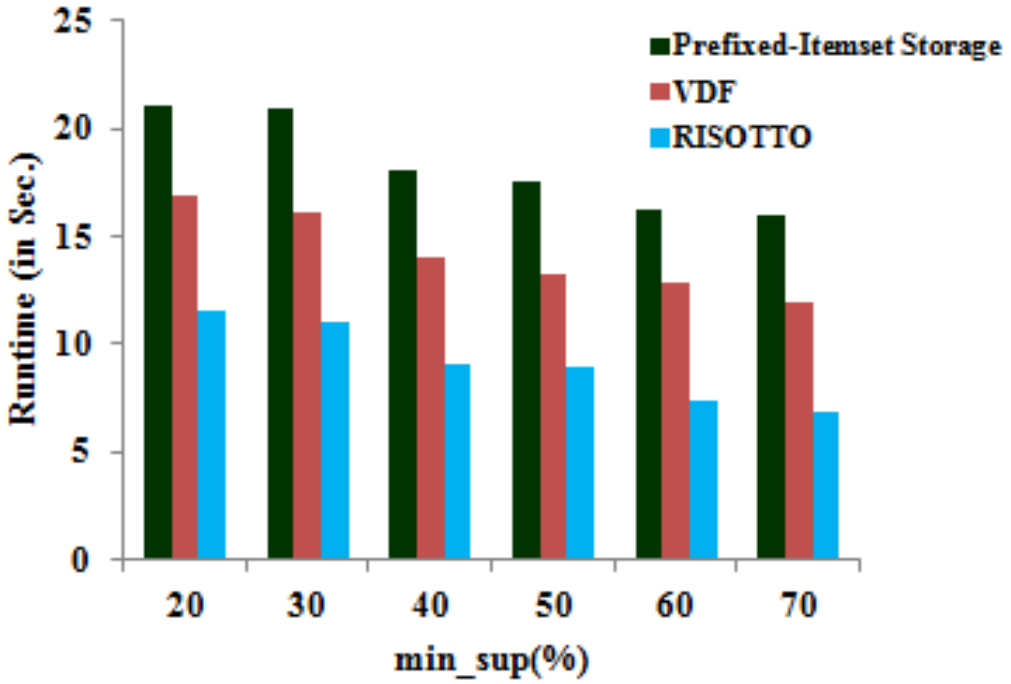


Figure 3.2 Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for chess Dataset

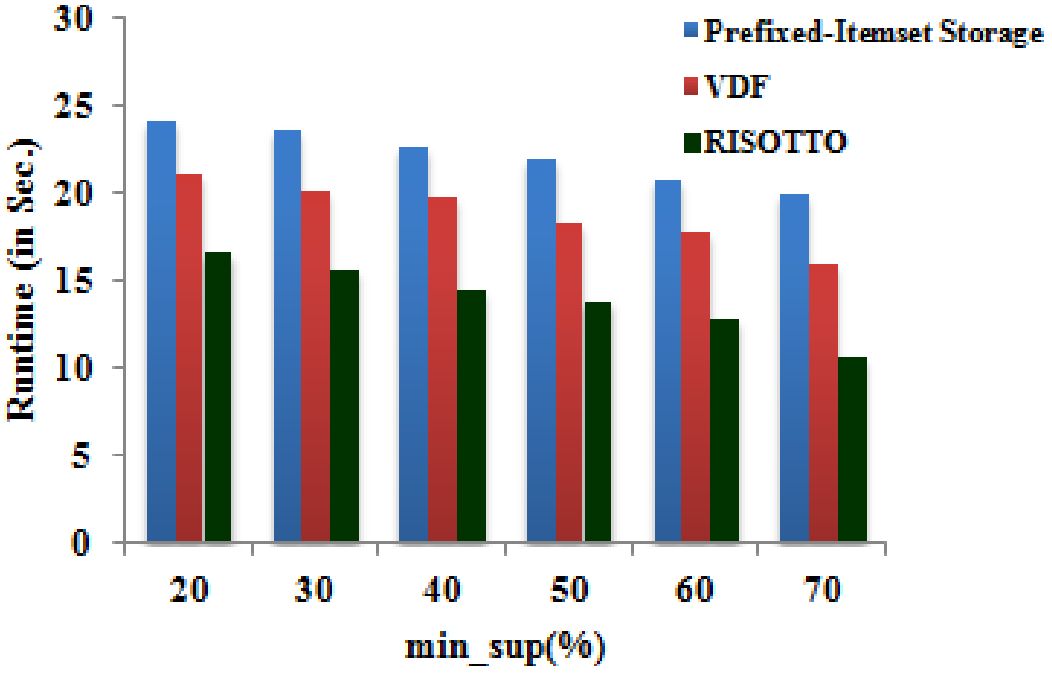


Figure 3.3 Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for mushroom Dataset

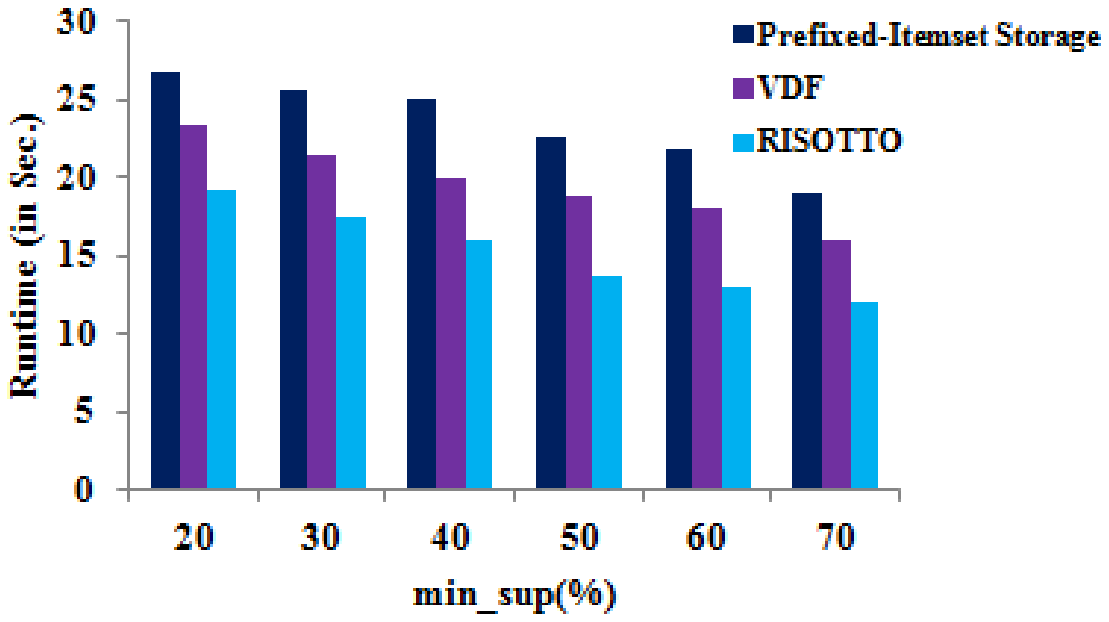


Figure 3.4 Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for t25i10d10k Dataset

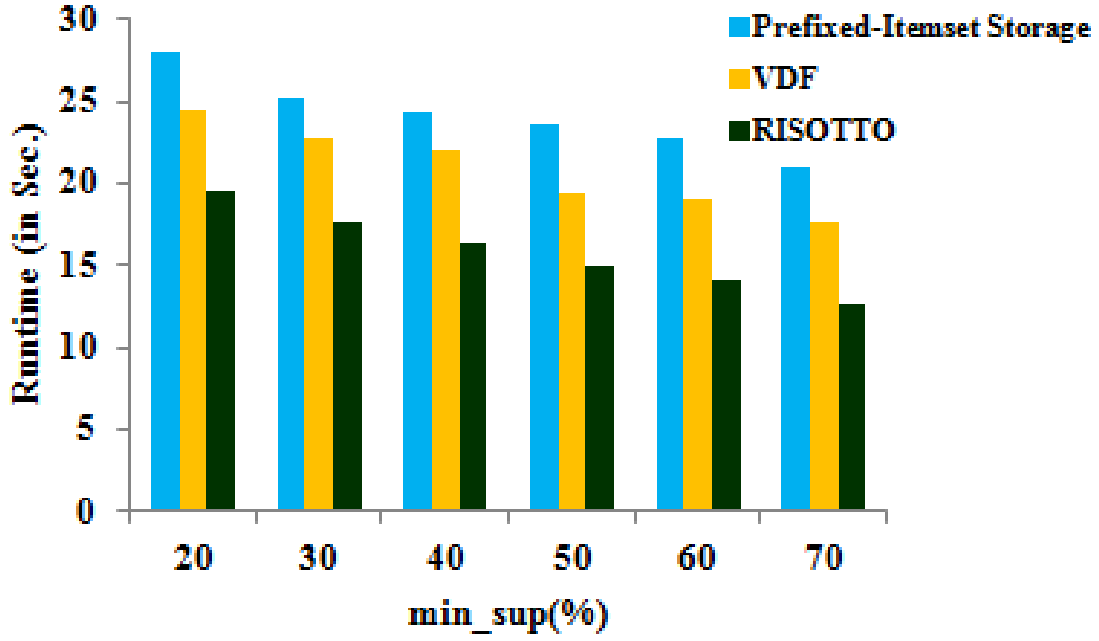


Figure 3.5 Runtime of Prefixed-Itemset Storage, VDF and RISOTTO for c20d10k Dataset

3.5.1 Welch's Two Sample *t*-test

The Welch's *t*-test is a statistical test applied when two groups of samples have unequal variances and/or unequal sizes with normally distributed data. It is named after the inventor Bernard Lewis Welch. It is also called an unequal variances *t*-test. It is calculated by taking the differences between the sample means and then dividing it by the standard error of that difference as shown in Equation 3.1.

$$t = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \quad \text{…Equation (3.1)}$$

where, $\overline{X_1}$ and $\overline{X_2}$ are the means, s_1^2 and s_2^2 are the variances, and n_1 and n_2 are the sizes of the two groups respectively.

The following hypothesis and level of significance (α) = 5% were considered for the statistical test.

Null Hypothesis (H_0)

There is no difference between the (*true*) means of the two groups i.e. $\mu_1 = \mu_2$.

Alternate Hypothesis (H_1)

There is a difference between the (*true*) means of the two groups. i.e. $\mu_1 < \mu_2$ or $\mu_1 > \mu_2$ or $\mu_1 \neq \mu_2$.

To prove statistically, a Welch's Two Sample *t*-test between the runtimes of the prefixed-itemset storage method and RISOTTO were performed in this research work using the R tool. The below example illustrates how to apply the *t*-test between the runtimes of the chess dataset for the prefixed-itemset storage and RISOTTO.

```
> prefix_chess = c(21.0054, 20.9810, 18.0054, 17.5612, 16.2378, 15.9301)

> RISOTTO_chess = c(11.5625, 11.0023, 9.0531, 8.9234,7.3456, 6.8421)

> t.test(prefix_chess,RISOTTO_chess)

        Welch Two Sample t-test

data:  prefix_chess and RISOTTO_chess

t = 7.6647, df = 9.732, p-value = 1.999e-05

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

 6.490968 11.839666

sample estimates:

mean of x mean of y

 18.28682  9.12150
```

The *p*-value for chess dataset is $1.999\times10^{-05} \leq 0.05$ (5%). Thus, the H_0 is rejected and H_1 is accepted. Therefore, it is concluded that the two means are not equal which means that there are significant differences between the runtimes of prefixed-itemset storage and RISOTTO. Similarly, the test is conducted for the remaining datasets used in the experiments and the results are tabulated in Table 3.19.

Table 3.19 Results of *t*-test

Dataset	***p*-value**
chess	1.999×10^{-05}
mushroom	3.031×10^{-05}
t25i10d10k	0.0005294
c20d10k	0.0001506

It is observed from Table 3.19 is that the *p*-values for all datasets are ≤ 0.05 (5%). So, it is concluded that there are significant differences between the runtimes. Therefore, the proposed method RISOTTO is more efficient in terms of runtime than prefixed-itemset storage.

3.6 Chapter Summary

The research work has introduced an enhanced Apriori algorithm called RISOTTO, a new hybrid approach for generating frequent itemsets that combine VDF and prefixed-itemset based storage *DS*. In the proposed method, the frequent 1-itemset stores the transactions in which the frequent 1-itemset occurs and restricts the number of database scans required to find the frequent itemsets to one and thereby reducing the I/O cost. The joining and pruning steps are performed using the values in the prefixed-itemset *DS* rather than the values in frequent itemsets as in classical Apriori, which reduced the time required to generate the candidate itemsets and also minimizes the number of candidate itemsets. Thus, the RISOTTO method enhances the existing Apriori algorithm. Though this algorithm reduces the running time when compared with the existing algorithms, it lacks in reducing memory consumption. To minimize the memory requirement, a memory-efficient implementation has been proposed in the next chapter and it also used the VDF approach for storing the database.

Chapter - 4

JAB-VDF: A JAGGED ARRAY BASED DATA STRUCTURE FOR VERTICAL DATA FORMAT

CHAPTER - 4

JAB-VDF: A JAGGED ARRAY BASED DATA STRUCTURE FOR VERTICAL DATA FORMAT

The purpose of critical thinking is rethinking: that is, reviewing, evaluating, and revising thought

--Jon Stratton

4.1 Background

Nowadays, volumes of data are exploding both in scientific and commercial domains. Data mining techniques are used to extract unknown information from a massive amount of data and became popular in many applications. But, the real-world datasets are sparse, dirt and also contain hundreds of items. Association Rule Mining (ARM) is an essential core data mining technique to discover patterns/rules among the items in large databases of variable-length transactions. Its goal is to identify the groups of items that most often occur together, i.e. it focuses on finding frequent itemsets, each occurring at more than a minimum support frequency (*min_sup*) among all transactions. It is widely used in market basket analysis and graph mining applications such as pattern finding in web browsing, substructure discovery in chemical compounds, word occurrence analysis in text documents, and so on [LLCL,08].

Apriori is one of the premier and classical data mining algorithms for finding frequent patterns but it is not an optimized one. Over the last two decades, remarkable variations and improvements were made to overcome the inefficiencies of the Apriori algorithm, such as FPGrowth, TreeProjection, Charm, LCM, Eclat and Direct Hashing and Pruning (DHP), RARM, ASPMS etc. In these algorithms, a minor enhancement improves the mining process considerably. The significant risks associated with

finding frequent itemsets are computational time and memory requirement. Even with a moderate-sized dataset, the search space and memory utilization of Frequent Pattern Mining (FPM) is enormous and exponential to the length of the transactions in the dataset. Therefore, it is essential to perform FPM analysis in a space-and-time efficient way.

Frequent itemset mining with Vertical Data Format (VDF) approach has been proposed in the literature to improve the classical Apriori. It reduces the number of database scans and uses an array storage structure. Since the VDF approach uses only one scan of the database, many researchers used this to reduce computational time to find frequent itemsets. Thus, this work reduces memory utilization using a space-efficient data structure called a jagged array with VDF.

4.2 Jagged Array

A jagged array or ragged grid is a data structure whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes and it is possible to create a 2-D array with a variable number of columns in each row. These types of arrays are sometimes called an "array of arrays" [Sch,07]. It is diagrammatically represented in Figure 4.1.

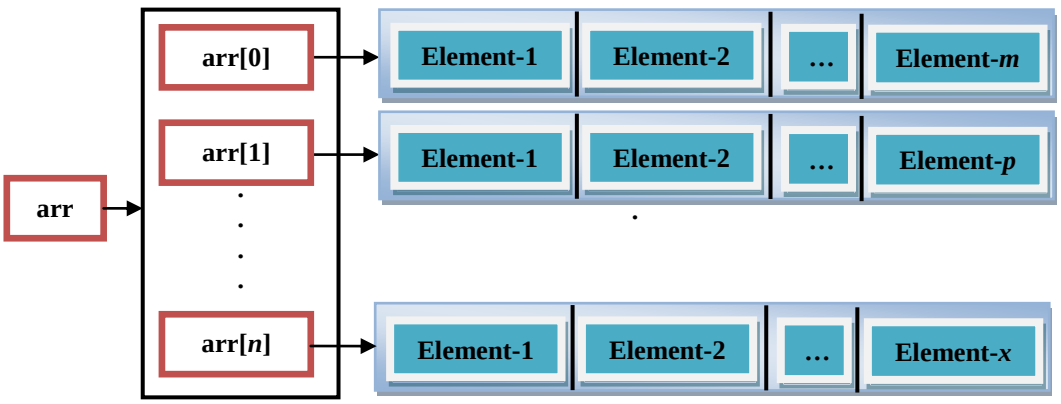


Figure 4.1 Jagged Array Representations

In the above diagram, *arr* is a jagged array that consists of *n* array and the length of each array can differ, i.e. $m \neq p \neq x$.

4.3 Proposed Methodology

Both Apriori and FP-growth algorithms mine frequent patterns using Horizontal Data Format (HDF), whereas the Eclat algorithm uses VDF. Both HDF and VDF approaches used array storage structures and observed that the VDF is a speedy method [IR,16]. To reduce the memory space further, this research work implements the VDF using the jagged array[2]. This concept is available in JAVA, Python, VB.NET and C#.NET [Sch,07].

The reason for choosing this data structure is that the customers will not buy all the items in the grocery shops. Each transaction in the transactional database contains a varied number of items purchased.

4.3.1 Illustration by an Example

To illustrate the memory requirement for VDF with jagged array representation, let us consider the transactional database (*D*) shown in Table 4.1. From Table 4.1, it is observed that the grocery shop sells *n* (12) items viz., a, b, c, d, e, f, g, h, i, k, p and m. *D* consists of *t* (10) transactions, and the *TID*'s are ranging from 0 to 9.

The VDF of Table 4.1 is illustrated in Table 4.2. The VDF is stored as a 2-D array in the memory, where the number of rows (*r*) = items in the grocery shop and the number of columns (*c*) = *t*. Here $r = 12$ and $c = 10$.

[2]**P.Sumathi**, S.Murugan, "A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach", International Journal of Computer Sciences and Engineering, E-ISSN: 2347-2693, Vol. 6, No. 11, pp.152-157, December 2018. (**UGC Approved Journal**).

Table 4.1 Transactional Database D

TID	Items Purchased
0	c, d, e, g, h, i, k, p, m
1	b, e, f, g, h, i, p, m
2	c, e, m
3	a, b, c, d, e, f, g, i, p
4	a, b, c, d, e, p
5	a, b, c, d, f, h, p
6	b, e, f, h, i, p, m
7	a, c, d, e, k, p, m
8	a, c, d, e, f, i, p, m
9	a, c, d, e, f, h, i, p, m

Table 4.2 D in VDF

Item	Transaction ID's (TID's)
a	{3, 4, 5, 7, 8, 9}
b	{1, 3, 4, 5, 6}
c	{0, 2, 3, 4, 5, 7, 8, 9}
d	{0, 3, 4, 5, 7, 8, 9}
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}
f	{1, 3, 5, 6, 8, 9}
g	{0, 1, 3}
h	{0, 1, 5, 6, 9}
i	{0, 1, 3, 6, 8, 9}
k	{0, 7}
m	{0, 1, 2, 6, 7, 8, 9}
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}

The memory required for storing candidate 1-itemset in the 2-D array for VDF is

$$TM_1 = (r \times c \times sizeof(tid)) + (sizeof(item_{11}) \times r) \quad \text{… Equation (4.1)}$$

where, $item_{11}$ is the first item in the candidate 1-itemset, *tid* is the transaction-id, and *sizeof* is a built-in function that says the number of bytes required for the

argument. Here each *tid* requires 2-bytes and $item_{11}$ requires 1-byte of memory, respectively. Therefore the VDF of candidate 1-itemset requires (12×10×2) + (1×12) = 252 bytes of memory i.e. TM_1 = 252 bytes.

The Support Count (*SC*) for each item is the number of *tid*'s that it contains, i.e. the *SC* of a, SC_a=count(a)=6. Similarly, SC_b=5, SC_c=8, SC_d=7, SC_e=9, SC_f=6, SC_g=3, SC_h=5, SC_i=6, SC_k=2, SC_m=7 and SC_p=9. Let the *min_sup* be 6. The frequent 1-itemset contains {a, c, d, e, f, i, m, p} and it is shown in Table 4.3.

Table 4.3 Frequent 1-itemset in VDF

Item	TID's
a	{3, 4, 5, 7, 8, 9}
c	{0, 2, 3, 4, 5, 7, 8, 9}
d	{0, 3, 4, 5, 7, 8, 9}
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}
f	{1, 3, 5, 6, 8, 9}
i	{0, 1, 3, 6, 8, 9}
m	{0, 1, 2, 6, 7, 8, 9}
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}

The spaces occupied by the in-frequent items say b, g, h and k in candidate 1-itemsets can be removed, saving memory considerably. The number of bytes of memory removed from candidate 1-itemset is computed as

$$rbytes_1 = (rr_1 \times c \times sizeof(tid)) + (rr_1 \times sizeof(item_{11})) \quad \text{…Equation (4.2)}$$

where, rr_1 is the number of rows to be removed as in-frequent items. For this example rr_1 = 4. Therefore, $rbytes_1$ = (4×10×2) + (4×1) = 84 bytes. Therefore the total bytes of memory for a frequent 1-itemset is

$$M_1 = TM_1 - rbytes_1 \quad \text{… Equation (4.3)}$$

Here M_1 = 252 - 84 = 168 bytes.

Similarly, in iteration 2, the possible 2-itemsets combinations are generated from frequent 1-itemsets, and it is {ac, ad, ae, af, ai, am, ap, cd, ce, cf, ci, cm, cp, de, df, di, dm, dp, ef, ei, em, ep, fi, fm, fp, im, ip, mp}. Suppose if there are *n* items in 1-itemset, the possible two-item combinations are $n \times (n\text{-}1)/2$ say tc_2. The numbers of itemset combinations say *x* may be in-frequent which need not be placed in VDF. Therefore, the memory required for a frequent 2-itemset is calculated using Equation 4.4.

$$TM_2 = ((tc_2 - x) \times c \times sizeof(tid)) + (sizeof(item_{21}) \times (tc_2 - x)) \quad \ldots \text{Equation (4.4)}$$

where, $item_{21}$ is the first item in the frequent 2-itemset. In this example, the item combinations viz., {ae, af, ai, am, cf, ci, cm, df, di, dm, ef, fi, fm, im} are in-frequent. Based on Equation 4.4, the VDF of frequent 2-itemset requires ((28 - 14) × 10 × 2) + (2 × (28 - 14)) = 280 + 28 = 308 bytes and the frequent 2-itemsets is shown in Table 4.4.

Similarly, from Table 4.4, the 3-itemset combinations satisfy the Apriori property viz., {acd, acp, adp, cde, cdp, cep, dep, emp, eip} are the candidate 3-itemset. In this case, all candidate 3-itemsets are frequent itemsets. Therefore the frequent 3-itemset requires ((9 - 0) × 10 × 2) + (3 × (9 - 0)) = 180 + 27 = 207 bytes of memory and it is shown in Table 4.5.

Similarly, the 4-itemsets combinations generated from frequent 3-itemsets are acdp, acde, acep, adep, cdep, cemp, ceip, demp, deip and eimp. Among them, the items acdp and cdep are satisfied Apriori property, which forms the candidate 4-itemset. All the candidate 4-itemsets satisfy the minimum support. The frequent 4-itemset is shown in Table 4.6.

Therefore, the frequent 4-itemset requires ((10 - 0) × 10 × 2) + (4 × (10 - 0)) = 200 + 40 = 240 bytes.

Table 4.4 VDF of frequent 2-itemsets

Item	TID's
ac	3, 4, 5, 7, 8, 9
ad	3, 4, 5, 7, 8, 9
ap	3, 4, 5, 7, 8, 9
cd	0, 3, 4, 5, 7, 8, 9
ce	0, 2, 3, 4, 7, 8, 9
cp	0, 3, 4, 5, 7, 8, 9
de	0, 3, 4, 7, 8, 9
dp	0, 3, 4, 5, 7, 8, 9
ei	0, 1, 3, 6, 8, 9
em	0, 1, 2, 6, 7, 8, 9
ep	0, 1, 3, 4, 6, 7, 8, 9
fp	1, 3, 5, 6, 8, 9
ip	0, 1, 3, 6, 8, 9
mp	0, 1, 6, 7, 8, 9

Table 4.5 VDF of frequent 3-itemsets

Item	TID's
acd	3, 4, 5, 7, 8, 9
acp	3, 4, 5, 7, 8, 9
adp	3, 4, 5, 7, 8, 9
cde	0, 3, 4, 7, 8, 9
cdp	0, 3, 4, 5, 7, 8, 9
cep	0, 3, 4, 7, 8, 9
dep	0, 3, 4, 7, 8, 9
emp	0, 1, 6, 7, 8, 9
eip	0, 1, 3, 6, 8, 9

Table 4.6 VDF of frequent 4-itemsets

Item	TID's
acdp	3, 4, 5, 7, 8, 9
cdep	0, 3, 4, 7, 8, 9

This process is repeated until no frequent itemsets are found. Now the candidate 5-itemset contains only one item, i.e. {acdep} and it is not frequent.

So the frequent 5-itemset is empty (∅), and the process is terminated. Therefore, the total memory required for VDF using a 2-D array is

$$TM = M_1 + \sum_{i=2}^{itemset_i \neq \varnothing} TM_i \quad \text{… Equation (4.5)}$$

where, M_1 is calculated using Equation 4.3 and TM_i are calculated using Equation 4.6.

$$TM_i = ((tc_i - x) \times c \times sizeof(tid)) + (sizeof(item_{i1}) \times (tc_i - x)) \quad \text{… Equation (4.6)}$$

where, tc_i and x is the number of frequent and in-frequent items in the candidate i-frequent itemset. For the above example TM = 168 + 308 + 207 + 240 = 923 bytes of memory. If the same is implemented using the jagged array, the memory requirement is reduced considerably. The memory required for candidate 1-itemset TM_1 is calculated as

$$TM_1 = \sum_{\forall item \in \{itemset_1\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \text{… Equation (4.7)}$$

As in 2-D representation, there may be x in-frequent items in candidate 1-itemset say $\{in\text{-}frequent\} = \{item_1, item_2, \ldots, item_x\}$ then the memory for $\{in\text{-}frequent\}$ can be saved by removing it and the amount of memory removed is computed as shown in Equation 4.8.

$$rbytes_1 = \sum_{\forall item \in \{in-frequent\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \text{… Equation (4.8)}$$

Therefore the total memory required for frequent 1-itemset in jagged array representation is computed using Equation 4.3 with the values calculated using Equations 4.7 and 4.8, respectively. The jagged array representation for frequent 1-itemset for D is shown in Table 4.7.

Table 4.7 Jagged Array Representation of frequent 1-itemset

Item	TID's								
a	3	4	5	7	8	9			
c	0	2	3	4	5	7	8	9	
d	0	3	4	5	7	8	9		
e	0	1	2	3	4	6	7	8	9
f	1	3	5	6	8	9			
i	0	1	3	6	8	9			
m	0	1	2	6	7	8	9		
p	0	1	3	4	5	6	7	8	9

The memory required for the above table is calculated as shown below.

TM_1 = (6×2+1) + (5×2+1) + (8×2+1) + (7×2+1) + (9×2+1) + (6×2+1) + (3×2+1)

+ (5×2+1) + (6×2+1) + (2×2+1) + (7×2+1) + (9×2+1)

= 13+11+17+15+19+13+7+11+13+5+15+19

= 158 bytes

$rbytes_1$ = (5×2+1) + (3×2+1) + (5×2+1) + (2×2+1)

= 11+7+11+5 = 34 bytes

Therefore, M_1 = 158 - 34 = 124 bytes. Similarly, the jagged array representation of frequent 2-itemsets shown in Table 4.8, requires $TM_2 - rbytes_2$ bytes of memory space where, TM_2 and $rbytes_2$ are calculated by using Equations 4.9 and 4.10 respectively.

$$TM_2 = \sum_{\forall item \in \{itemset_2\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \ldots \text{Equation (4.9)}$$

$$rbytes_2 = \sum_{\forall item \in \{in-frequent\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \ldots \text{Equation (4.10)}$$

Table 4.8 Jagged Array Representation of frequent 2-itemset

Item	TID's							
ac	3	4	5	7	8	9		
ad	3	4	5	7	8	9		
ap	3	4	5	7	8	9		
cd	0	3	4	5	7	8	9	
ce	0	2	3	4	7	8	9	
cp	0	3	4	5	7	8	9	
de	0	3	4	7	8	9		
dp	0	3	4	5	7	8	9	
ei	0	1	3	6	8	9		
em	0	1	2	6	7	8	9	
ep	0	1	3	4	6	7	8	9
fp	1	3	5	6	8	9		
ip	0	1	3	6	8	9		
mp	0	1	6	7	8	9		

For the above table,

TM_2 = (6×2+2) + (6×2+2) + (5×2+2) + (4×2+2) + (3×2+2) + (3×2+2) +

(6×2+2) + (7×2+2) + (7×2+2) + (4×2+2) + (4×2+2) + (5×2+2) +

(7×2+2) + (6×2+2) + (4×2+2) + (4×2+2) + (4×2+2) + (7×2+2) +

(5×2+2) + (6×2+2) + (7×2+2) + (8×2+2) + (5×2+2) + (4×2+2) +

(6×2+2) + (5×2+2) + (6×2+2) + (6×2+2)

= 14+14+12+10+8+8+14+16+16+10+10+12+16+14+10+10+10+16+12+14+16

+18+12+10+14+12+14+14

= 356 bytes

$rbytes_2$ = (5×2+2) + (4×2+2) + (3×2+2) + (3×2+2) + (4×2+2) + (4×2+2) +

(5×2+2) + (4×2+2) + (4×2+2) + (4×2+2) + (5×2+2) + (5×2+2) +

(4×2+2) + (5×2+2)

= 12+10+8+8+10+10+12+10+10+10+12+12+10+12

= 146 bytes

and therefore, M_2 requires 356 - 146 = 210 bytes of memory. Similarly, the jagged array representation of frequent 3-itemsets shown in Table 4.9 requires $TM_3 - rbytes_3$ memory.

Table 4.9 Jagged Array Representation of frequent 3-itemset

Item	TID's						
acd	3	4	5	7	8	9	
acp	3	4	5	7	8	9	
adp	3	4	5	7	8	9	
cde	0	3	4	7	8	9	
cdp	0	3	4	5	7	8	9
cep	0	3	4	7	8	9	
dep	0	3	4	7	8	9	
emp	0	1	6	7	8	9	
eip	0	1	3	6	8	9	

For Table 4.9,

TM_3 = (6×2+3) + (6×2+3) + (6×2+3) + (6×2+3) + (7×2+3) + (6×2+3) + (6×2+3) + (6×2+3) + (6×2+3)

= 15+15+15+15+17+15+15+15+15

= 137 bytes

$rbytes_3$ = 0 bytes

and therefore M_3 requires 137 - 0 = 137 bytes of memory.

Similar to the previous cases, the memory for frequent 4-itemsets is calculated as

TM_4 = (6×2+4) + (6×2+4) = 16 + 16 = 32 bytes.

$rbytes_4$ = 0 bytes

and

M_4 = 32 - 0 = 32 bytes of memory for Table 4.10.

Table 4.10 Jagged Array Representation of frequent 4-itemsets

Item	TID's					
acdp	3	4	5	7	8	9
cdep	0	3	4	7	8	9

This process continues until no more frequent itemsets are found. For this case, the candidate 5-itemset is *NULL*, and the algorithm terminates. Therefore, the total memory required for the jagged implementation is calculated using Equation 4.11.

$$TM = \sum_{i=1}^{itemset_i \neq \varnothing} TM_i - rbytes_i \quad \text{… Equation (4.11)}$$

where, TM_i and $rbytes_i$ are calculated using Equations 4.12 and 4.13, respectively.

$$TM_i = \sum_{\forall item \in \{itemset_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \text{… Equation (4.12)}$$

$$rbytes_i = \sum_{\forall item \in \{in-frequent_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \text{… Equation (4.13)}$$

Therefore, the jagged array representation for the sample transactional database *D* requires

$$TM = M_1 + M_2 + M_3 + M_4$$

TM = 124 + 210 + 137 + 32 = 503 bytes of memory and it is less when compared to the original 2-D array representation.

The jagged array representation of VDF has several advantages. They are:

i. No memory space is wasted as in a 2-D array because a jagged array allocates space only to the transactions in which the item occurs

ii. Minimizes the memory space required than the original array implementation

Thus, it is finalized that the jagged representation saves memory significantly and also it is fast when compared with the HDF approaches.

4.4 Experimental Results and Discussion

To analyze the memory usage of VDF using the jagged array, an empirical study has been performed for the datasets namely chess, mushroom, t25i10d10k and c20d10k using Python implementation. All the datasets were obtained from the FIMI repository (http://fimi.ua.ac.be) and the open-source data mining library (http://www.philippe-fournier-viger.com/spmf). The chess dataset contains 3196 transactions, 75 items and 37 average item count per transaction. Similarly, the mushroom, t25i10d10k and c20d10k contain 8416, 9976, and 10000 transactions, 119, 929 and 192 items and 23, 24.77 and 20 average item count per transaction respectively. The memory usage of JAB-VDF is compared with VDF (2-D array) is carried for the four datasets with δ=20% and it is tabulated in Table 4.11 and Figure 4.2.

Table 4.11 Comparison of Memory Consumption (in GB) between JAB-VDF and VDF with δ=20%

Datasets	VDF	JAB-VDF
chess	1.2500	0.7500
mushroom	1.5000	0.6750
t25i10d10k	1.7500	0.7000
c20d10k	1.6700	0.9185
Average	**1.5425**	**0.7609**

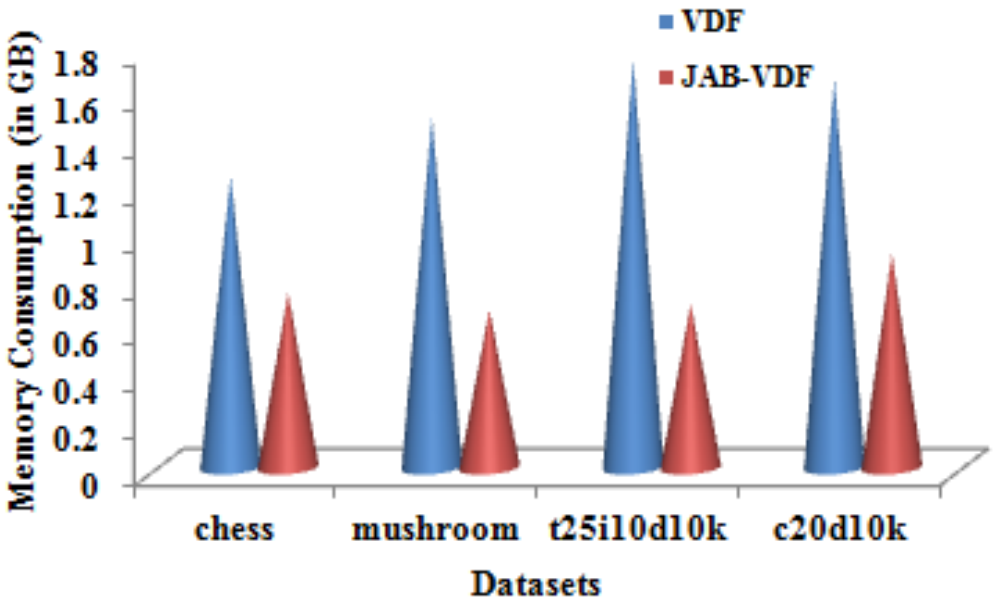


Figure 4.2 Comparison of Memory Consumption (in GB) between JAB-VDF and VDF with δ=20%

From Table 4.11 and Figure 4.2, it was observed that the memory needed for JAB-VDF is reduced by 49.33% when compared with VDF. Further, to prove statistically, Welch's two-sample *t*-test was performed between the memory usage of JAB-VDF and VDF. The *t*-test was performed using the R tool and the *p*-value is 0.0023 which is $\leq$ 0.05 (5%). It is concluded that the two means are not equal, which means that there are significant differences between the memory usage of JAB-VDF and VDF. Therefore, the proposed method JAB-VDF consumes less memory than VDF considerably.

4.5 Chapter Summary

From the literature, it is also found that the VDF approaches restrict the database scans to one and find the support counts by intersection. Though it is best, the array storage structure used by VDF consumes huge memory space because it assumes that each item may fall almost in all transactions. But in real-world grocery datasets, each transaction will not contain all items, and each item may not be present

in all transactions. Thus, to reduce memory consumption and utilize memory efficiently, this research work used the jagged array representation. From the experimental results, it has been observed that the JAB-VDF reduces memory consumption for storing frequent itemsets when compared with the traditional 2-D array. The next chapter focuses on developing an algorithm for finding frequent patterns by reducing both time and memory using a multithreaded approach and jagged array.

Chapter - 5

TB-NPF-VDF: A MULTITHREADED, NOVEL PATTERN FORMATION FOR VERTICAL DATA FORMAT WITH JAGGED ARRAY

CHAPTER - 5

TB-NPF-VDF: A MULTITHREADED, NOVEL PATTERN FORMATION FOR VERTICAL DATA FORMAT WITH JAGGED ARRAY

Imagination encircles the entire world, stimulating progress, giving birth to evolution

--Albert Einstein

5.1 Background

Association Rule Mining (ARM) is one of the most extensively used knowledge discovery techniques and a promising area in the mining domain [AHGA^{+},18]. ARM is used in several applications such as inventory control, mobile mining, educational mining, market basket analysis, risk management, telecommunication networks, graph mining, etc. [SK,19]. The problem of mining frequent itemset/pattern is a sub-problem of ARM [GAF,17]. Frequent patterns are patterns that frequently appear in a dataset with a frequency more than a user-specified threshold. Frequent Pattern Mining (FPM) is an essential task of discovering hidden items from a database with more than a prescribed threshold. It generates qualitative knowledge that helps the decision makers make good business insights [HPK,12].

Many researchers narrated novel algorithms for finding frequent itemset mining, which is achieved using a single thread, but still, there is a need for time, memory efficient and scalable one. Therefore, the research study proposed an approach for finding frequent patterns, namely TB-NPF-VDF (Thread Based, Novel Pattern Formations with Vertical Data Format), which uses a new way of generating candidate items to minimize the time. Also, it employs multithreading which runs

several threads simultaneously, one for each frequent 1-itemset to generate the remaining frequent itemsets (frequent 2-itemsets, frequent 3-itemsets, etc.) for that item until the candidate or frequent itemsets are not empty. Further, to reduce the memory requirement significantly, it also employs a jagged array structure for storing the frequent patterns, as illustrated in chapter 4.

The research work has been implemented and tested using four standard benchmark datasets from the frequent itemset mining repository. Further, it is compared with VDF and NPF-VDF (without multithread), and the experimental results revealed that TB-NPF-VDF outperforms in terms of execution time and memory significantly.

5.2 Multithreading

It is a process of executing multiple threads simultaneously, i.e. thread-based multitasking. A thread is a lightweight sub-process, and it is the smallest unit of a process. Each thread has a separate path of execution and executed inside a process. The multithreading uses a shared memory area and thus saves memory space considerably. Similarly, the context switching between threads takes less time than the process. The pictorial representation of multithreading is shown in Figure 5.1.

Concurrent activity speeds applications up is one of the main benefits of multithreading. Apart from this, it has numerous advantages. They are:

i. Requires less overhead to create, maintain, and manage threads than a traditional process
ii. Improves throughput
iii. Improves the application and server responsiveness

iv. Minimizes the usage of system resources

v. Simplifies the structure of a complex program

vi. The cost of communication between threads is low

vii. It doesn't block users or affect other threads if an exception occurs because threads are independent

viii. Saves time to complete the task

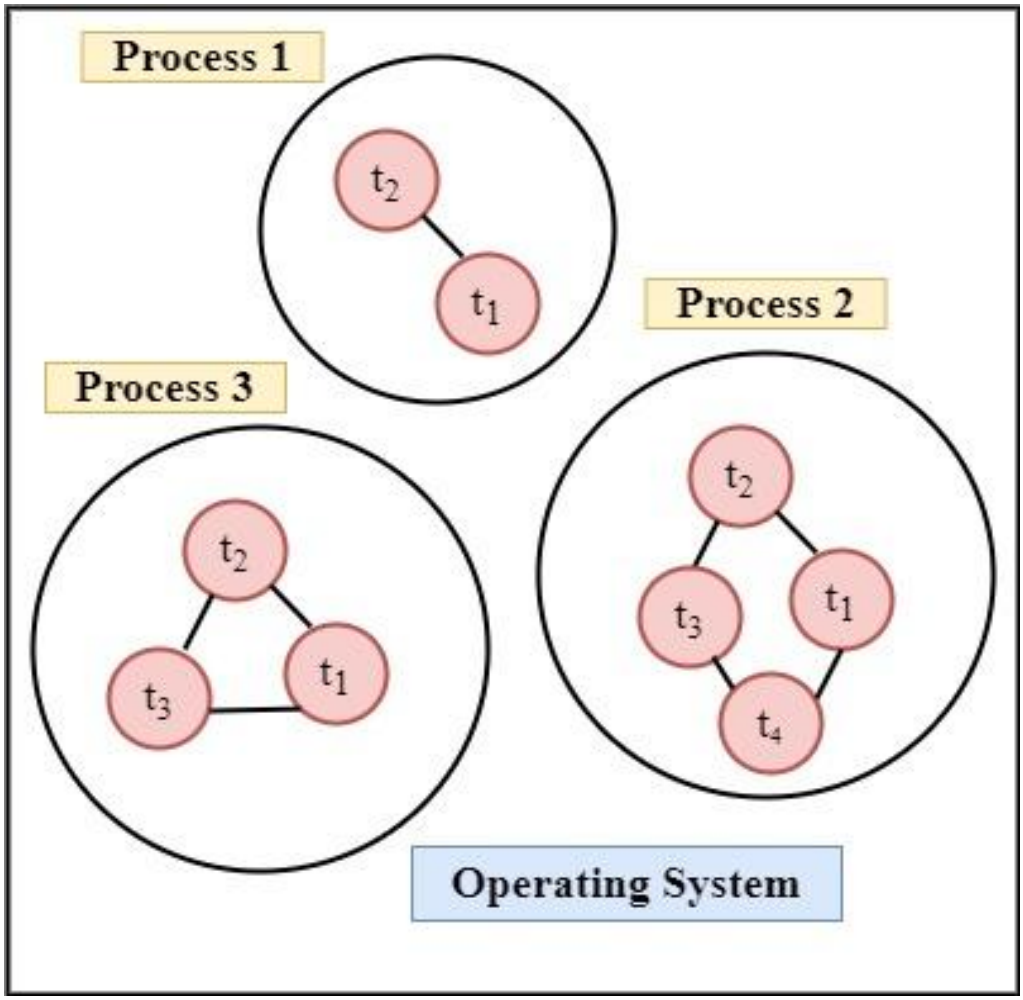


Figure 5.1 Multithreading

By considering these advantages, the research work proposed in this chapter uses the multithreading concept to increase the runtime speed.

5.3 Proposed Methodology

The main idea of the proposed work is to find the frequent patterns for the transactional database. It consists of four phases. The first phase scans *D* and converts it into VDF. The second phase determines the frequent 1-itemset from VDF. The third phase sorts the frequent 1-itemset in ascending order based on the *min_sup* (δ) threshold, and it is stored in a matrix form using a jagged array. The δ of an itemset *X* is calculated by dividing the number of transactions in which *X* appears by a total number of transactions [Kal,17]. The fourth phase creates *n*-1 threads, one for each frequent 1-itemset except for the last one, where *n* is the number of items in frequent 1-itemset. Let the frequent 1-itemset be $L_1=\{I_1,I_2,\ldots,I_n\}$, each thread generates frequent itemsets starting from frequent 2-itemset to frequent *k*-itemset until it is non-empty, where $k \geq 2$.

For finding frequent *i*-itemset, $i \geq 2$, each thread ($t_{x,1\leq x\leq n-1}$) uses the following procedure.

i. When i=2, the thread forms the candidate patterns by combining I_x with I_{x+1} and finds the transactions in which the combination I_xI_{x+1} occur by intersecting the transactions in I_x and I_{x+1}. The item combinations whose Support Count (SC) $\geq \delta$ is selected as frequent *i*-itemset for item *x*.
ii. For i>2, each item in frequent (i-1)-itemset is combined with each frequent 1-itemset starting from the next item in the last item of the frequent(i-1)-item and finds the transactions in which the combination occurs is determined by intersecting the item infrequent(i-1)-itemset and the appropriate item in frequent 1-itemset. This process is repeated until the frequent *k*-itemset is not empty.

As the proposed method uses multithreads, novel pattern formation with VDF to find frequent patterns is named TB-NPF-VDF[3]. The main advantage of this method is that it generates less number of candidate itemsets when compared with the classical Apriori and VDF because it avoids the items whose *SC* is lesser than the item at any instance of time for generating the patterns. As threads are used, the CPU is effectively utilized, and they are faster when compared to processes. This method avoids checking the pattern for the Apriori property because the candidate patterns generated satisfies the Apriori property by default. Further, the time required for TB-NPF-VDF is less when compared to VDF. Since the algorithm also uses the matrix notation using a jagged array, the memory requirement is also minimized [SM,18]. The algorithm for the proposed method is shown in Algorithm 5.1 and the workflow of TB-NPF-VDF is illustrated in Figure 5.2.

Algorithm 5.1: TB-NPF-VDF: An algorithm for finding frequent itemsets

Input:

- A dataset *D* with *n* transactions;
- δ - minimum support threshold.

Output: Frequent patterns.

Method:

(1) *vdf*←scan *D* and store it in <*itemset, TID list*> format;

(2) $C_1 \leftarrow \emptyset$;

(3) **for** each $item_i$ in *vdf* **do**

[3]**P.Sumathi**, Dr.S.Murugan, Dr.V.Umadevi, "A Multithread, Novel Pattern Based Algorithm for Finding Frequent Patterns With Jagged Array and Vertical Data Format", Indian Journal of Computer Science and Engineering (IJCSE), e-ISSN:0976-5166, p-ISSN:2231-3850, Vol.12, No.5, pp.1353-1363, Sep-Oct 2021. DOI:10.21817/indjcse/2021/v12i5/211205078 (**UGC Care List - II, Scopus Indexed**).

begin

$SC \leftarrow$count(TID_list_{itemi}); //determines the number of transactions in $item_i$

$C_1 \leftarrow C_1$.append ({*itemset, TID list, SC*}) // adds a row into C_1

endfor

(4) **for** each $item_i$ in C_1 **do**

begin

$L_1 \leftarrow \{item_i \mid SC(item_i) \geq \delta\}$

endfor

(5) $L_1 \leftarrow$jagged(sort(L_1)); //sorts L_1 and converts it into a jagged matrix format

(6) *no_freq1_itemset*$\leftarrow$count(L_1); //determines the number of itemset in L_1

(7) **for** (x=1; $x \leq$ (*no_freq1_itemset*-1); x++) **do**

begin

$t_x \leftarrow$create(thread); // creates a thread for the item $L_1[x]$

for (k=2; $L_k \neq \emptyset$; k++) **do**

begin

if k==2 **then**

new_pattern$\leftarrow <I_x I_{x+1}>$;

new_TID_list$\leftarrow$Transactions(I_x) $\cap$Transactions(I_{x+1});

else if $k \geq 2$ **then**

for each $item_j$ in L_{k-1} **do**

begin

new_item$\leftarrow$last item in $item_{j;}$

new_pattern$\leftarrow \{<item_j I_y> \mid I_y \leftarrow$next(*new_item*)};

new_TID_list$\leftarrow$Transactions($item_j$)$\cap$Transactions(I_y)

endfor

end if

$SC \leftarrow$ count(*new_TID_list*);

$C_k \leftarrow C_k$.append({*new_pattern*, *new_TID_list*});

$L_k \leftarrow \{C_k \mid SC(C_k) \geq \delta\}$

endfor

endfor

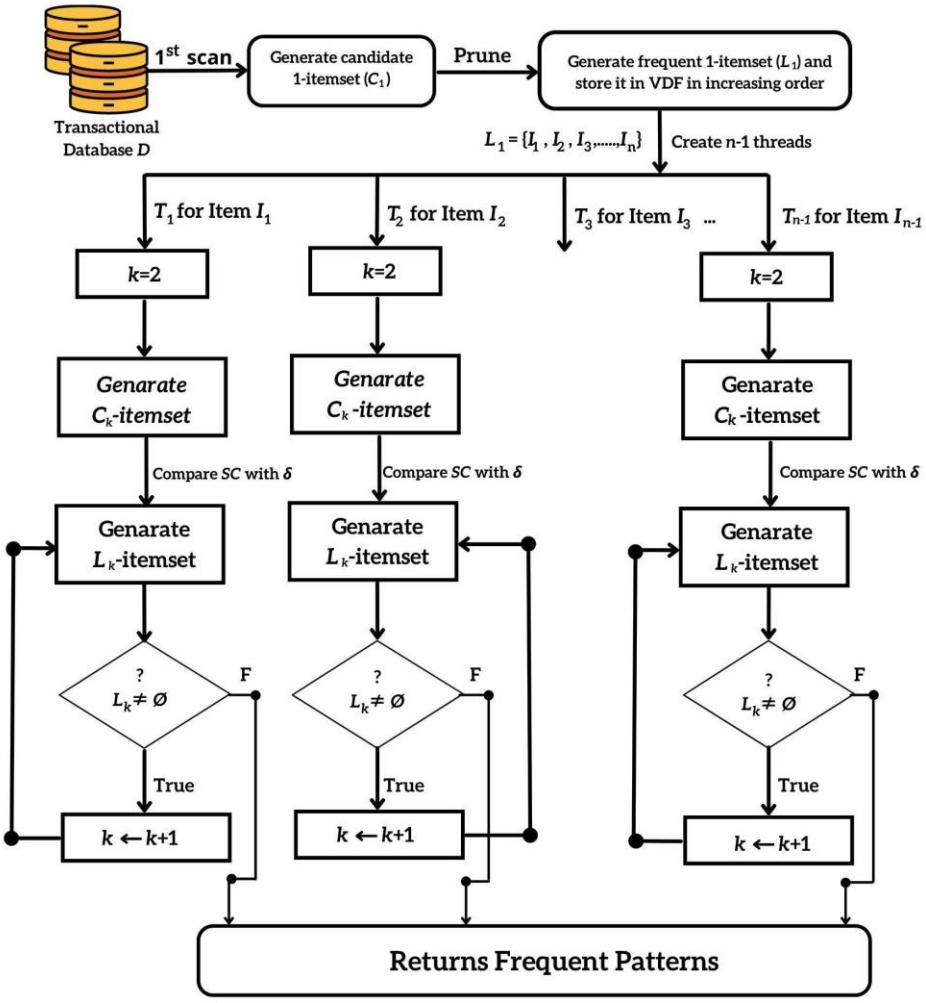


Figure 5.2 Workflow of TB-NPF-VDF

5.3.1 Illustration by an Example

The transactional database *D* shown in Table 5.1 is taken to illustrate the proposed work. It contains 12 items viz., {a, b, c, d, e, f, g, h, i, k, m, p}. The vertical

representation of *D* is shown in Table 5.2. Each item is represented by a row containing the name of the item and the transactions in which the item occurs.

Table 5.1 Transactional Database D

TID	Items Purchased
0	c, d, e, g, h, i, k, p, m
1	b, e, f, g, h, i, p, m
2	c, e, m
3	a, b, c, d, e, f, g, i, p
4	a, b, c, d, e, p
5	a ,b ,c, d, f, h, p
6	b, e, f, h, i, p, m
7	a, c, d, e, k, p, m
8	a, c , d, e, f, i, p, m
9	a, c, d, e, f, h, i, p, m

Table 5.2 D in VDF

Item	Transaction ID's (TID's)
a	{3, 4, 5, 7, 8, 9}
b	{1, 3, 4, 5, 6}
c	{0, 2, 3, 4, 5, 7, 8, 9}
d	{0, 3, 4, 5, 7, 8, 9}
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}
f	{1, 3, 5, 6, 8, 9}
g	{0, 1, 3}
h	{0, 1, 5, 6, 9}
i	{0, 1, 3, 6, 8, 9}
k	{0, 7}
m	{0, 1, 2, 6, 7, 8, 9}
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}

Let δ is 6. The candidate 1-itemset (C_1) contains all the items in *D*, the transactions in which the item occurs, and also the *SC*, i.e. the number of transactions in which the item appears. The C_1 for *D* is shown in Table 5.3. Among them, the items {a, c, d, e, f, i, m, p} satisfies the δ and hence forms the frequent 1-itemset. The jagged array representation of the same is shown in Table 5.4.

To generate fewer candidate itemsets, this research work uses a novel pattern generation method rather than the natural join used in the Apriori algorithm. For that, the frequent 1-itemset (L_1) is sorted in ascending order based on *SC*, and it is replaced with L_1. The frequent 1-itemset after sorting is illustrated in Table 5.5.

Table 5.3 Candidate 1-itemset

C_1

Itemset	TID's	SC
a	{3, 4, 5, 7, 8, 9}	6
b	{1, 3, 4, 5, 6}	5
c	{0, 2, 3, 4, 5, 7, 8, 9}	8
d	{0, 3, 4, 5, 7, 8, 9}	7
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}	9
f	{1, 3, 5, 6, 8, 9}	6
g	{0, 1, 3}	3
h	{0, 1, 5, 6, 9}	5
i	{0, 1, 3, 6, 8, 9}	6
k	{0, 7}	2
m	{0, 1, 2, 6, 7, 8, 9}	7
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}	9

Table 5.4 Jagged Array Representation of frequent 1-itemset

L_1

1- Itemset	TID's								
a	3	4	5	7	8	9			
c	0	2	3	4	5	7	8	9	
d	0	3	4	5	7	8	9		
e	0	1	2	3	4	6	7	8	9
f	1	3	5	6	8	9			
i	0	1	3	6	8	9			
m	0	1	2	6	7	8	9		
p	0	1	3	4	5	6	7	8	9

Now this work creates seven threads because the frequent 1-itemset contains eight items. Thread-1 is for the item <a>, Thread-2 is for item <f> and so on. The Thread-1 first generates the following patterns.

<af>, <ai>, <ad>, <am>, <ac>, <ae> and <ap>

and for each pattern, set intersection is calculated by using the *TID*'s in each item of the pattern. For example, for the pattern <af> the set intersection is calculated as {3, 4, 5, 7, 8, 9}∩{1, 3, 5, 6, 8, 9} = {3,5,8,9} and *SC*=4. Similarly, the *SC* for other patterns viz., <ai>, <ad>, <am>, <ac>, <ae> and <ap> is calculated as stated above. The patterns namely <ad>, <ac> and <ap> satisfies the δ will be considered as the frequent 2-itemset for the item <a> and are represented in Table 5.6.

Table 5.5 Sorted frequent 1-itemset L_1

1- Itemset	TID's								
a	3	4	5	7	8	9			
f	1	3	5	6	8	9			
i	0	1	3	6	8	9			
d	0	3	4	5	7	8	9		
m	0	1	2	6	7	8	9		
c	0	2	3	4	5	7	8	9	
e	0	1	2	3	4	6	7	8	9
p	0	1	3	4	5	6	7	8	9

Table 5.6 Frequent 2-itemset for <a> by Thread-1

Item	TID's					
<ad>	3	4	5	7	8	9
<ac>	3	4	5	7	8	9
<ap>	3	4	5	7	8	9

Next, the method generates the candidate 3-itemsets for each frequent 2-itemset in Table 5.6 as follows:

i. For the frequent 2-item <ad>, the items viz., <m>, <c>, <e> and <p> are considered from frequent-1 itemset because <m> is the next item after <d> where <d> is the last item in frequent 2-itemset <ad>. The patterns generated are <adm>, <adc>, <ade> and <adp> and for them, the transactions in which the pattern occurs and *SC* is calculated as follows:

   From Table 5.6 *TID*'s of <ad> is {3, 4, 5, 7, 8, 9} and from Table 5.5 the *TID*'s of <m> is {0, 1, 2, 6, 7, 8, 9}. Therefore, {3, 4, 5, 7, 8, 9}∩{0, 1, 2, 6, 7, 8, 9} ={7,8,9} and *SC*=3. Similarly, for <adc>, <ade> and <adp> is also calculated.

ii. For the frequent 2-item <ac>, the items from <e> i.e. <e> and <p> are considered. The patterns generated are <ace> and <acp> and *SC* is calculated as above.

iii. For the frequent 2-item <ap>, there is no candidate 3-itemset because there is no next item after <p>.

The candidate 3-itemset generated by Thread-1 are <adm>, <adc>, <ade>, <adp>, <ace> and <acp>. Among them the patterns viz., <adc>, <adp> and <acp> satisfies δ forms frequent 3-itemset and it is shown in Table 5.7.

Table 5.7 Frequent 3-itemsets for <a> by Thread-1

Itemset	TID's					
<adc>	3	4	5	7	8	9
<adp>	3	4	5	7	8	9
<acp>	3	4	5	7	8	9

The frequent 3-itemset for <a> is not empty, so the method generates the candidate 4-itemset. They are <adce> and <adcp>. The *SC* for <adce> is calculated as {3, 4, 5, 7, 8, 9}∩{0, 1, 2, 3, 4, 6, 7, 8, 9} = {3,4,7,8,9} and *SC* of <adcp> is 5. Similarly, for <adcp> is {3, 4, 5, 7, 8, 9}∩{0, 1, 3, 4, 5, 6, 7, 8, 9}={3,4,5,7,8,9} and the *SC*=6 and is shown in Table 5.8.

Table 5.8 Frequent 4-itemsets for <a> by Thread-1

Itemset	TID's					
<adcp>	3	4	5	7	8	9

Now, candidate 5-itemset for the item <a> is *Ø*. So Thread-1 stops its execution and returns <ad>, <ac>, <ap>, <adc>, <adp>, <acp> and <adcp> as frequent items for <a>. Similarly, the other threads generate frequent itemsets for other frequent 1-itemset in parallel and are shown from Table 5.9 to Table 5.19.

Table 5.9 Frequent 2-itemset for <f> by Thread-2

Itemset	TID's					
<fp>	1	3	5	6	8	9

Table 5.10 Frequent 2-itemset for <i> by Thread-3

Itemset	TID's					
<ie>	0	1	3	6	8	9
<ip>	0	1	3	6	8	9

Table 5.11 Frequent 3-itemset for <i> by Thread-3

Itemset	TID's					
<iep>	0	1	3	6	8	9

Table 5.12 Frequent 2-itemset for <d> by Thread-4

Itemset	TID's						
<dc>	0	3	4	5	7	8	9
<de>	0	3	4	7	8	9	
<dp>	0	3	4	5	7	8	9

Table 5.13 Frequent 3-itemset for <d> by Thread-4

Itemset	TID's						
<dce>	0	3	4	7	8	9	
<dcp>	0	3	4	5	7	8	9
<dep>	0	3	4	7	8	9	

Table 5.14 Frequent 4-itemset for <d> by Thread-4

Itemset	TID's					
<dcep>	0	3	4	7	8	9

Table 5.15 Frequent 2-itemset for <m> by Thread-5

Itemset	TID's						
<me>	0	1	2	6	7	8	9
<mp>	0	1	6	7	8	9	

Table 5.16 Frequent 3-itemset for <m> by Thread-5

Itemset	TID's					
<mep>	0	1	6	7	8	9

Table 5.17 Frequent 2-itemset for <c> by Thread-6

Itemset	TID's						
<ce>	0	2	3	4	7	8	9
<cp>	0	3	4	5	7	8	9

Table 5.18 Frequent 3-itemset for <c> by Thread-6

Itemset	TID's					
<cep>	0	3	4	7	8	9

Table 5.19 Frequent 2-itemset for <e> by Thread-7

Itemset	TID's							
<ep>	0	1	3	4	6	7	8	9

Table 5.20 depicts the candidate items, frequent items, number of candidates and frequent items generated by the TB-NPF-VDF for *D*. The total number of candidate items generated using TB-NPF-VDF is 56, which is less when compared to VDF.

5.4 Experimental Results and Discussion

The runtime performance of all algorithms (Matrix-Apriori [PVG,06], VDF, NPF-VDF, TB-NPF-VDF) for the four datasets depicted in Table 1.4 with different *min_sup* percentage were tabulated in Table 5.21. The *min_sup* is varied from 20% to 70%. Figures 5.3 to 5.6 show the graphical representation of the runtime comparison between the algorithms viz., Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for the datasets, namely chess, mushroom, t25i10d10k and c20d10k, respectively.

Table 5.20 Details of Itemsets for D

Itemset	Candidate Items	Total[#]	Frequent Items	Total[$]
1-itemset	{a, b, c, d, e, f, g, h, i, k, m, p, m}	13	{a, c, d, e, f, i, m, p}	8
2-itemset	{af, ai, ad, am, ac, ae, ap, fi, fd, fm, fc, fe, fp, id, im, ic, ie, ip, dm, dc, de, dp, mc, me, mp, ce, cp, ep}	28	{ad, ac, ap, fp, ie, ip, dc, de, dp, me, mp, ce, cp, ep}	14
3-itemset	{adm, adc, ade, adp, ace, acp, iep, dce, dcp, dep, mep, cep}	12	{adc, adp, acp, iep, dce, dcp, dep, mep, cep}	9
4-itemset	{adce, adcp, dcep}	3	{adcp, dcep}	2
Total		**56**		**33**

[#] Number of Candidate Items [$] number of Frequent Items

From Table 5.21 and Figures 5.3 to 5.6, the TB-NPF-VDF outperforms than the other existing methods viz., Matrix-Apriori, VDF and NPF-VDF. On an average, the runtime is reduced from 20.3092 to 9.9094.

5.4.1 Welch's Two Sample *t*-test

To prove statistically, a Welch's two sample *t*-test is being performed between the runtimes of Matrix-Apriori and TB-NPF-VDF and it is used to determine whether the means of the two groups are equal to each other or not. The null hypothesis is taken as that the two means are equal i.e. $\mu_1 = \mu_2$, and the alternative is that they are not equal i.e. $\mu_1 < \mu_2$ or $\mu_1 > \mu_2$ or $\mu_1 \neq \mu_2$. The test is performed using the R tool for

each dataset shown in Table 1.4, and the results are tabulated in Table 5.22. From Table 5.22, it was observed that the *p*-values for all datasets are ≤ 0.05 (5%) and it is concluded that the two means are not equal, which means that there are significant differences between the runtimes. Therefore, the proposed method TB-NPF-VDF is more efficient in terms of runtime than the others.

Table 5.21 Performance Results of TB-NPF-VDF in seconds

min_sup (%)	**Runtime (in Sec.)**			
	Matrix-Apriori	**VDF**	**NPF-VDF**	**TB-NPF-VDF**
		chess		
20	20.7578	16.8578	13.3578	6.5267
30	19.6365	16.0452	12.1455	5.0325
40	17.7750	14.0750	10.0720	4.5635
50	16.3028	13.3017	9.0017	3.2634
60	15.3625	12.7943	8.2934	2.4571
70	14.8546	11.9825	7.4822	2.0012
Average	**17.4482**	**14.1761**	**10.0588**	**3.9741**
		mushroom		
20	23.2135	21.1215	18.0016	12.1024
30	21.3426	20.0462	17.0642	11.5642
40	20.0035	19.7083	14.1038	10.7869
50	19.2002	18.2058	13.2044	10.0063
60	18.0805	17.7898	12.7240	8.5698
70	17.5652	15.9575	11.4530	7.9586
Average	**19.9009**	**18.8049**	**14.4252**	**10.1647**
		t25i10d10k		
20	25.2145	23.3254	20.3325	15.1267
30	23.9625	21.4578	19.4258	13.9568
40	21.5467	20.0025	17.9857	12.0127
50	20.3859	18.7621	16.2456	11.6321
60	19.5321	18.0056	15.0012	10.5212
70	18.4521	16.0527	13.7564	9.2451
Average	**21.5156**	**19.6010**	**17.1245**	**12.0824**
		c20d10k		
20	26.0014	24.4253	22.8342	17.7586
30	24.9532	22.6752	21.5062	15.9802
40	22.4251	21.9546	20.0412	13.7542
50	21.5621	19.4316	18.8562	11.9892
60	20.1425	19.0012	17.0124	11.0016
70	19.1478	17.5242	15.9351	10.0142
Average	**22.3720**	**20.8354**	**19.3642**	**13.4163**
Overall Average (All Datasets)	**20.3092**	**18.3543**	**15.2432**	**9.9094**

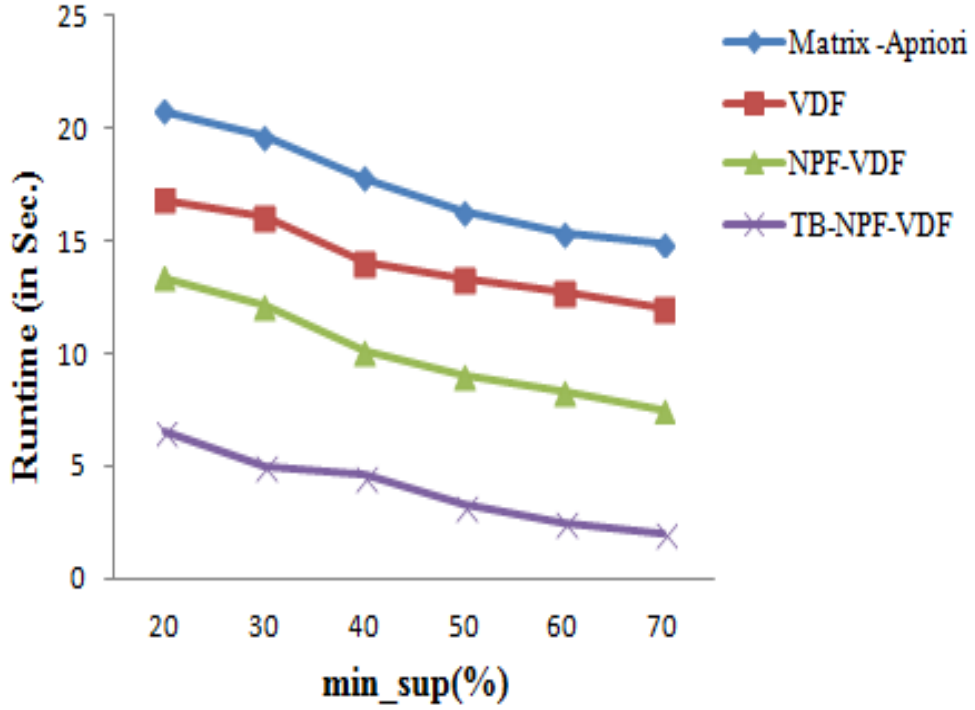


Figure 5.3 Runtime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for chess Dataset

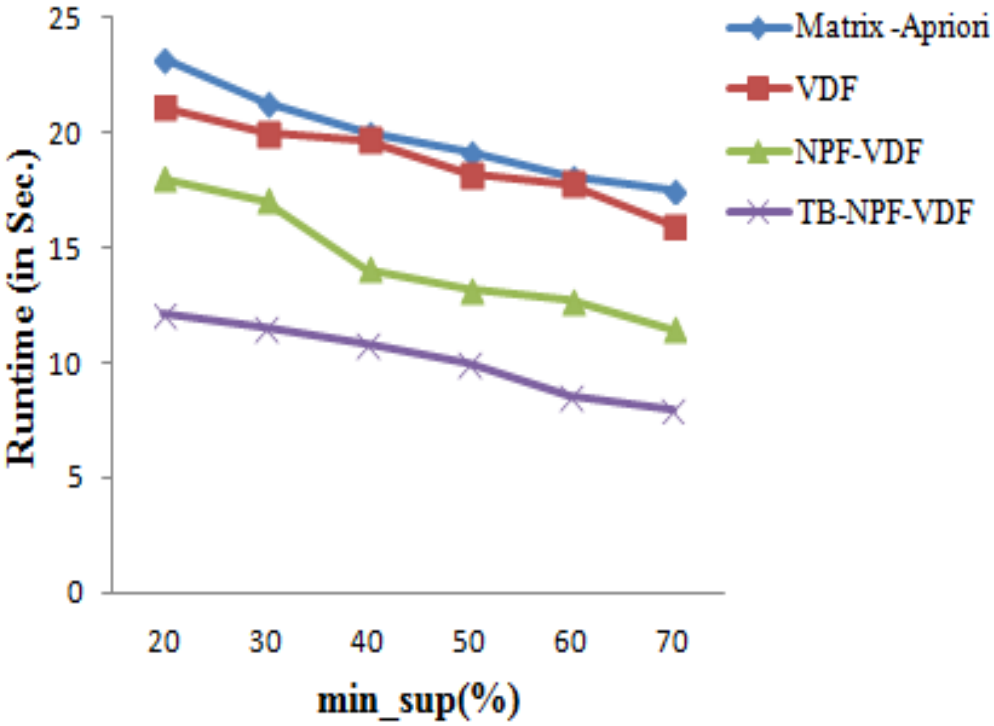


Figure 5.4 Runime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for mushroom Dataset

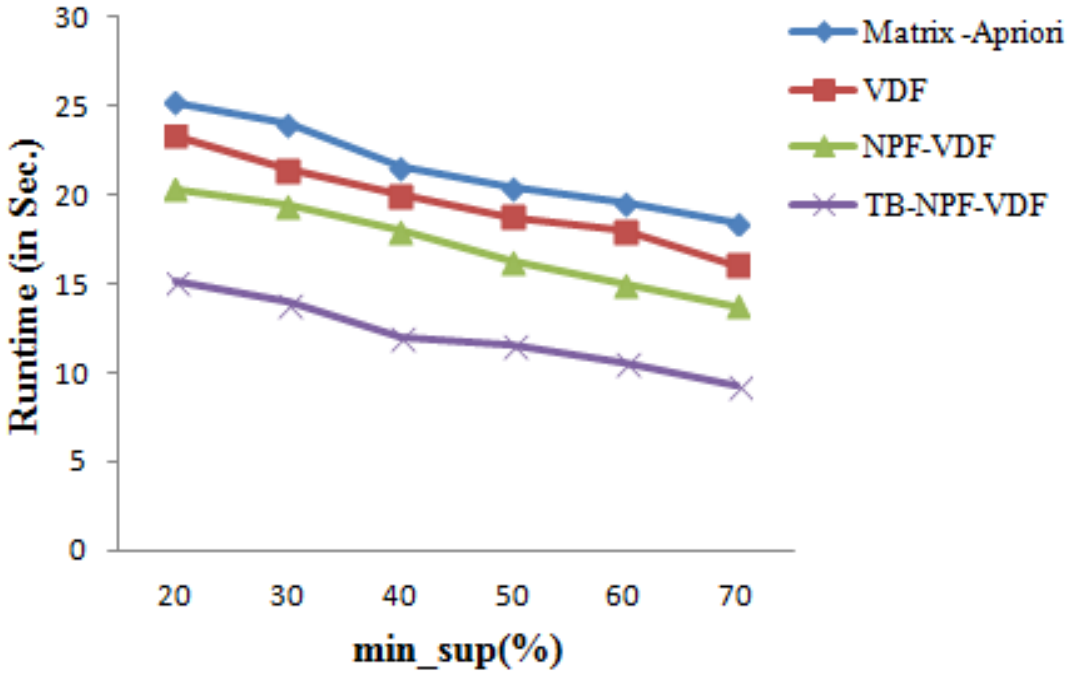


Figure 5.5 Runtime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for t25i10d10k Dataset

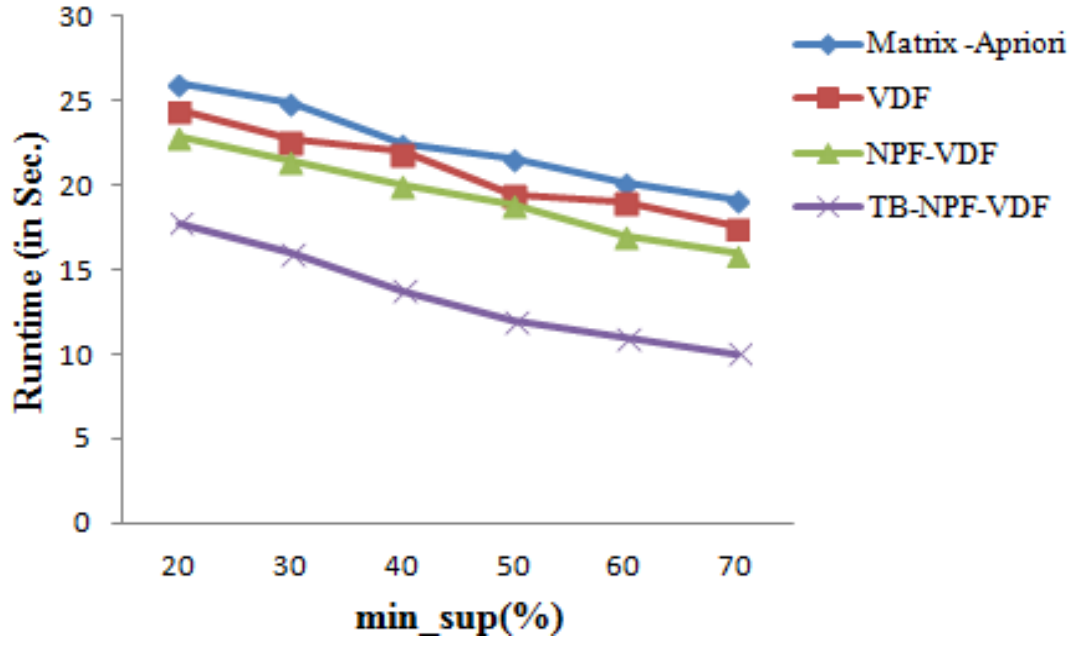


Figure 5.6 Runtime of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for c20d10k Dataset

Table 5.22 Results of *t*-test

Dataset	*p*-value
chess	1.207×10^{-06}
mushroom	6.785×10^{-06}
t25i10d10k	5.611×10^{-05}
c20d10k	0.0002914

The reason for enhancing the performance is that the concurrent tasks using a multithreaded approach speeds applications up, reduce the time required for execution and utilizes CPU effectively. With novel pattern generation, it generates less number of candidate itemsets than the existing ones. Further, it scans the database only once during the entire process.

5.5 Chapter Summary

Numerous FPM algorithms have been introduced in the field of Data Mining. Each algorithm has its own merits and demerits and not suits for many real-life scenarios. In this research article, a new approach, TB-NPF-VDF has been introduced to discover the frequent patterns that combine the power of VDF, NPF and multithread concept in an efficient way. Experiments were carried out with real-time datasets using Python implementation for the existing and proposed method, and it has been proved that the TB-NPF-VDF outperforms the other sequential approaches in terms of execution time and memory. The main advantage of this method is that it discovers the frequent patterns with less amount of time and saves memory with jagged array representation for the VDF matrix. Though, the TB-NPF-VDF reduces the runtime and memory with multithreading and jagged array, the multithreading has inherent demerits, thus the usage of GPU has been introduced in the next chapter for reducing the runtime and memory significantly than the TB-NPF-VDF.

Chapter - 6

GNVDF: A GPU-ACCELERATED NOVEL ALGORITHM USING VERTICAL DATA FORMAT AND JAGGED ARRAY

CHAPTER - 6

GNVDF: A GPU-ACCELERATED NOVEL ALGORITHM USING VERTICAL DATA FORMAT AND JAGGED ARRAY

Research is to see what everybody else has seen, and to think what nobody else has taught

-- Albert Szent-Gyorgyi

6.1 Background

Data Mining (DM) is a part of Knowledge Discovery in Databases (KDD) [HD,16] and explores the hidden patterns from transactional databases for making business decisions. It is being associated with many fields such as database systems, data warehousing, statistics, machine learning, information retrieval, and high-level computing [HPK,12],[LS,20]. It is also supported by other sciences like neural networks, pattern recognition, spatial data analysis, image databases and signal processing [HPK,12],[LS,20]. Frequent Pattern Mining (FPM) is a computationally crucial step in DM [VA,15]. It is used to determine the frequent patterns and associations from databases such as relational and transactional databases and other data repositories. The Apriori is one of the most significant algorithms, which generate the frequent itemsets for the boolean association rule. It has many problems such as more database scan and I/O cost, a large amount of time and memory in finding frequent itemsets. So, the researchers have done several enhancements to Apriori in the last two decades.

However, enhancing execution speed and reducing memory requirements are the essential parameters while determining the frequent patterns nowadays because of the rise of big data in various domains and sources in human endeavour. Also, when

the transactional database size increases, demand for storage is increased and requires high-speed algorithms to find frequent patterns. But with a single-threaded approach, it's tough to minimize time. The GPU accelerated computing employs GPUs along with CPUs. It enables superior performance by supporting a parallel programming paradigm with multiple cores. It saves time and cost in scientific and other high computing tasks [AFB[b],14].

Thus, the research work introduced in this chapter uses GPU acceleration for finding the frequent patterns with Novel pattern formation using Vertical Data Format (GNVDF). In this, the candidate *i*-itemsets is divided into two buckets viz., Bucket-1 and Bucket-2. Bucket-1 contain all the possible items to form candidate-(*i*+1) itemsets. Bucket-2 has the items that cannot include in the candidate-(*i*+1) itemsets. It also employs a compact data structure called jagged array to minimize the memory requirement and also remove common transactions among the frequent 1-itemsets. It also utilizes a vertical representation of data for efficiently extracting the frequent itemsets by scanning the database only once. Further, the GPU acceleration enhances the execution speed of the algorithm. The proposed algorithm was implemented using Python and tested with four standard benchmark datasets and compared the same without the GPU usage. The comparison result revealed that GNVDF with GPU acceleration is faster by 94% than the method without GPU acceleration.

6.2 Graphical Processing Unit

It is a device specifically designed for graphics processing. Two types of GPUs exist in the market are i) integrated and ii) discrete. The integrated GPUs are embedded alongside the CPU whereas the discrete GPUs comes as a distinct chip built up in a separate circuit board and is typically attached to a PCI express slot.

GPUs are widely used in large-scale hashing and matrix computations because it supports parallelism and serve as the base for mining and machine learning. CUDA and OpenCL are two popular GPGPU programming framework tools. NVIDIA has designed a parallel computing platform and programming called Compute Unified Device Architecture (CUDA) [LSHW,15],[WDY,13]. The CUDA-based program can only be run on the NVIDIA-produced GPU. A typical CPU may contain four or eight cores, an NVIDIA GPU consists of thousands of CUDA cores and a pipeline that supports parallel processing on thousands of threads, increasing the speed significantly.

With Numba, the Python developer can quickly enter into GPU-accelerated computing. It makes use of both GPU and CPU to facilitate processing-intensive operations viz., deep learning, analytics, and engineering applications. The CUDA Python and Numba help to enhance the speed by targeting both CPUs and NVIDIA GPUs. With this advantage of CUDA python and Numba, the implementation of this proposed work will be GPU accelerated. Numba is compatible with Windows 7 and later (32-bit and 64-bit), Python 3.6 or later, and Numpy versions 1.15 or later.

6.2.1 Processing Flow of CUDA

In a typical CUDA programming, the data is first sent from the main memory to the GPU memory, then the CPU sends instructions to the GPU, then the GPU schedules and executes the kernel on the available parallel hardware, and finally resulting data are copied back from the GPU memory to the main memory. The processing flow of CUDA is illustrated in Figure 6.1. When using CUDA, the developers can program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few keywords.

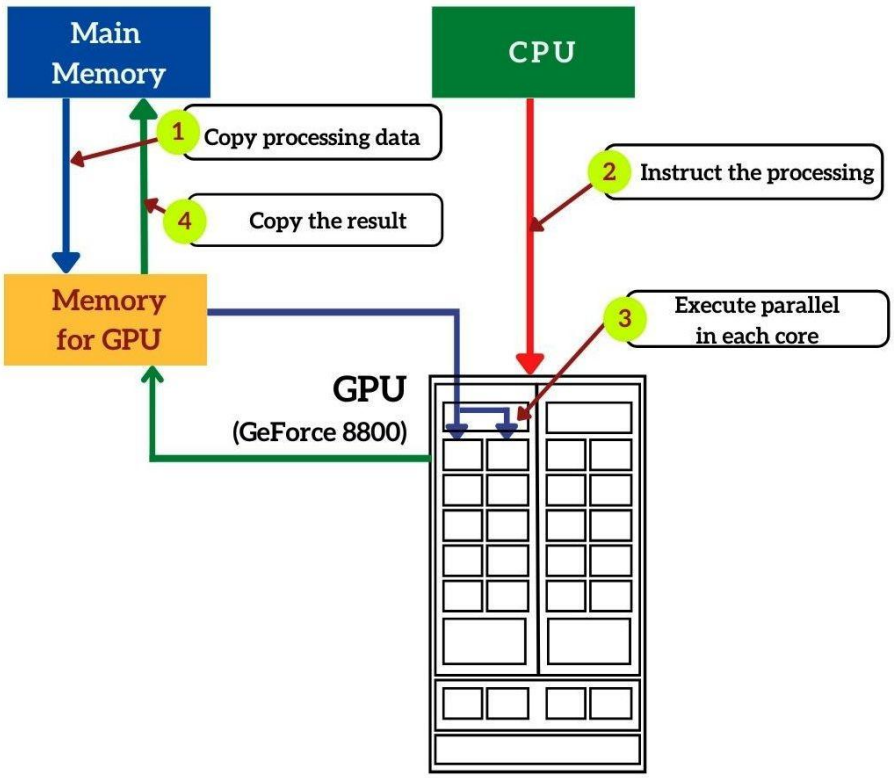


Figure 6.1 Processing Flow of CUDA

6.3 Proposed Methodology

The main objective of the proposed work is to find the essential frequent itemsets from the transactional database with less memory space and time by ignoring the least probable ones. The method used a jagged array storage structure [WXXS,18] and GPU to minimize memory usage and execution time. The proposed method GNVDF[4] first removes the null/void transactions in the dataset. Null/void transactions are those which contain only one item. Then the dataset is scanned once and converted into VDF format.

[4]**Sumathi**, S.Murugan, "GNVDF: A GPU-accelerated Novel Algorithm for Finding Frequent Patterns Using Vertical Data Format Approach and Jagged Array", International Journal of Modern Education and Computer Science (IJMECS), ISSN: 2075-0161 (Print), ISSN: 2075-017X (Online), Vol.13, No.4, pp. 28-41, August 2021. DOI: 10.5815/ijmecs.2021.04.03 (**UGC Care List - II, Scopus Indexed**).

The Support Count (*SC*) for each item is calculated by counting the number of transactions that contain each item. Now the candidate 1-itemset C_1 is formed. Next, the frequent 1-itemset is formed by removing the items whose $SC<min\_sup(\delta)$ and storing it in jagged array representation [SM,18] in sorted order based on *SC*. From L_1 the common transactions among all items are determined either by intersecting or ANDing the transaction in each item, and it is preserved in the Common Transaction List (C_{TID_list}). The transactions in C_{TID_list}'s are removed from each item in L_1, forming the final frequent 1-itemset. The *SC* for each item in L_1 is updated by $SC - n$, where n is the number of transactions in C_{TID_list}. Next, the new *min_sup* (δ_{new}) is determined as $\delta_{new} = \delta - n$, and it will be the *min_sup* from the 2nd iteration onwards.

Before finding the frequent 2-itemset, the final frequent 1-itemset is divided into two logical buckets, LB_1 and LB_2 respectively. LB_1 contains all the items whose $SC = \delta_{new,}$ and the rest will be placed in LB_2. The itemset combinations among the items in LB_1 are least probable of being a candidate 2-itemset because the *SC* of each item is equal to δ_{new}. So it is not considered for generating candidate 2-itemset. The candidate 2-itemsets patterns are generated by combining each item I_x in LB_1 with each item I_y in LB_2 and each item I_z in LB_2 with I_{z+1} in LB_2 until the last item in LB_2. The itemset combination that ends with the last item in LB_2 will be placed in C_{2_2} and the rest in C_{2_1}. From C_{2_1} and C_{2_2}, the items whose *SC* below the δ_{new} is removed as infrequent and formed L_{2_1} and L_{2_2}.

For generating candidate 3-itemset, each itemset I_x in L_{2_1} is combined with the next item I_y in LB_2 after the last item in I_x. Similar to the previous iteration, the combinations that end with the last item in LB_2 are placed in C_{3_2} and the

rest in $C_{3\_1.}$ It is noted that the itemset combinations in L_{2_2} are not used in the formation of candidate 3-itemsets. The L_{3_1} and L_{3_2} were formed by removing the infrequent itemsets in C_{3_1} and C_{3_2}. The process is continued until L_{n_1} is not null. Further, to increase the execution speed of the proposed method, it is being accelerated with GPU. The proposed algorithm is shown in Algorithm 6.1, and the workflow diagram in Figure 6.2.

Algorithm 6.1 GNVDF: An algorithm for finding frequent itemsets

Input : D - a dataset with n transactions;

δ - minimum support threshold;

Output : Frequent patterns;

1: $D \leftarrow$ eliminate_null(D);

2: $vdf \leftarrow$ scan D and convert it in VDF;

3: $L_1 \leftarrow$ one_frequent_itemset(vdf, δ);

4: $C_{TID\_list} \leftarrow$ find_common_TID(L_1);

5: $L_1 \leftarrow$ remove the transactions in C_{TID_list} for each item in L_1;

6: $\delta_{new} \leftarrow \delta$ - number of transactions in C_{TID_list};

7: $LB_1 \leftarrow \{\forall$ frequent 1-itemset $| SC=\delta_{new} \}$;

8: $LB_2 \leftarrow \{\forall$ frequent 1-itemset $| SC > \delta_{new}\}$;

9: $L_{2\_1}, L_{2\_2} \leftarrow$ find_two_freq_itemset(LB_1,LB_2,δ_{new});

10: i=2;

11: **while** $L_{i\_1} \neq \emptyset$ **do**

12: L_{i+1_1},$L_{i+1\_2} \leftarrow$ n_frequent_itemset(L_{i_1},LB_2,δ_{new});

13: $i=i+1$;

14: **end while**

procedure eliminate_null(*D*-a dataset with *n* transactions)

1: **for each** $T_i \in D$ **do**

2: $cnt \leftarrow$count the number of items in T_i;

3: **if** $cnt == 1$ **then**

4: remove T_i from D;

5: **end if**;

6: **end for**;

7: return D;

procedure one_frequent_itemset(*D*: Dataset after removing null transactions; δ :minimum support threshold)

1: $L_1 \leftarrow \emptyset$;

2: **for each** $item_i$ in D **do**

3: $TID_{list} \leftarrow$transactions in which $item_i$ occurs;

4: $SC \leftarrow$count the number of transactions in TID_{list};

5: **if** $SC \geq \delta$ **then**

6: add $\{item_i, TID_{list}, SC\}$ into L_1;

7: **end if**

8: **end for**

9: sort L_1 and store it in jagged array format;

10: return L_1;

procedure find_common_TID (L_1: frequent 1-itemset)

1: $n \leftarrow$find the number of items in L_1;

2: $C_{TID\_list} \leftarrow \{TID_{list1} \cap TID_{list2} \cap \ldots \cap TID_{listn}\}$;

3: return C_{TID_list};

procedure two_freq_itemset (LB_1: frequent 1-itemset1, LB_2: frequent 1-itemset2, δ:minimum support)

1: $last_item \leftarrow$find last item in LB_2;

2: **for each** $item_i$ in LB_1 **do**

3: **for each** $item_j$ in LB_2 **do**

4: $new\_pattern \leftarrow <item_i item_j>$;

5: $new_tid \leftarrow$TIDs($item_i$)$\cap$TIDs($item_j$);

6: *new_sc*←count the transactions in *new_tid*;

7: **if** *new_pattern* contains *last_item* **then**

8: append{*new_pattern*,*new_tid*,*new_sc*} in C_{2_2};

9: **else**

10: append{*new_pattern*,*new_tid*,*new_sc*} in C_{2_1};

11: **end if**

12: **end for**

13: **end for**

14: $L_{2\_1} \leftarrow \{C_{2\_1} \mid SC(C_{2\_1}) \geq \delta\}$;

15: $L_{2\_2} \leftarrow \{C_{2\_2} \mid SC(C_{2\_2}) \geq \delta\}$;

16: return L_{2_1}, L_{2_2};

procedure n_frequent_itemset (L_{i_1}: frequent *i*-itemset1, LB_2: frequent 1-itemset2, δ_{new}: minimum support)

1: **for each** $item_i$ in L_{i_1} **do**

2: *last_item*←find the last item in $item_i$;

3: **for each** $item_j$ in LB_2 after *last_item* **do**

4: *new_item*←{<$item_i item_j$>};

5: *new_tid*←TIDs($item_i$)∩ TIDs($item_j$);

6: *new_sc*←count the transactions in *new_tid*;

7: **if** *new_item* contains last element in LB_2 **then**

8: append{*new_item*,*new_tid*,*new_sc*}in C_{n_2};

9: **else**

10: append{*new_item*,*new_tid*,*new_sc*}in C_{n_1};

11: **end if**

12: **end for**

13: **end for**

14: $L_{n\_1} \leftarrow \{C_{n\_1} \mid SC(C_{n\_1}) \geq \delta\}$;

15: $L_{n\_2} \leftarrow \{C_{n\_2} \mid SC(C_{n\_2}) \geq \delta\}$;

16: return L_{n_1}, L_{n_2};

The main advantage of the proposed method is that it reduces the number of candidate itemsets to be generated in each iteration because the itemsets in L_{i_2}, for $i \geq 3$ will not be considered for creating candidate itemsets and removal of items in *CTL* in final L_1. Additionally, GPU and jagged array enhance the performance in terms of speed and usage of memory.

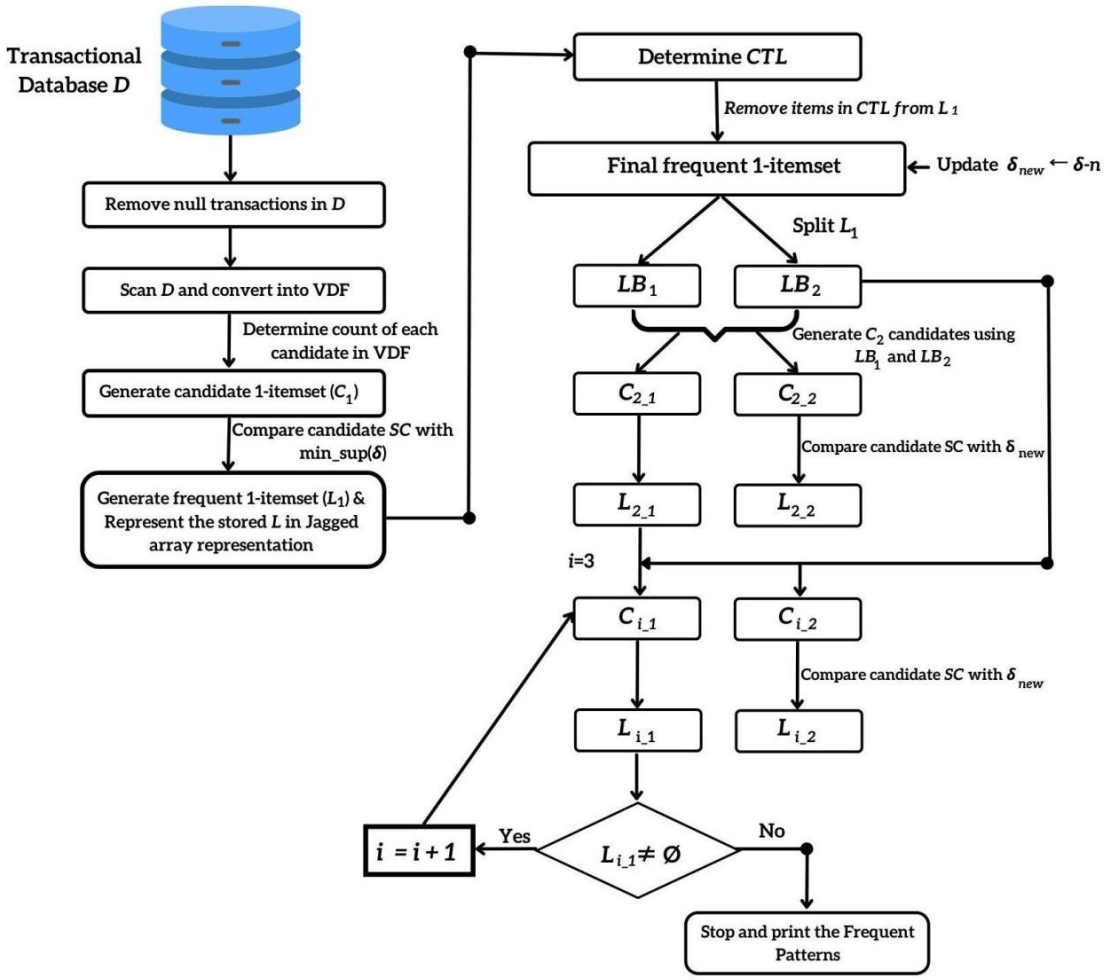


Figure 6.2 Workflow of GNVDF

6.3.1 Memory Requirement Calculation

From [DMPW$^+$,10],[SM,18], it was observed that the memory requirement using a jagged array structure for the frequent itemsets could be calculated based on the following equation:

$$TM = \sum_{i=1}^{itemset_i \neq \phi} TM_i - rbytes_i \quad \text{… Equation (6.1)}$$

where, TM_i is the total memory required for the candidate *i*-itemset, and $rbytes_i$ is the memory occupied by the infrequent/rare items in the candidate *i*-itemset. By subtracting $rbytes_i$ from TM_i, the memory for L_i i.e. frequent *i*-itemsets can be found. TM_i and $rbytes_i$ were calculated using Equations 6.2 and 6.3 respectively.

$$TM_i = \sum_{\forall item \in \{itemset_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \text{… Equation (6.2)}$$

$$rbytes_i = \sum_{\forall item \in \{in-frequent_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad \text{… Equation (6.3)}$$

As in [DMPW$^+$,10], the GNVDF also used the same jagged storage structure for storing frequent itemsets, and the amount of memory requirement was calculated as follows. It first fetches the common transactions among items in the frequent 1-itemsets and then removes them from frequent 1-itemsets. Suppose if the frequent 1-itemset contains *n* items say $item_1$, $item_2$, $item_3$,…, $item_n$ and the corresponding *TID* lists say $TID\text{-}List_1$, $TID\text{-}List_2$, $TID\text{-}List_3$,…,$TID\text{-}List_n$, then the common *TID*s(C_{TID}) among the *n* items were found by set intersection operation using Equation 6.4 shown below.

$$C_{TID} = \{TID - List_1\} \cap \{TID - List_2\} \cap ... \cap \{TID - List_n\} \quad \text{… Equation (6.4)}$$

The memory space required for C_{TID} was calculated using Equation 6.5.

$$CM = \sum_{i=1}^{length(C_{TID})} sizeof(C_{TID_i}) \quad \text{… Equation (6.5)}$$

Since the method removes the C_{TID} from frequent 1-itemsets, the C_{TID} need not be repeated in the subsequent frequent itemsets, saving memory space considerably.

The amount of memory saved (*MS*) for the entire dataset was calculated using Equation 6.6.

$$MS = count(itemset_i) \times CM + \sum_{i=2}^{itemset_i \neq \phi} \{count(itemset_{i-1}) + count(itemset_{i-2})\} \times CM$$

...Equation (6.6)

where, $count(itemset_1)$, $count(itemset_{i\_1})$, and $count(itemset_{i\_2})$ refer to the number of items in frequent 1-itemset, first and the second part of frequent *i*-itemsets, respectively. Thus, the total memory required for the frequent itemsets of the entire dataset using the proposed method was calculated using Equation 6.7.

$$TM_{final} = \{ \sum_{i=1}^{itemset_i \neq \phi} TM_i - rbytes_i \} - MS \quad \text{... Equation (6.7)}$$

6.3.2 Illustration by an Example

The vertical representation of transactional dataset *D* shown in Table 6.1 is considered for illustrating the proposed methodology.

Table 6.1 Vertical Data Format of D

Item	Transaction ID's (TID's)
a	{3, 4, 5, 7, 8, 9}
b	{1, 3, 4, 5, 6}
c	{0, 2, 3, 4, 5, 7, 8, 9}
d	{0, 3, 4, 5, 7, 8, 9}
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}
f	{1, 3, 5, 6, 8, 9}
g	{0, 1, 3}
h	{0, 1, 5, 6, 9}
i	{0, 1, 3, 6, 8, 9}
k	{0, 7}
m	{0, 1, 2, 6, 7, 8, 9}
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}

The transaction database *D* contains 12 items viz., {a, b, c, d, e, f, g, h, i, k, m, p}. Each item is represented by a row containing the name of the item and the

transactions in which the item occurs (*TID*s) [SNM,15]. Let δ is 6. From Table 6.1, the candidate 1-itemset is calculated. The candidate 1-itemset contains all the items in *D*, the *TID*s in which the item occurs and the *SC*. It is shown in Table 6.2.

Table 6.2 Candidate 1-itemset(C_1)

Item	TIDs	SC
a	{3, 4, 5, 7, 8, 9}	6
b	{1, 3, 4, 5, 6}	5
c	{0, 2, 3, 4, 5, 7, 8, 9}	8
d	{0, 3, 4, 5, 7, 8, 9}	7
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}	9
f	{1, 3, 5, 6, 8, 9}	6
g	{0, 1, 3}	3
h	{0, 1, 5, 6, 9}	5
i	{0, 1, 3, 6, 8, 9}	6
k	{0, 7}	2
m	{0, 1, 2, 6, 7, 8, 9}	7
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}	9

From the table above, the items viz., b, g, h and k are removed as infrequent because the items do not satisfy δ. The frequent 1-itemset is shown in Table 6.3. Since the common transactions (*CTL*) are stored in Table 6.4, they are removed from each item in L_1, the final L_1 is formed, and it is shown in Table 6.5. Now the *new_min* is calculated by removing the number of items in *CTL* as $\delta_{new} = \delta - n = 6 - 2 = 4$. The logical buckets from final L_1, i.e. LB_1 and LB_2, are shown in Tables 6.6 and 6.7.

Table 6.3 Frequent 1-itemset (L_1)

1-itemset	TIDs								
a	3	4	5	7	8	9			
f	1	3	5	6	8	9			
i	0	1	3	6	8	9			
d	0	3	4	5	7	8	9		
m	0	1	2	6	7	8	9		
c	0	2	3	4	5	7	8	9	
e	0	1	2	3	4	6	7	8	9
p	0	1	3	4	5	6	7	8	9

To reduce the storage space requirement further, this method finds the common transaction in which the all items occurs either by AND operation or intersection of the *TID*s of all frequent 1-itemset. i.e. {3,4,5,7,8,9}∩{1,3,5,6,8,9}∩{0,1,3,6,8,9}∩{0,3,4,5,7,8,9}∩{0,1,2,6,7,8,9}∩{0,2,3,4,5,7,8,9}∩{0,1,2,3,4,5,6,7,8,9}∩{0,1,3,4,5,6,7,8,9} = {8,9} and it is stored in *CTL*. The *CTL* is shown in Table 6.4.

Table 6.4 Common Transaction List (CTL)

CTL	
8	9

Table 6.5 Final Frequent 1-itemset (L_1)

1-itemset	TIDs						
a	3	4	5	7			
f	1	3	5	6			
i	0	1	3	6			
d	0	3	4	5	7		
m	0	1	2	6	7		
c	0	2	3	4	5	7	
e	0	1	2	3	4	6	7
p	0	1	3	4	5	6	7

Table 6.6 Logical Bucket-1 (LB_1)

1-itemset	TIDs			
a	3	4	5	7
f	1	3	5	6
i	0	1	3	6

Table 6.7 Logical Bucket-2 (LB_2)

1-itemset	TIDs						
d	0	3	4	5	7		
m	0	1	2	6	7		
c	0	2	3	4	5	7	
e	0	1	2	3	4	6	7
p	0	1	3	4	5	6	7

The 2-itemset combinations viz., ad, am, ac, ae, fd, fm, fc, fe, id, im, ic, ie, dm, dc, de, mc, me, mp, and ce are in C_{2_1} and the items viz., ap, fp, ip, dp, mp, cp and ep

are stored in C_{2_2}. The possible combinations viz., af, ai and fi need not be generated. It is shown in Tables 6.8 and 6.9 respectively.

Table 6.8 Candidate 2-itemset - Part I

C_{2_1}	TIDs	SC
ad	3, 4, 5, 7	4
am	7	1
ac	3, 4, 5, 7	4
ae	3,4,7	3
fd	3,5	2
fm	1,6	2
fc	3,5	2
fe	1,3,6	3
id	0,3	2
im	0,1,6	3
ic	0,3	2
ie	0, 1, 3, 6	4
dm	0	1
dc	0, 3, 4, 5, 7	5
de	0, 3, 4, 7	4
mc	0,2	2
me	0, 1, 2, 6, 7	5
ce	0, 2, 3, 4, 7	5

The items viz., am, ae, fd, fm, fc, fe, id, im, ic, dm and mc are infrequent in C_{2_1} and no item is infrequent in C_{2_2}. Therefore, the frequent 2-itemsets are stored in L_{2_1} and L_{2_2} in jagged array notation as shown in Tables 6.10 and 6.11 respectively.

Table 6.9 Candidate 2-itemset - Part II

C_{2_2}	TIDs	SC
ap	3, 4, 5, 7	4
fp	1, 3, 5, 6	4
ip	0,1,3,6	4
dp	0, 3, 4, 5, 7	5
mp	0, 1, 6, 7	4
cp	0, 3, 4, 5, 7	5
ep	0, 1, 3, 4, 6, 7	6

The candidate 3-itemsets from L_{2_1} and LB_2 viz., adm, adc, ade, ace and dce,

stored in C_{3_1} and the patterns adp, acp, iep, dep, mep, dcp and cep are kept in C_{3_2} as shown in Tables 6.12 and 6.13 respectively.

Table 6.10 Frequent 2-itemset - Part I

L_{2_1}	TIDs				
ad	3	4	5	7	
ac	3	4	5	7	
ie	0	1	3	6	
dc	0	3	4	5	7
de	0	3	4	7	
me	0	1	2	6	7
ce	0	2	3	4	7

Table 6.11 Frequent 2-itemset - Part II

L_{2_2}	TIDs					
ap	3	4	5	7		
fp	1	3	5	6		
ip	0	1	3	6		
dp	0	3	4	5	7	
mp	0	1	6	7		
cp	0	3	4	5	7	
ep	0	1	3	4	6	7

Table 6.12 Candidate 3-itemset - Part I

C_{3_1}	TIDs	SC
adm	7	1
adc	3, 4, 5, 7	4
ade	3,4,7	3
ace	3,4,7	3
dce	0, 3, 4, 7	4

Table 6.13 Candidate 3-itemset - Part II

C_{3_2}	TIDs	SC
adp	3, 4, 5, 7	4
acp	3,4,5,7	4
iep	0, 1, 3, 6	4
dep	0, 3, 4, 7	4
mep	0, 1, 6, 7	4
dcp	0, 3, 4, 5, 7	5
cep	0, 3, 4, 7	4

The L_{3_1} and L_{3_2} are shown in Tables 6.14 and 6.15, respectively. Similarly, C_{4_1} and C_{4_2} are shown in Tables 6.16 and 6.17, respectively. L_{4_1} and L_{4_2} are L_{4_1} = {} and L_{4_2} is shown in Table 6.18.

Table 6.14 Frequent 3-itemset - Part I

L_{3_1}	TIDs			
adc	3	4	5	7
dce	0	3	4	7

Table 6.15 Frequent 3-itemset - Part II

L_{3_2}	TIDs				
adp	3	4	5	7	
acp	3	4	5	7	
iep	0	1	3	6	
dcp	0	3	4	5	7
dep	0	3	4	7	
mep	0	1	6	7	
cep	0	3	4	7	

Table 6.16 Candidate 4-itemset - Part I

C_{4_1}	TIDs	SC
adce	3,4,7	3

Table 6.17 Candidate 4-itemset - Part II

C_{4_2}	TIDs	SC
adcp	3, 4, 5, 7	4
dcep	0, 3, 4, 7	4

Table 6.18 Frequent 4-itemset - Part II

L_{4_2}	TIDs			
adcp	3	4	5	7
dcep	0	3	4	7

Now, L_{4_1} is an empty list, so the algorithm terminates. It is observed from the experiment that the time needed for finding frequent items for sample dataset *D* in the example without the use of GPU is 0.8111 sec, whereas the wall time is 0.0073 ms

with GPU. The total memory requirement for the frequent itemset for the above dataset using the method in [SM,18] is *TM*=124+210+137+32=503 bytes. By using GNVDF, the memory requirement for the common transaction is *CM* = 2+2 = 4 bytes and the amount of memory saved using the proposed method is *MS* = (8×4) + {(7×4 + 7×4) + (2×4 + 7×4) + (0×4 + 2×4)} = 32 + 56 + 36 + 8 = 132 bytes. Therefore, the final memory requirement is TM_{final} = 503 - 132 = 371 which is 26.24% of memory saved for this example dataset compared to the memory requirement in [SM,18]. It is also noted that the number of common transactions is directly proportional to the amount of memory saved.

6.4 Experimental Results and Discussion

The proposed algorithm was implemented using Python with CUDA Toolkit with NVIDIA GPU. An extensive experiment was conducted using four real-time datasets viz., chess, mushroom, t25i10d10k and c20d10k to evaluate the performance of GNVDF. The datasets and their details were shown in Table 1.4. They were obtained from the FIMI repository and an open-source data mining library. The reason for choosing those datasets is that many researchers used those bench-mark datasets in Frequent Itemset Mining (FIM) and Association Rule Mining (ARM) based research. The runtime performance of the proposed method without GPU acceleration was obtained for each dataset, with the minimum threshold values ranging from 20% to 70% and is shown in Table 6.19.

From Tables 6.19 and 6.20 it was observed that when the number of items and transactions in a dataset increases, the time required for finding frequent patterns also increases. In general, there is an inverse relationship between the *min_sup* threshold and the time needed to determine the frequent patterns. i.e. when the *min_sup*

threshold is increased, the number of generated candidate itemsets, followed by frequent patterns, is minimized, consuming less time for the higher threshold.

Table 6.19 Runtime (in ms) Performance of the Proposed Algorithm without GPU

DS# → / MS* ↓	chess	mushroom	t25i10d10k	c20d10k
20	10759.6	14501.6	16332.5	16334.2
30	9845.5	13464.2	16225.8	16006.2
40	7972.0	11103.8	13885.7	15441.2
50	7101.7	10224.4	12645.6	14956.2
60	6293.4	9834.0	11101.2	13412.4
70	5082.2	8253.0	9256.4	12035.1
Average	**7842.4**	**11230.17**	**13241.2**	**14697.55**

#DS-Dataset *MS-*min_sup*(*δ*)

Similarly, the proposed algorithm was executed with GPU acceleration using the same minimum support range and results were tabulated in Table 6.20.

Table 6.20 Runtime (in ms) Performance of the Proposed algorithm with GPU-acceleration

DS# → / MS* ↓	chess	mushroom	t25i10d10k	c20d10k
20	119.5511	145.0160	161.7079	161.7248
30	107.0163	138.0940	156.0173	158.4772
40	83.9158	117.2770	129.7729	131.9761
50	73.2134	104.5091	108.3670	110.6496
60	64.2184	88.8096	102.4380	105.3511
70	53.4968	74.0512	83.6424	92.9924
Average	**83.57**	**111.29**	**123.66**	**126.86**

#DS-Dataset *MS-min_sup(*δ*)

The graphical representation of the runtime performance of each dataset with and without GPU usage was illustrated in Figures 6.3 to 6.6.

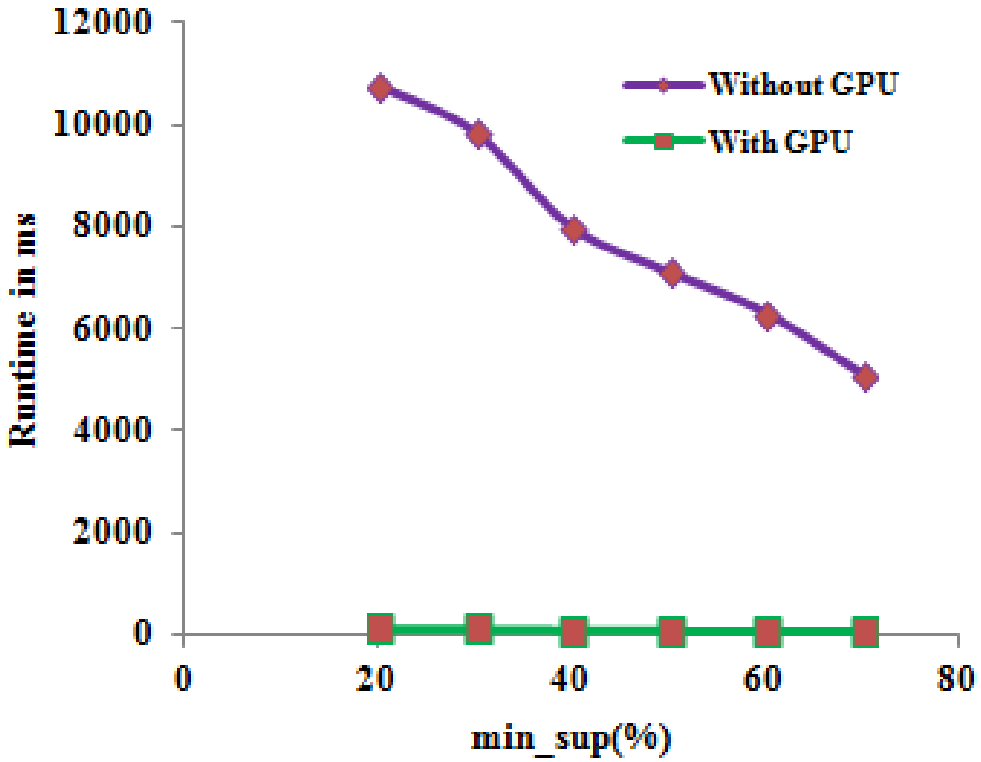


Figure 6.3 Runtime Performance of GNVDF with and without GPU-acceleration for chess Dataset

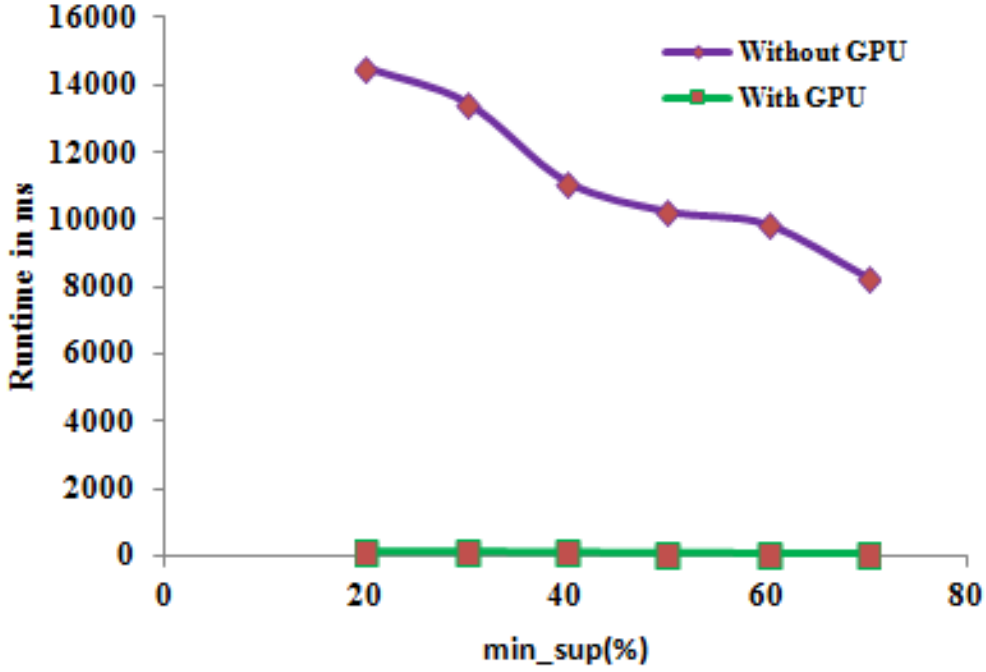


Figure 6.4 Runtime Performance of GNVDF with and without GPU-acceleration for mushroom Dataset

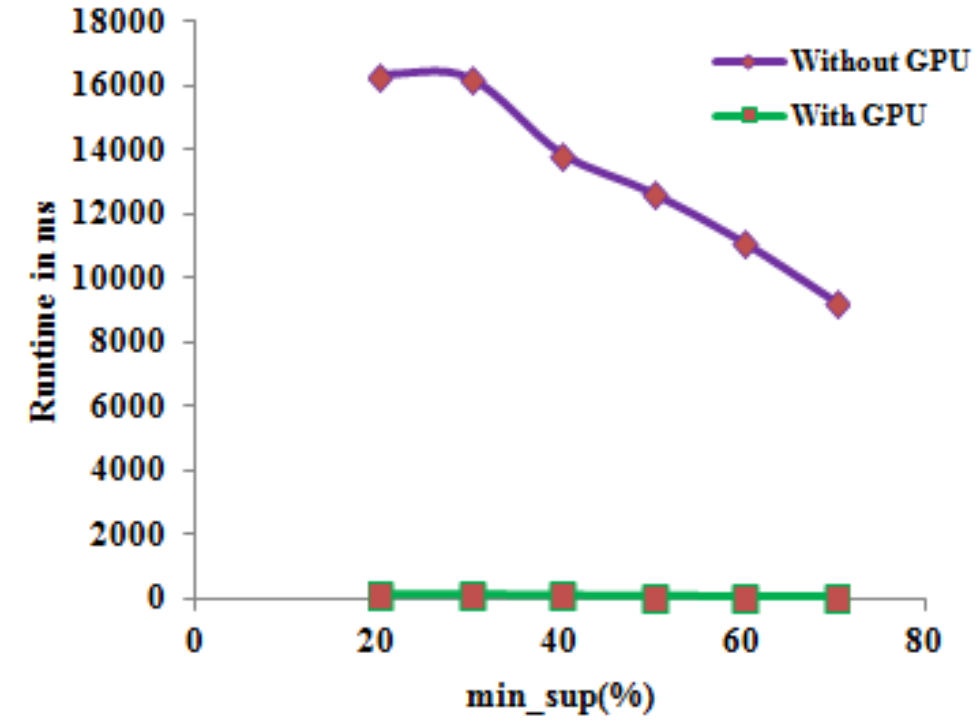


Figure 6.5 Runtime Performance of GNVDF with and without GPU-acceleration for t25i10d10k Dataset

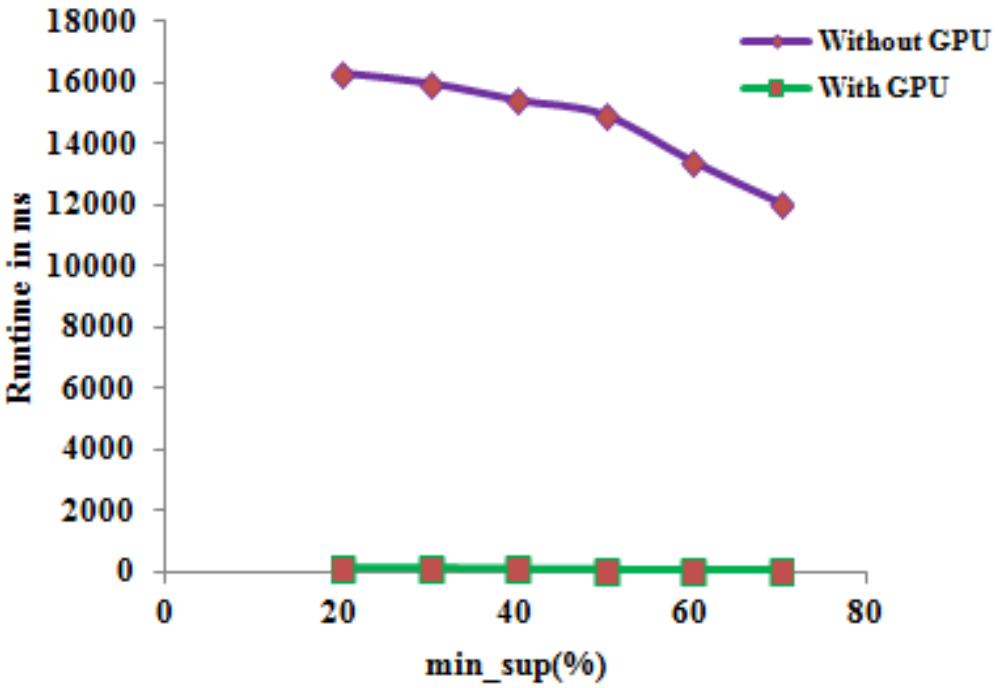


Figure 6.6 Runtime Performance of GNVDF with and without GPU-acceleration for c20d10k Dataset

Figures 6.3 through 6.6 revealed that the GPU acceleration significantly enables the execution speed of the proposed methodology, and GNVDF with GPU is faster by 94% when compared with GNVDF without GPU acceleration. The reason for the performance enhancement is that the GPUs have many computing cores that allow the parallel execution of computation-intensive tasks. Since the GNVDF uses the VDF approach, the number of database scans is restricted to one [SK,19] for determining each item's support count, which in turn reduces the overtime for finding the frequent patterns. But, VDF requires more memory for additional information like *TID*s than HDF [SK,19], so a jagged array has been used to minimize memory space is an advantage. Further, the elements in *CTL* removed from frequent 1-itemset save the memory space considerably more than the existing classical algorithms.

6.5 Chapter Summary

A GPU-accelerated novel method for finding the frequent itemset called GNVDF has been proposed in this research article. It uses an innovative approach to discover the candidate and frequent itemsets by removing unnecessary itemsets to form the subsequent itemsets. It also utilizes GPU for speeding up the process. It also empowers the use of a jagged array storage structure and removes the common elements in 1-frequent itemsets. With GPU acceleration and an innovative way of determining itemsets, the time required is significantly decreased. Similarly, with a jagged storage structure, the memory requirement is also minimized than the classical algorithms. From the extensive experiments made, it is observed that the GNVDF with GPU is 94% faster than with GNVDF without GPU and also proved that it suits both sparse and dense datasets. Further, the use of the VDF approach restricts the database scan to one.

Chapter - 7

CONCLUSION

CHAPTER - 7

CONCLUSION

In the end, when it's over, all that matters is what you've done

--Alexander the Great

The discovery of frequent patterns, associations, and correlation relationships among the huge amounts of data is useful in marketing, decision analysis, and business management. A popular application is "market basket analysis", which analyzes the buying behaviours of the customers by searching for itemsets that are bought together frequently. Many efficient and scalable algorithms have been contributed by the researchers for Frequent Pattern Mining (FPM), from which the correlation and association rules can be derived. Though there are two decades of research in FPM, the prolonged processing time and huge memory consumption have become the major issues. So, it necessitates developing better algorithms with reduced runtime and less memory usage. Thus, this research work concentrates on developing efficient FPM algorithms for finding frequent patterns in such a way that the runtime and usage of memory to be reduced than the existing algorithms.

7.1 Summary of the Contributions

In this thesis, a framework called SUMsFPM has been developed to minimize the runtime and memory usage in discovering the frequent patterns from transactional databases. The thesis mainly concentrates on two major issues associated with FPM and it contains three categories of research models viz., time-efficient (RISOTTO), memory-efficient (JAB-VDF) and both time and memory-efficient (TB-NPF-VDF and GNVDF) models. All models were implemented using Python programming.

The key contributions made in this research work are summarized below:

i) The prefixed-itemset storage structure proposed in the literature stores the frequent *i*-itemsets as <*prefix-key,* v*alues*>. It uses the *values* in the frequent *i*-itemset of prefixed-itemset storage for generating candidate (*i*+1)-itemset combinations, thereby reducing the number of candidate itemsets to be generated during each iteration. But for determining the Support Count (*SC*) of each candidate (*i*+1)-itemsets, it scans the dataset again and again. So, in order to reduce the number of database scans and candidate itemsets, the RISOTTO method proposed in the thesis combines the prefixed-itemset storage structure with Vertical Data Format (VDF) approach, which restricts the database scans to one. Further, the RISOTTO algorithm avoids storing the frequent *i*-itemsets with only one item in *values* because with one item, there is no possibility for (*i*+1)-itemset combinations which saves both time and memory. It is found from the experimental results that the RISOTTO algorithm outperforms the existing algorithms viz., prefixed-itemset storage and VDF i.e. RISOTTO reduces the runtime from 22.0163 to 13.5594 and from 18.3543 to 13.5594 seconds on an average when compared with prefixed-itemset storage and VDF respectively.

ii) It is noted that the VDF is faster and requires only one scan of the database than HDF. With the array storage structure adopted by VDF, the memory required for storing *tid*'s is huge. With a varied number of *tid*'s for each item, the memory was underutilized than the assigned. Thus, to save memory space considerably, JAB-VDF, a jagged array-based VDF has been proposed in this research work. Based on the experimental results, it has been observed that the

JAB-VDF reduces memory consumption from 1.5425 GB to 0.7609 GB on an average when compared with the 2-D array used by VDF with δ=20%.

iii) Many of the VDF-based research works in the literature is based on a single-threaded approach. It is noted that the multithreaded approach saves time to complete the task and also gives an improved throughput than the single-threaded approach. By considering these advantages, a multithreaded based FPM algorithm with a novel way of generating patterns using VDF called TB-NPF-VDF has been proposed in this thesis. With the extensive experiments, it has been identified that the TB-NPF-VDF reduced the runtime from 20.3092 to 9.9094 seconds on an average than the Matrix-Apriori. Similarly, the TB-NPF-VDF declined the runtime from 18.3543 to 9.9094, from 15.2432 to 9.9094 on an average when compared with VDF and NPF-VDF (proposed work with single-threaded approach) respectively. The usage of the jagged array in TB-NPF-VDF saves memory significantly as in JAB-VDF.

iv) The usage of multithreading in the TB-NPF-VDF method optimizes the processor usage and thereby increases the speed of the processes than the single-threaded approach. But, when the database size increases, it's tough to minimize runtime even with multithreading on a single CPU. With GPU accelerated computing, the GPUs can be employed along with CPUs and it supports parallel programming paradigm with multiple cores. Thus, the research work used GPU acceleration for finding the frequent patterns with a novel way of generating patterns using VDF called GNVDF. It is evident from the experiment that the GNVDF is faster when compared with the GNVDF

without GPU acceleration. i.e. the usage of GPU in GNVDF and the novel pattern formation enhances the speed by 94% with GPU acceleration. Further, the removal of common transactions from frequent 1-itemset saves the memory space considerably than JAB-VDF. It is also evident from the results the GNVDF is the more efficient method than the other proposed methods and existing methods viz., prefixed-itemset based storage, VDF, Matrix-Apriori, NPF-VDF and GNVDF without GPU-acceleration.

7.2 Limitations and Future Research Directions

The proposed algorithms has achieved an improved efficiency in finding frequent patterns in terms of time and memory as discussed in the section 7.1, but all the proposed models were experimented with four real-time and synthetic datasets downloaded from the repositories and not tested with dynamic transactional datasets and also suitable only for the transactional databases.

The following are some of the future research directions that can be done with the proposed models:

i. The research works may be extended by evaluating with dynamic datasets and also experimenting with other types of datasets such as unstructured text, video and audio.
ii. Map-Reduce based parallel processing can be applied with cloud resources and data can be stored in a distributed storage system in order to handle the big data.
iii. Mine several kinds of frequent patterns such as frequent closed itemsets, max-patterns, sequential patterns, and constraint-based frequent patterns.

7.3 Endnote

This research work has formulated a new architectural framework called "SUMsFPM", which incorporates four methods viz., RISOTTO, JAB-VDF, TB-NPF-VDF and GNVDF for minimizing the runtime and memory requirement in finding frequent patterns from transactional databases than the existing algorithms. The ideas projected in this thesis are original, innovative and unique and it is not present elsewhere in the literature and tested its effectiveness using four datasets both real-time and synthetic types derived from the FIMI repository (http://fimi.ua.ac.be) and an open-source Data Mining Library (http://www.philippe-fournier-viger.com/spmf). It has been proved from the experiments that the proposed algorithms enhance the performance more than the state-of-art methods in terms of reduced runtime and memory usage. This work is non-existent earlier in literature and the same is endorsed by a few journals and conferences for its veracity.

REFERENCES

REFERENCES

[ABH,14] Aggarwal, C. C., Bhuiyan, M. A., & Hasan, M. A. (2014). Frequent pattern mining algorithms: A survey. In *Frequent pattern mining*, Springer, Cham.

[AFB[a],14] Albert, D.W., Fayaz, K., & Babu, D.V. (2014). HSApriori: high speed association rule mining using apriori based algorithm for GPU. *Int. J. of Multidisciplinary and Current research*.

[AFB[b],14] Albert, D. W., Fayaz, K., & Babu, D. V. (2014). Exploiting Parallel Processing Power of GPU for High Speed Frequent Pattern Mining. *International Journal of Computer Engineering and Applications*, 7(2), 71 - 81.

[AH, 14] Aggarwal, C. C., & Han, J. (2014). Frequent Pattern Mining. *Springer International Publishing Switzerland*, ISBN: 978-3-319-07820-5, ISBN 978-3-319-07821-2 (eBook).

[AH,15] Agyapong, K. B., & Hayfron-Acquah, J. B. (2015). An Improved Apriori Algorithm Established on Probability Matrix. *International Journal of Scientific & Technology Research*, 4(11), 125-128.

[AHGA[+],18] Aqra, I., Herawan, T., Ghani, N. A., Akhunzada, A., Ali, A., Razali, R.B., & Choo, K. K. R. (2018). A novel association rule mining approach using TID intermediate itemset. *PloS one*, 13(1), 01-32.

[AP,13] Alwa, A. R. H., & Patil, B. A. V. (2013). New Matrix Approach to Improve Apriori Algorithm. *International Journal of Computer Science and Network Solutions*, 1(4), 102-109.

[AR,14] Aguru, S., & Rao, B. M. (2014). A Hash Based Frequent Itemset Mining using Rehashing. *International Journal on Recent and Innovation Trends in Computing and Communication*, 2(12), 4198-4204.

[BDH, 16] Benhamouda, N. C., Drias, H., & Hirèche, C. (2016). Meta-Apriori: A New Algorithm for Frequent Pattern Detection. *In Asian Conference on Intelligent Information and Database Systems*, Springer, Berlin, Heidelberg, 9622, 277-285.

[BGD,15] Bhandari, A., Gupta, A., & Das, D. (2015). Improvised Apriori algorithm using frequent pattern tree for real time applications in data mining. *Procedia Computer Science*, 46, 644-651.

[BML,14] Bhat, M. P. S., Malviya, M. M., & Lade, M. S. (2014). Optimization of MDSRRC with Matrix Apriori. *International Journal of Operations and Logistics Management*, 3(2), 140-147.

[BPG,17] Bhandari, B., Pant, B., & Goudar, R. H. (2017). ARAA: A Fast Advanced Reverse Apriori Algorithm for Mining Association Rules in Web Data. *International Journal of Engineering and Technology (IJET)*, 8(6), 2956-2963.

[CGGK,00] Chou, P. B., Grossman, E., Gunopulos, D., & Kamesam, P. (2000). Identifying prospective customers. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 447-456.

[Cha,14] Chaudhary, V. (2014). Multiple Minimum Support Implementations with Dynamic Matrix Apriori Algorithm for Efficient Mining of Association Rules. *International Journal for Scientific Research and Development*, 2(7), 489-500.

[CHK,18] Chon, K. W., Hwang, S. H., & Kim, M. S. (2018). GMiner: A fast GPU-based frequent itemset mining method for large-scale data. *Information Sciences*, 439, 19-38.

[CJAH$^+$,19] Chee, C. H., Jaafar, J., Aziz, I. A., Hasan, M. H., & Yeoh, W. (2019). Algorithms for frequent itemset mining: a literature review. *Artificial Intelligence Review*, *52*(4), 2603-2621.

[CSS,15] Chaudhary, R., Sharma, S., & Sharma, V. K. (2015). Improving the performance of MS-Apriori algorithm using dynamic matrix technique and map-reduce framework. *Int. J. Innov. Res. Sci. Technol*, 2(5), 2349-6010.

[DD,12] Dhange, N., & Dhande, S. (2012). Matrix based Efficient Apriori Algorithm. *International Journal of Advanced Research in Computer Science*, 3(4), 341-343.

[DDBC,19] Djenouri, Y., Djenouri, D., Belhadi, A., & Cano, A. (2019). Exploiting GPU and cluster parallelism in single scan frequent itemset mining. *Information Sciences*, 496, 363-377.

[DMPW+,10] De Alwis, B., Malinga, S., Pradeeban, K., Weerasiri, D., & Perera, S. (2010). Horizontal format data mining with extended bitmaps. In *International Conference of Soft Computing and Pattern Recognition*, IEEE, 220-223.

[DS,16] Dhak, B. S., & Sawarkar, M. (2016). Apriori: a promising data warehouse tool for finding frequent itemset and to define association rules. *International Journal of Engineering Research and General Science*, 4(1), 60-65.

[DZZC,16] Du, J., Zhang, X., Zhang, H., & Chen, L. (2016). Research and improvement of Apriori algorithm. *In Sixth International Conference on Information Science and Technology (ICIST)*, IEEE, 117-121.

[EZ,03] El-Hajj, M., & Zaïane, O. R. (2003). Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. *In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 109-118.

[FAB,14] Fageeri, S. O., Ahmad, R., & Baharudin, B. B. (2014). An Enhanced Semi-Apriori Algorithm for Mining Association Rules. *Journal of Theoretical and Applied Information Technology*, 63(2), 298-304.

[FD,11] Fakhrahmad, S.M., & Dastghaibyfard, G. (2011). An Efficient Frequent Pattern Mining Method and its Parallelization in Transactional Databases. *Journal of Information Science and Engineering*, 27(2), 511-525.

[FLXH+,09] Fang, W., Lu, M., Xiao, X., He, B., & Luo, Q. (2009). Frequent itemset mining on graphics processors. In *Proceedings of the fifth international workshop on data management on new hardware*, 34-42.

[FPS,96] Fayyad, U., Piatetsky-shapiro, G., & Smyth, P. (1996). From data Science to knowledge discovery in databases. *AI Magazine*, 17(3), 37-54.

[GAF,17] Gawwad, M. A., Ahmed, M. F., & Fayek, M. B. (2017). Frequent itemset mining for big data using greatest common divisor technique. *Data Science Journal*, 16(25), 1-10.

[GLFC+,19] Gan, W., Lin, J. C. W., Fournier-Viger, P., Chao, H. C., Yu, P. S. (2019). A survey of parallel sequential pattern mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*,13(3), 1-34.

[GR,13] Guo, J., & Ren, Y. G. (2013). Research on improved Apriori algorithm based on coding and mapreduce. *In Web Information System and Application Conference (WISA)*, IEEE, 294-299.

[GSG,16] Ganesh, C., Sathyabhama, B., & Geetha, D. T. (2016). Fast frequent pattern mining using vertical data format for knowledge discovery. *International Journal of Engineering Research in Management & Technology*, 5, 141-149.

[GW,10] Guo, Y. M., & Wang, Z. J. (2010). A vertical format algorithm for mining frequent item sets. In *2010 2nd International Conference on Advanced Computer Control*, IEEE, 4, 11-13.

[HD,11] He, Y. S., & Du, P. (2011). Improved Apriori algorithm based on compressing transactional matrix multiplication. *In Key Engineering Materials*, 460, 409-413.

[HD,16] Hamidi, H., & Daraee, A. (2016). Analysis of pre-processing and post-processing methods and using data mining to diagnose heart diseases. *International Journal of Engineering*(*IJE*), 29(7), 921-930.

[HH,16] Hashemzadeh, E., & Hamidi, H. (2016). Using a data mining tool and fp-growth algorithm application for extraction of the rules in two different dataset. *International Journal of Engineering*, 29(6), 788-796.

[HL,15] Huang, C. H., & Leu, Y. (2015). A LINQ-based conditional pattern collection algorithm for parallel frequent itemset mining on a multi-core computer. *In Proceedings of the ASE BigData & SocialInformatics*, 1-6.

[HPK,12] Han, J., Pei, J., & Kamber, M. (2012). *Data mining: concepts and techniques*. 3rd Ed., Elsevier.

[HT,16] Hung, L. N., & Thu, T. N. T. (2016). Mining Frequent Itemsets with Weights over Data Stream Using Inverted Matrix. *International Journal of Information Technology and Computer Science (IJITCS)*, 8(10), 63-71.

[HTDV,19] Huynh, B., Trinh, C., Dang, V., Vo, B. (2019). A parallel method for mining frequent patterns with multiple minimum support thresholds. *International Journal of Innovative Computing. Information and Control*, 15(2), 479-488.

[HYW,08] Hsieh, C. Y., Yang, D. L., & Wu, J. (2008). An efficient sequential pattern mining algorithm based on the 2-sequence matrix. *In 2008 IEEE International Conference on Data Mining Workshops*, 583-591.

[HYZH$^+$,13] Huang, Y. S., Yu, K. M., Zhou, L. W., Hsu, C. H., & Liu, S. H. (2013). Accelerating parallel frequent itemset mining on graphics processors with sorting. *In IFIP International Conference on Network and Parallel Computing, Springer, Berlin, Heidelberg*, 245-256.

[IMA,15] Ibrahim, H. M., Marghny, M., & Abdelaziz, N. M. (2015). Fast Vertical Mining Using Boolean Algebra. *International Journal of Advanced Computer Science and Applications*, 6(1), 89-96.

[IR,16] Ishita, R., & Rathod, A. (2016). Frequent Itemset Mining in Data Mining: A Survey. *International Journal of Computer Applications*, 139(9).

[Jin,10] Jin, H. (2010). A counting mining algorithm of maximum frequent itemset based on matrix. *In IEEE Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, 3, 1418-1422.

[JMG,16] Jen, T. Y., Marinica, C., & Ghariani, A. (2016). Mining frequent itemsets with vertical data layout in MapReduce. *In International Workshop on Information Search, Integration, and Personalization*, Springer, Cham, 66-82.

[JS,15] Jaiswal, R., & Soni, R. (2015). A Novel Apriori Algorithm for Association Rules Mining. *International Journal of Modern Trends in Engineering and Research*, 2(3), 374-378.

[Kal,17] Kalpana, D. (2017). Data Mining Apriori Algorithm Implementation Using R. *International Research Journal of Engineering and Technology*, 4(11), 1810-1815.

[KK,17] Kumar, B., & Kumar, D. (2017). A Matrix based Maximal Frequent Itemset Mining Algorithm without Subset Creation. *International Journal of Computer Applications*, 159(6), 23-26.

[KSG,16] Kaur, J., Singh, R., & Gurm, R.K. (2016). Performance Evaluation of Apriori Algorithm using Association Rule Mining Technique. *International Journal of Technology and Computing*, 2(5), 126-132.

[KSK,12] Kumar, G. V., Sreedevi, M., & Kumar, N. P. (2012). Mining Regular Patterns in Data Streams Using Vertical Format. *International Journal of Computer Science and Security (IJCSS)*, 6(2), 142-149.

[Lan,18] Lang, Z. (2018). The improved Apriori algorithm based on matrix pruning and weight analysis. *In AIP Conference Proceedings*, 1955(1), 040113-1-040113-6.

[LLCL,08] Liu, Y., Liao, W. K., Choudhary, A. N., & Li, J. (2008). Parallel Data Mining Algorithms for Association Rules and Clustering. *In Intl. Conf. on Management of Data*, 1-25.

[LS,16] Lodha, A., & Shrivastava, V. (2016). A Modified Apriori Algorithm for Mining Frequent Pattern and Deriving Association Rules using Greedy and Vectorization Method. *International Journal of Innovative Research in Computer and Communication Engineering*, 4(6), 10722-10726.

[LS,20] Lisnawati, H., & Sinaga, A. (2020). Data Mining with Associated Methods to Predict Consumer Purchasing Patterns. *International Journal of Modern Education and Computer Science(IJMECS)*, 12(5), 16-28.

[LSHW,15] Li, J., Sun, F., Hu, X., & Wei, W. (2015). A multi-GPU implementation of apriori algorithm for mining association rules in medical data. *ICIC Express Letters*, 9(5), 1303-1310.

[LVSM,14] Logeswari, T., Valarmathi, N., Sangeetha, A., & Masilamani, M. (2014) Analysis of Traditional and Enhanced Apriori Algorithms in Association Rule Mining. *International Journal of Computer Applications*, 87(19), 4-8.

[LXYC,17] Li, Y., Xu, J., Yuan, Y. H., & Chen, L. (2017). A new closed frequent itemset mining algorithm based on GPU and improved vertical structure. *Concurrency and Computation: Practice and Experience*, 29(6), e3904.

[MDA,11] Mohamed, M. H., Darwieesh, M. M., & Ali, A. S. (2011). Advanced Matrix Algorithm (AMA): reducing number of scans for association rule generation. *International Journal of Business Intelligence and Data Mining*, 6(2), 202-214.

[MLWY+,00] Ma,Y., Liu,B., Wong,C.K., Yu,P.S., & Lee,S.M. (2000). Targeting the right students using data mining. In *Proceedings of the 6th Int. conference on Knowledge discovery and data mining*, 457-464.

[MR,16] Mohan, V., & Rajpoot, D.S. (2016). Matrix-OverApriori: An Improvement Over Apriori Using Matrix. *International Journal of Computer Science Engineering (IJCSE)*, 5(1), 1-6.

[MSB,12] Mujawar, T. N., Shinde, S. K., & Bhojane, V. (2012). XML Data Mining using XQuery and Improved Apriori Algorithm. *International Journal of Advanced Research in Computer Science*, 3(3), 516-521.

[MYZL,16] Ma, Z., Yang, J., Zhang, T., & Liu, F. (2016). An improved Eclat algorithm for mining association rules based on increased search strategy. *International Journal of Database Theory and Application*, 9(5), 251-266.

[NJGC+,17] Niu, K., Jiao, H., Gao, Z., Chen, C., & Zhang, H. (2017). A developed Apriori algorithm based on frequent matrix. *In Proceedings of the 5th international conference on bioinformatics and computational biology*, 55-58.

[OE,12] Oguz, D., & Ergenc, B. (2012). Incremental itemset mining based on matrix Apriori algorithm. In *International Conference on Data Warehousing and Knowledge Discovery*, Springer, Berlin, Heidelberg, 192-204.

[OKSI,00] Oyama, T., Kitano, K., Satou, K., & Ito, T. (2000). Mining association rules related to protein-protein interactions. *Genome Informatics*, 11, 358-359.

[PD,16] Patil, S. D., & Deshmukh, R. R. (2016). Review and Analysis of Apriori Algorithm for Association Rule. *International Journal of Latest Trends in Engineering and Technology*, 6(4), 104-112.

[PP,15] Prithiviraj, P., & Porkodi, R. (2015). A comparative analysis of association rule mining algorithms in data mining: a study. *American Journal of Computer Science and Engineering Survey*, 3(98), 98-119.

[PVG,06] Pavón, J., Viana, S., & Gómez, S. (2006). Matrix Apriori: Speeding Up the Search for Frequent Patterns. In *Databases and Applications*, 75-82.

[QGYH,14] Qiu, H., Gu, R., Yuan, C., & Huang, Y. (2014): YAFIM: a parallel frequent itemset mining algorithm with spark. Proceedings of IEEE. *International Parallel & Distributed Processing Symposium Workshops*, 1664-1671.

[QL,12] Qin, X., & Liu, Y. (2012). Matrix-based multidimensional sequential pattern mining algorithm and application. In *IEEE International Conference on Computer Science and Information Processing (CSIP)*, 879-882.

[RS_1,16] Rathod, S., & Sharma, A. (2016). Implementation of Enhancement of Apriori Algorithm. *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, 4(5), 402- 408.

[RS_2,16] Ravikiran, D., & Srinivasu, S. V. N. (2016). Regular Pattern Mining on Crime Data Set using Vertical Data Format. *International Journal of Computer Applications*, 143(13).

[SAR, 20] Sahoo, Anasuya, & Rajiv Senapati. (2020). A Boolean Load-Matrix Based Frequent Pattern Mining Algorithm. In *International Conference on Artificial Intelligence and Signal Processing (AISP)*, IEEE,1-5.

[SBE,21] Shawkat, M., Badawi, M., & Eldesouky, A. I. (2021). A Novel Approach of Frequent Itemsets Mining for Coronavirus Disease (COVID-19). *European Journal of Electrical Engineering and Computer Science*, *5*(2), 5-12.

[Sch, 07] Schildt, H. (2007). Java™ : The Complete Reference, 7th Edn., Tata McGraw Hill, , ISBN: 978-0-07-226385-5.

[SD,13] Singh, H., & Dhir, R. (2013). A new efficient matrix based frequent itemset mining algorithm with tags. *International Journal of Future Computer and Communication*, 2(4), 355-358.

[SD,15] Surati Sandip, B., & Desai Apurva, A. (2015). Latest Survey on Frequent Pattern Mining: Mine the Frequent Patterns from Transaction Database. *Vnsgu Journal of Science And Technology*, 4(1), 1-7.

[Sin,16] Singla, V. (2016). A Review: Frequent Pattern Mining Techniques in Static and Stream Data Environment. *Indian Journal of Science and Technology*, 9(45), 1-7.

[SJ,20] Shuwen, L., & Jiyi, X. (2020). An improved apriori algorithm based on matrix. In *2020 12th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, IEEE, 488-491.

[SK,19] Subhashini, A., & Karthikeyan, M. (2019). Itemset Mining using Horizontal and Vertical Data Format. *International Journal for Research in Engineering Application & Management*, 5(3), 534-539.

[SL,20] Sun, R., & Li, Y. (2020). Applying Prefixed-Itemset and Compression Matrix to Optimize the MapReduce-based Apriori Algorithm on Hadoop. In *Proceedings of the 9th International Conference on Software and Computer Applications*, 89-93.

[SM,18] Sumathi, P., & Murugan, S. (2018). A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach. *International Journal of Computer Sciences and Engineering*, 6(11), 152-157.

[SNM,15] Suresh, P., Nithya, K.N., & Murugan, K. (2015). Improved Generation of Frequent Itemsets using Apriori Algorithm. *International Journal of Advanced Research in Computer and Communication Engineering*, 4(10), 25-27.

[SS,20] Sahoo, A., & Senapati, R. (2020). A Boolean load-matrix based frequent pattern mining algorithm. In *2020 International Conference on Artificial Intelligence and Signal Processing (AISP)*, IEEE, 1-5.

[ST,16] Samoliya, M., & Tiwari, A. (2016). On the use of rough set theory for mining periodic frequent patterns. *International Journal of Information Technology and Computer Sciences*, 8(7), 53-60.

[SV,17] Sharmila, S., & Vijayarani, S. (2017). Frequent Itemset Mining and Association Rule Generation using Enhanced Apriori and Enhanced Eclat Algorithms. *International Journal of Innovative Research in Computer and Communication Engineering*, 5(4), 6793-6804.

[TC,16] Thakur, K., & Chopra, V. (2016). To Enhance & Optimize the Apriori Algorithm using Tokenization based Association Rule Mining, *International Journal of Advance Engineering and Research Development*, 3(6), 237-242.

[TG,15] Tanna, P., & Ghodasara, Y. (2015). Analytical Study and Newer Approach towards Frequent Pattern Mining using Boolean Matrix. *IOSR Journal of Computer Engineering*, 17(3), 105-109.

[THY,09] Tsay, Y. J., Hsu, T. J., & Yu, J. R. (2009). FIUT: A new method for mining frequent itemsets. *Information Sciences*, 179(11), 1724-1737.

[TSM,14] Tiwary, M., Sahoo, A. K., & Misra, R. (2014). Efficient implementation of apriori algorithm on HDFS using GPU. In *International Conference on High Performance Computing and Applications (ICHPCA)*, IEEE.1-7.

[VA,15] Vu, L., & Alaghband, G. (2015). A self-adaptive method for frequent pattern mining using a CPU-GPU hybrid model. In *Proceedings of the Symposium on High Performance Computing*, 192-201.

[VD,19] Vijay, K., & Deshpande, B. (2019). Data Science: Concepts and Practice, 2nd Edition, 1-18, doi:0.1016/B978-0-12-814761-0.00001-0.

[VLC+,16] Vo, B., Le, T., Coenen, F, et al. (2016). Mining frequent itemsets using the N-list and subsume concepts, *International Journal of Machine Learning and Cybernetics*, 7(2), 253-265.

[VP,15] Vijayalakshmi, V., & Pethalakshmi, A. (2015). An efficient count based transaction reduction approach for mining frequent patterns. *Procedia Computer Science*, 47, 52-61.

[VV,13] Vijay Kumar, G., & Valli Kumari, V. (2013). Parallel Regular-Frequent Pattern Mining in Large Databases. *International Journal of Scientific & Engineering Research*, 4(6).

[Wan,11] Wang, P. S. (2011). A New Algorithm of Association Rules Mining Based on Relation Matrix. In *Advanced Materials Research*, 179, 55-59.

[WDY,13] Wang, F., Dong, J., & Yuan, B. (2013). Graph-based substructure pattern mining using CUDA dynamic parallelism. In *Int. conference on intelligent data engineering and automated learning*, 342-349.

[Wet,02] Wetjen, T. (2002). Discovery of frequent gene patterns in microbial genomes. *TZI-Report, Technologie Zentrum Informatik (TZI)*, 27.

[WS,11] Wang, B. L., & Shen, Y. G. (2011). Improvement of Apriori algorithm based on Boolean matrix. In *Advanced Materials Research*, 159, 144-148.

[WXXS,18] Wang, Y., Xu, T., Xue, S., & Shen, Y. (2018). D2P-Apriori: A deep parallel frequent itemset mining algorithm with dynamic queue. In *10th International Conference on Advanced Computational Intelligence (ICACI)*, 649-654, IEEE.

[XJW,19] Xuan, Q., Jiuyuan, H., & Weitao, W. (2019). Research on Improvement of Parallel Apriori Algorithm Based on Boolean Matrix and Weight. In *12th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, IEEE, 96-99.

[YE,10] Yıldız, B., & Ergenç, B. (2010). Comparison of two association rule mining algorithms without candidate generation. In *the 10th IASTED international conference on Artificial Intelligence and Applications*, 450-457.

[YH,05] Yuan, Y., & Huang, T. (2005). A matrix algorithm for mining association rules. In *International Conference on Intelligent Computing*, Springer, Berlin, Heidelberg, 3644, 370-379.

[YWWJ,11] Yu, H., Wen, J., Wang, H., & Jun, L. (2011). An improved Apriori algorithm based on the Boolean matrix and Hadoop. *Procedia Engineering*, 15, 1827-1831.

[YXHJ+,13] Yongchun, J., Xiaona, L., Hairong, C., Jiao, X., & Yingchun, W. (2013). Improved mining frequent itemsets algorithm based on sim. *Information Technology Journal*, 12(11), 2246-2250.

[YZ,16] Yu, S., & Zhou, Y. (2016). A Prefixed-Itemset-Based Improvement for Apriori Algorithm. *arXiv preprint arXiv:1601.01746.*

[ZG,03] Zaki, M. J., & Gouda, K. (2003). Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 326-335.

[ZLZ,08] Zhang, Z., Liu, J., & Zhang, J. (2008). A Fast Algorithm for Mining Association Rules Based on Boolean Matrix. In *IEEE 4th International Conference on Wireless Communications, Networking and Mobile Computing*, 1-3.

[ZOKL+,19] Zhou, D., Ouyang, M., Kuang, Z., Li, Z., Zhou, J. P., & Cheng, X. (2019). Incremental association rule mining based on matrix compression for edge computing. *IEEE Access*, 7, 173044-173053.

[ZWH,04] Zhang, Z., Wu, W., & Huang, Y. (2004). Mining dynamic interdimension association rules for local-scale weather prediction. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, IEEE*, 2, 146-149.

[ZWX,10] Zhen-yu, L., Wei-xiang, X., & Xumin, L. (2010). Efficiently using matrix in mining maximum frequent itemset. In *IEEE 3rd International Conference on Knowledge Discovery and Data Mining*, 50-54.

[ZY,12] Zong-Yu, Z., & Ya-Ping, Z. (2012). A parallel algorithm of frequent itemsets mining based on bit matrix. In *IEEE International Conference on Industrial Control and Electronics Engineering*, 1210-1213.

[ZYW,10] Zhou, J., Yu, K. M., & Wu, B. C. (2010). Parallel frequent patterns mining algorithm on GPU. In *IEEE International Conference on Systems, Man and Cybernetics*, IEEE, 435-440.

[ZZ,17] Zheng, J., & Zhang, J. (2017). Improvement of Apriori algorithm based on matrix compression. In *7th International Conference on Education, Management, Information and Mechanical Engineering*, 76, 131-135.

Web References

[1] http://fimi.ua.ac.be

[2] http://www.philippe-fournier-viger.com/spmf

[3] https://link.springer.com/chapter/10.1007/978-3-319-07821-2_2

[4] https://www.sciencedirect.com/science/article/pii/B978012381479100006X

[5] https://www.sciencedirect.com/topics/computer-science/frequent-patterns

[6] https://www.sciencedirect.com/topics/computer-science/knowledge-discovery-in-database

[7] https://en.wikipedia.org/wiki/Welch%27s_t-test

APPENDICES

Appendix -A

i) Google Scholar Image Showing the Research Scholar Publications

ii) Papers Included in International Digital Libraries

Appendix - B

i) Papers Published in the International Journals

sumathi parasuraman - Google S

scholar.google.com/citations?user=SR_28jAAAAAJ&hl=en&authuser=2

sumathi parasuraman

FOLLOW

Assistant Professor
No verified email
Database and Data Mining

TITLE	CITED BY	YEAR
GNVDF: A GPU-accelerated Novel Algorithm for Finding Frequent Patterns Using Vertical Data Format Approach and Jagged Array P Sumathi, S Murugan International Journal of Modern Education & Computer Science 13 (4)	1	2021
A Multithread, Novel Pattern Based Algorithm for Finding Frequent Patterns With Jagged Array and Vertical Data Format P Sumathi, S Murugan, V Umadevi Indian Journal of Computer Science and Engineering 12 (5), 1353-1363		2021
A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach P Sumathi, S Murugan International Journal of Computer Sciences and Engineering 6 (11), 152-157		2018
RISOTTO-A Novel Hybrid Approach for Enhancing Classical Apriori Algorithm P Sumathi, S Murugan		

Articles 1–4 SHOW MORE

Cited by

	All	Since 2017
Citations	1	1
h-index	1	1
i10-index	0	0

Co-authors EDIT

No co-authors

This author profile is generated by Scopus Learn more

Sumathi, P.

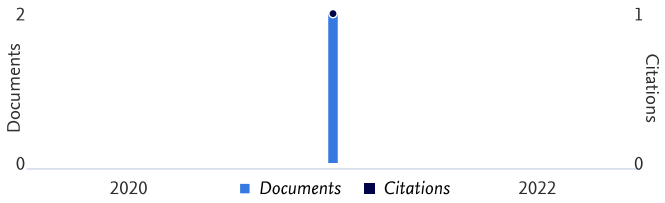
Nehru Memorial College, Puthanampatti, Puthanampatti, India
Connect to ORCID

Edit profile Set alert Save to list Potential author matches
Export to SciVal

Metrics overview

2 Documents by author
1 Citations by **1 documents**
1 *h*-index:

Document & citation trends



Most contributed Topics 2016–2020

This author has no topics at the moment. To learn why, or more about topics in general. Learn more about Topics

View all Topics

2 Documents Cited by 1 Documents Beta 0 Preprints 2 Co-Authors Topics
0 Awarded grants

Note:
Scopus Preview users can only view an author's last 10 documents, while most other features are disabled. Do you have access through your institution? Check your institution's access to view all documents and features.

Export all Save all to list Sort by Date (newest)

View list in search results format
View references
Set document alert

Article • *Open access*

A multithread, novel pattern based algorithm for finding frequent patterns with jagged array and vertical data format

0 Citations

Sumathi, P., Murugan, S., Umadevi, V.

Indian Journal of Computer Science and Engineering, 2021, 12(5), pp. 1353–1363

Show abstract Related documents

Article • *Open access*

Gnvdf: A gpu-accelerated novel algorithm for finding frequent patterns using vertical data format approach and jagged array

1 Citations

Sumathi, P., Murugan, S.

ISSN 2319 – 1953
International Journal of Scientific Research in Computer Science Applications and Management Studies

RISOTTO - A Novel Hybrid Approach for Enhancing Classical Apriori Algorithm

P.Sumathi[#1], S. Murugan[*2]

[#1]*Research Scholar,* [*2]*Associate Professor*
[#1],[*2]*Department of Computer Science, Nehru Memorial College (Autonomous), Puthanampatti, Tiruchirappalli -Dt, TamilNadu, India*
[#1]sumiparasu@gmail.com, [*2]smurugan_nmc@hotmail.com

Abstract **- Discovering frequent itemsets is the computationally intensive step in the task of mining association rules and Apriori is one of the most significant algorithms for finding the frequent itemsets. The main challenge in the classical Apriori is that, the mining often needs to generate a huge number of candidate itemsets and requires more number of database scans which increases time and decreases efficiency. It also increase the I/O cost and requires more memory. To eradicate these issues a lot of improvements to Apriori have been proposed in the literature. In this series, this research work also introduces a refinement to the Apriori which uses a data structure called prefixed-itemset and the horizontal data format approach. Based on the comparative analysis with the classical Apriori, the proposed approach truncates the number of database scans and reduces the time required for finding candidate generation.**

Keywords **- Apriori, Association Rule Mining, Candidate itemsets, Frequent itemsets, Horizontal data format, Prefixed-itemset.**

I. INTRODUCTION

Association Rule Mining (ARM) is a process for finding relations between data items in datasets. ARM has been a successful technique for extracting knowledge from databases [15]. Frequent patterns are the patterns (a set of items, subsequences, subgraphs, etc.) that occur frequently in a data set. Frequent pattern mining is an essential data mining task in the field of data mining and mining frequent patterns from large scale databases has emerged as an important research problem in data mining and knowledge discovery community. Association rules are the main technique used to determine the frequent item set in data mining. Apriori algorithm is the first algorithm proposed by R.Agrawal and R.Srikant in 1994 in the field of data mining and it is a classical algorithm of ARM. It generates frequent item sets for boolean association rule. As the Apriori algorithm uses the prior knowledge of frequent item set properties it is named as Apriori. Apriori employs an iterative approach known as level-wise search, where kth item set is used to explore (k+1)th-item sets. There are two steps involved in each iteration and is repeated when no candidate set can be generated.

They are

1. Generation of candidate item sets
2. Finding the occurrence of each candidate item set in database and pruning all disqualified candidate set based on support count (threshold) first and on closure property. Ie., if a set of items is frequent, then all of its proper subsets are also frequent

After finding the frequent item sets, the association rules are generated from those large item sets with the constraints of minimal confidence (min_conf) and minimum support (min_sup) thresholds. But this classical algorithm is inefficient because

1. It is not suitable for large databases
2. It defines the presence and absence of an item
3. It allows uniform minimum support threshold
4. More scanning of transaction database is needed for generating frequent item sets
5. More I/O cost is required
6. Generation of candidate item-sets and support counting are expensive

Thus, to eradicate the said disadvantages, there are many efficient pattern mining algorithms have been discovered in the last two decades and some of the recent articles in the literature are shown in section 2, but still research is going on in creating efficient frequent pattern mining algorithm and ARM. In this succession, a novel hybrid approach for enhancing classical Apriori has been introduced in this paper.

The remaining paper is organized as follows. Section 2 describes the review of literature. The proposed approach of this paper is presented in section 3. An illustrative example for the proposed methodology is presented in section 4. Section 5 discusses the results. Finally section 6 ends with conclusion.

II. REVIEW OF LITERATURE

Association Rule Mining (ARM) is a successful technique for finding relations between data items in databases. The most widely used Apriori algorithm for generating association rule discovers frequent patterns by generating candidate item sets which is a costly and memory consuming one. Research in improving the Apriori is a common issue and is an ongoing research topic these days. This section presents a brief overview of the recent literature related to enhancing classical Apriori algorithm and it provides a stronger lead to the proposed work.

In [1], the authors have introduced a Modified Apriori algorithm using greedy and vectorization method. They compared the execution time of traditional Apriori and Modified Apriori by varying the number of transactions and proved that the Modified Apriori requires less time than the

Apriori. They also proved that the proposed method reduces the number of rules generated than the original Apriori. The authors in [2] have developed a new recursive algorithm based on Apriori called Meta-Apriori. In that, they partitioned the whole database into smaller ones using divide and conquer approach. After partitioned them, they applied Meta-Apriori if the partition is huge or Apriori if it is of reasonable size. Finally, they merged the achieved results to get the result for whole database and proved that Meta-Apriori requires less time than the Apriori.

In [3], the authors have proposed a modified Apriori called DC_Apriori. In this, the authors have restructured the storage structure of the database and they generated *k*-frequent item sets by joining the 1-frequent item sets with k-1-frequent item sets. They avoided the unnecessary invalid candidate sets and also reduced the number of database scanning and also improved the efficiency of frequent item sets generation. A modified Apriori have been proposed in [4] using Transposition technique and proved that it is less complex than the classical Apriori.

A method called Advanced Reverse Apriori Algorithm (ARAA) has been proposed in [5], which is opposite to Apriori. In that the authors have generated the k^{th} itemset first and move on to the lower level sets i.e., k-1,k-2,…,1. They compared Apriori Algorithm (AA), Reverse Apriori Algorithm (RAA) and ARAA and proved that the number of scans in ARAA is less than the AA but greater than RAA and is equal to number of transactions in the database. Also, proved that the ARAA is more suitable for all type of datasets but RAA is applicable for higher datasets. An enhanced Apriori algorithm and enhanced Eclat algorithm with different threshold value for each item have been proposed in [6]. The authors compared them with different size of dataset and with different size of items and proved that the enhanced Apriori is best than the enhanced Eclat in terms of the number of frequent items and rules.

A modified Apriori algorithm called FMA (Frequent Matrix Apriori) has been proposed by Kun Nin et al [7]. In that, they scanned the dataset only once to store frequent item set information in the frequent matrix, then discretizing the matrix by minimum support parameter in the frequent matrix and finally, the most frequent item sets are found recursively by scanning the discretized dataset. It was proved by them that the FMA is more effective than the AA in terms of time. An improved Apriori has been designed in [8]. In this method, the transaction ID's along with the support count is maintained in the frequent item sets and they generated the k+1 itemset by set intersection and proved that the number of database scans is reduced than the classical Apriori algorithm.

A prefixed-itemset based data structure for candidate itemset generation has been proposed in [9]. In that, the candidate itemsets are stored with smaller storage space and performed the connecting and the pruning step of the Apriori algorithm much faster. It was analyzed that the proposed structure improved the efficiency of the classical Apriori algorithm. A new algorithm called enhanced Apriori algorithms has been introduced in [10], which takes less scanning time and reduces the I/O spending time by cutting down the unwanted transaction records in the database.

A new algorithm called semi-Apriori using a binary based data structure for mining frequent itemsets as well as association rule has been proposed in [11] and proved that this technique outperforms Apriori in terms of execution time. In [12], an improved Apriori algorithm has been presented and made a comparison between conventional Apriori and Improved Apriori algorithm. It was proved that the improved Apriori provides better performance than classical Apriori algorithm. A novel Apriori algorithm has been proposed in [13] to overcome the limitations of the classical Apriori algorithm based on local and global power set and observed that the novel algorithm requires only two scans instead of many scans in classical Apriori algorithm. in [14], The authors surveyed the good improved approaches of Apriori from 2012 to 2015.

From the literature it has been found that the Apriori algorithm has been alleviated to several levels, which pawed way for enhancing the classical Apriori. In succession, the RISOTTO algorithm has been proposed in this paper for enhancing the conventional Apriori.

III. Proposed Methodology

The proposed methodology combines both prefixed-itemset based storage concept [9] and horizontal data format approach [8] for enhancing the conventional Apriori algorithm in terms of time and database scans. The algorithm progresses as follows:

In the first step, the proposed algorithm finds the frequent 1-itemset from the transaction database by scanning it once as in classical Apriori. But, it also maintains the transaction ID's in which the frequent 1-items occurs along with the support count (SC) or TNR (Total Number of Transactions) as in horizontal data format approach which forms the candidate itemset C_1. L_1 is constructed from C_1 by removing the items whose SC is less than the minimum support count (min_sup). The transaction ID's are only maintained in C_1 and L_1. Also, the frequent 1-itemset is stored in a new data structure (DS) called prefixed-itemset based storage which contains a prefix-key and values. The prefix for frequent 1-itemset is always NULL and the values are the items in L_1. In general, the frequent *k*-itemset where $k = 1,2,3, \ldots, n$ contains $(k-1)$-items as prefix-key (LK_k) and the last item content as the value (LV_k).

In the second step, the values in frequent *1*- itemset in the prefixed-itemset based storage LV_1 is joined by itself ($LV_1 \bowtie LV_1$) instead $L_1 \bowtie L_1$ and the items which do not satisfy the Apriori property is removed and then they are combined with the prefix key which forms C_2. The Apriori property i.e., all nonempty subsets of a frequent itemset must also be frequent is considered to improve the efficiency by reducing the search space. The support count for the items in C_2 is calculated just by performing intersection of the transaction ID's in L_1 instead of scanning the database as in classical Apriori, which minimizes the database scans. From C_2, L_2 is formed by

removing those elements from C_2 whose support count is less than the min_sup. Similar to the previous step, the frequent *2*-itemsets are appended to the prefixed-itemset based storage with the appropriate prefix and the values. The second step is repeated with *k = 3,4,5, …* until there is no more candidate itemsets found. The proposed approach is named as RISOTTO abbreviated from the phrase "pRefixed ItemSet hOrizonTal daTa fOrmat". The steps involved in RISOTTO are shown in the algorithm 1.1.

Algorithm 1.1: RISOTTO. Finding frequent itemsets

Input:

- D, a database of transactions.
- min_sup, the minimum support count threshold.

Output:

- L, the frequent itemsets in D.

Method:

(1) L←Ø
(2) C_1←scan D and generate candidate 1-itemsets
(3) L_1 ← generate frequent 1-itemsets using min_sup
(4) L←L ∪ L_1
(5) PIDS←create a prefixed-itemset DS
(6) PIDS(LK_1)←NULL
(7) PIDS(LV_1)←items in L_1
(8) for (k=2; L_{k-1}≠Ø; k++) do
(9) C_{k_init}←PIDS(LV_{k-1})⋈ PIDS(LV_{k-1})
(10) Prune candidate *k*-items in C_{k_init}
(11) C_k←join PIDS(LK_{k-1}) ⋈ C_{k_init}
(12) L_k← generate frequent *k*-itemsets using min_sup
(13) PIDS(LK_k)←(*k*-1)-items in L_k
(14) PIDS(LV_k)←k^{th} item in L_k
(15) L←L ∪ L_k
(16) endfor
(17) return L

The main advantage of this hybrid approach is that, it reduces the number of database scans because it finds the SC for frequent k-itemsets where k=2,3,4,… by set intersection method from the transaction ID's in L1 which in turn minimizes the I/O cost. Using the prefixed- itemset storage, the number of candidate itemsets produced is reduced than the classical Apriori algorithm

A. Proposed methodology: An Example

To illustrate the proposed methodology, a sample transaction database D shown in Table 1 has been considered which consists of 9 transactions. Each transaction comprises of TID (Transaction ID) and items bought from the items available in the business enterprise namely A, B, C, D and E respectively. Let the min_sup=2. The frequent 1-itemset is computed as in the classical Apriori but the L1 in the proposed method contains TID's and TNR or SC. The computation of C1 and L1 is shown in Table 2.

TABLE I
TRANSACTION DATABASE D

Transaction ID (TID)	Items bought
T1	A,B,E
T2	B,D
T3	B,C
T4	A,B,D
T5	A,C
T6	B,C
T7	A,C
T8	A,B,C
T9	A,B,C,E

TABLE II
COMPUTATION OF C_1 AND L_1

C_1

Item	Transaction ID's	TNT or SC
{A}	T1, T4, T5, T7, T8, T9	6
{B}	T1,T2, T3,T4, T6, T8, T9	7
{C}	T3, T5, T6, T7, T8, T9	6
{D}	T2, T4	2
{E}	T1,T9	2

L_1

Item	Transaction ID's	TNT or SC
{A}	T1, T4, T5, T7, T8, T9	6
{B}	T1,T2, T3,T4, T6, T8, T9	7
{C}	T3, T5, T6, T7, T8, T9	6
{D}	T2, T4	2
{E}	T1,T9	2

In this case, for computing C1 and L1 one database scan is performed. Also prefixed-itemset storage is used for keeping the frequent k-itemset which contains 3 columns. Columns 1, 2 and 3 indicate the type of frequent itemset, prefix-key and values in frequent k-itemsets. For frequent 1-itemset, the itemsets contain 1-itemset, the prefix-key is NULL and values are {A,B,C,D,E} which is shown in Table 3.

TABLE III
PREFIXED-ITEMSET STORAGE WITH FREQUENT 1-ITEMSET

Itemsets	Prefix – Key	Values
1 - itemset	NULL	{A,B,C,D,E}

Now {A,B,C,D,E} ⋈ {A,B,C,D,E} is performed which is {AB,AC,AD,AE,BC,BD,BE,CD, CE,DE}. All items in {AB,AC,AD,AE,BC,BD,BE,CD,CE,DE} supports Apriori property and the set of items are the items in C2. The SC of {AB}=count({T1,T4,T5,T7,T8,T9}∩{T1,T2,T3,T4,T6,T8,T9 })=count({T1,T4,T8,T9})=4.SC of {AC}=count({T1 , T4, T5, T7,T8,T9}∩{T3,T5,T6,T7,T8,T9})=count({ T5, T7, T8, T9}) = 4. Similarly, the SC for other items in C2 is computed and it is shown in Table 4. Out of these items in C2, only the items AB, AC, AE, BC, BD and BE satisfies the min_sup and which forms L2. The frequent 2-itemsets are appended to prefixed itemset storage. In L2, the items AB, AC and AE has the

common prefix A and values are {B,C,E}. Similarly, the items BC, BD and BE has the common prefix B and {C,D,E} are the values and it is shown in Table 5.

From Table 5, {B, C, E} ⋈ {B, C, E}={BC,BE,CE}and the item CE not satisfies the Apriori property, therefore {B, C, E} ⋈ {B, C, E}={BC,BE} and each item is prefixed with the prefix-key A gives {ABC,ABE}. Similarly {C, D, E} ⋈ {C, D, E} = {CD,CE,DE} and the items in {CD,CE,DE} does not satisfies the Apriori property. There {C, D, E} ⋈ {C, D, E}=Ø. Now the frequent 3-itemset contains only two items {ABC,ABE}. The SC for ABC = count({T1, T4, T5, T7, T8, T9}∩{T1, T2, T3, T4, T6, T8, T9}∩{T3, T5, T6, T7, T8, T9}) = count ({T8,T9})=2.

TABLE IV
COMPUTATION OF FREQUENT 2-ITEMSET

C_2

Itemset	SC (By set intersection)
{AB}	4
{AC}	4
{AD}	1
{AE}	2
{BC}	4
{BD}	2
{BE}	2
{CD}	0
{CE}	1
{DE}	0

L_2

Itemset	SC
{AB}	4
{AC}	4
{AE}	2
{BC}	4
{BD}	2
{BE}	2

TABLE V
PREFIXED-ITEMSET STORAGE WITH 1-ITEMSET AND 2-ITEMSET

Itemsets	Prefix – Key	Values
1-itemset	NULL	{A,B,C,D,E}
2-itemset	A	{B, C, E}
	B	{C, D, E}

TABLE VI
COMPUTATION OF 3-FREQUENT ITEMSET

L_3

Itemset	SC(By set intersection)
{ABC}	2
{ABE}	2

C_3

Itemset	SC (By set intersection)
{ABC}	2
{ABE}	2

Similarly, SC for ABE = count ({T1,T4,T5,T7,T8,T9 }∩{T1,T2,T3,T4,T6,T8, T9 }∩{ T1,T9})=count({T1,T9})=2. Both counts satisfies the min_sup, therefore L3, frequent 3-itemsets contain {ABC},{ABE} which is shown in Table 6. The same is appended in the prefixed itemset storage with {AB} as prefix and {C,E} as values and it is shown shown in Table 7.

TABLE VII
PREFIXED-ITEMSET STORAGE WITH 1-ITEMSET,2-ITEMSET AND 3-ITEMSET

Itemsets	Prefix – Key	Values
1-itemset	NULL	{A,B,C,D,E}
2-itemset	A	{B, C, E}
	B	{C, D, E}
3-itemset	AB	{C,E}

Now {C,E}⋈{C,E}={CE} and the item CE does not satisfies the Apriori property. Therefore frequent 4-candidate item set C4 is NULL and the algorithm terminates.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

An extensive experiment for RISOTTO and Classical Apriori is made using the transaction database D shown in Table 1. The candidate itemsets and frequent itemsets generated using the classical Apriori and RISOTTO algorithms are shown in Table 8 and 9.

TABLE VIII
CANDIDATE ITEMSETS AND FREQUENT ITEMSETS OF TABLE 1 USING CLASSICAL APRIORI

Candidate itemsets using Classical Apriori (After Join)		Candidate itemsets using Classical Apriori (After Prune)		Frequent itemsets using Classical Apriori	
C_1	{A,B,C,D,E}	C_1	{A,B,C,D,E}	L_1	{A,B,C,D,E}
C_2	{AB,AC,AD, AE,BC,BD, BE,CD,DE}	C_2	{AB,AC,AE.B C,BD,BE}	L_2	{AB,AC,AE. BC,BD,BE}
C_3	{ABC,ABE, ACE, BCD, BCE,BDE}	C_3	{ABC,ABE}	L_3	{ABC,ABE}
C_4	{ABCE}	C_4	Ø	L_4	Ø

TABLE IX
CANDIDATE ITEMSETS AND FREQUENT ITEMSETS OF TABLE 1 USING RISOTTO

Candidate itemsets using RISOTTO (After Join)		Candidate itemsets using RISOTTO (After Prune)		RISOTTO	
C_1	{A,B,C,D,E}	C_1	{A,B,C,D,E}	L_1	{A,B,C,D,E}
C_2	{AB,AC,AD, AE,BC,BD,B E,CD,DE}	C_2	{AB,AC,AE. BC,BD,BE}	L_2	{AB,AC,AE.B C,BD,BE}
C_3	{ABC,ABE}	C_3	{ABC,ABE}	L_3	{ABC,ABE}
C_4	Ø				

The RISOTTO outperforms well than the classical Apriori and the comparison results were shown in Table 10. From table 10, it is observed that the RISOTTO algorithm minimizes the database scan to 1 and requires less time for generating the candidate itemset and frequent itemsets. This is because the proposed algorithm maintains the transaction in which frequent 1-itemset occurs and also the prefixed-itemset

storage DS enhances the time to generate the candidate itemsets.

TABLE X
COMPARISON OF CLASSICAL APRIORI VS. RISOTTO

Parameters	Classical Apriori	RISOTTO
Number of database scans	3	1
Number of candidate itemsets	More than RISOTTO	Less than Apriori
Time Required for finding candidate generation	More than RISSOTTO	Less than Apriori

V. CONCLUSION

The research work has introduced an enhanced Apriori algorithm called RISOTTO, a new hybrid approach for generating frequent itemsets which combines both horizontal data format approach and prefixed-itemset based storage DS. In the proposed method, frequent 1-itemset stores the transactions in which the frequent 1-itemset occurs which reduces the number of database scans required to find the frequent itemsets and also reduces the I/O cost. The joining and pruning steps are performed using the values in the prefixed-itemset DS rather than the values in frequent itemsets as in classical Apriori which reduced the time required to generate the candidate itemsets. Thus the RISOTTO, method enhances the existing Apriori algorithm.

REFERENCES

[1]. Lodha, A., & Shrivastava, V. (2016, June). A Modified Apriori Algorithm for Mining Frequent Pattern and Deriving Association Rules using Greedy and Vectorization Method", *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 4, issue 6, pp. 10722-10726.

[2]. Benhamouda, N. C., Drias, H., & Hirèche, C. (2016, March). Meta-Apriori: A New Algorithm for Frequent Pattern Detection. *In Asian Conference on Intelligent Information and Database Systems*, Springer, Berlin, Heidelberg, vol.9622, pp. 277-285.

[3]. Du, J., Zhang, X., Zhang, H., & Chen, L. (2016, May). Research and improvement of Apriori algorithm. *In Sixth International Conference on Information Science and Technology (ICIST)*, IEEE, pp. 117-121.

[4]. Kaur, J., Singh, R., and Gurm, R.K. (2016, May). Performance Evaluation Of Apriori Algorithm Using Association Rule Mining Technique, *International Journal of Technology and Computing*, vol. 2, issue 5, pp.126-132.

[5]. Bhandari, B., Pant, B., and Goudar, R. H. (Dec 2016-Jan 2017). ARAA: A Fast Advanced Reverse Apriori Algorithm for Mining Association Rules in Web Data, *International Journal of Engineering and Technology (IJET)*,vol. 8, issue 6, pp.2956-2963.

[6]. Sharmila, S., and Vijayarani, S. (2017). Frequent Itemset Mining and Association Rule Generation using Enhanced Apriori and Enhanced Eclat Algorithms, *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 5, issue 4, pp. 6793-6804.

[7]. Niu, K., Jiao, H., Gao, Z., Chen, C., and Zhang, H. (2017, January). A developed apriori algorithm based on frequent matrix. *In Proceedings of the 5th international conference on bioinformatics and computational biology*, ACM, pp. 55-58.

[8]. Suresh,P., Nithya, K.N., & Murugan, K. (2015, October). Improved Generation of Frequent Item Sets using Apriori Algorithm, *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 4, issue 10, pp. 25-27.

[9]. Yu, S., & Zhou, Y. (2016). A Prefixed-Itemset-Based Improvement For Apriori Algorithm. *arXiv preprint arXiv:1601.01746*.

[10]. Logeswari, T., Valarmathi, N., Sangeetha, A., and Masilamani, M. (2014) Analysis of Traditional and Enhanced Apriori Algorithms in Association Rule Mining, *International Journal of Computer Applications*,vol. 87, issue 19,pp.4-8.

[11]. Fageeri,S.O., Ahmad, R., Baharudin,B.B. (2014). An Enhanced Semi-Apriori Algorithm For Mining Association Rules, *Journal of Theoretical and Applied Information Technology*, vol. 63, issue 2, pp. 298-304.

[12]. Rathod, S., and Sharma, A. (2016, May). Implementation of Enhancement of Apriori Algorithm, *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, vol. 4, issue 5, pp.402- 408.

[13]. Jaiswal, R., and Soni, R. (2015, March). A Novel Apriori Algorithm for Association Rules Mining, *International Journal of Modern Trends in Engineering and Research*, vol. 02, issue 03, pp.374-378.

[14]. Patil, S.D., and Deshmukh, R.R. (2016, March). Review and Analysis of Apriori Algorithm for Association Rule, *International Journal of Latest Trends in Engineering and Technology*, voal.6, issue 4, pp. 104-112.

[15]. Yuan, Y. and Huang, T. (2005, August). "A Matrix Algorithm for Mining Association Rules", Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, vol. 3644, pp. 370–379.

IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.6, Special Issue.11, Dec 2018 E-ISSN: 2347-2693

A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach

P. Sumathi[1*], Dr. S. Murugan[2]

[1]Department of Computer Science, Nehru Memorial College (Autonomous), Tiruchirappalli, India
[2]Department of Computer Science, Nehru Memorial College (Autonomous), Tiruchirappalli, India

[*]*Corresponding Author: sumiparasu@gmail.com*

Available online at: www.ijcseonline.org

***Abstract*—** Data mining is the process of extracting the concealed information and rules from large databases. But the real world datasets are sparse, dirt and also contain hundreds of items. Frequent Pattern Mining (FPM) is one of the most intensive problems in discovering frequent itemsets from such datasets. Apriori is one of the premier and classical data mining algorithms for finding frequent patterns but it is not an optimized one. So over last two decades a remarkable variations and improvements were made to overcome the inefficiencies of Apriori algorithm such as FPGrowth, TreeProjection, Charm, LCM, Eclat and Direct Hashing and Pruning (DHP), RARM, ASPMS etc., In any case, a little enhancement in the algorithm improves the mining process considerably. Frequent itemset mining with vertical data format approach has been proposed as an improvement over the basic Apriori, which reduces the number of database scans and also uses array storage structure. This research paper has proposed a space efficient implementation of finding frequent itemsets with vertical data format using jagged array. It reduces the usage of memory by allocating exact memory. An experiment is done between the array implementation of vertical data format approach and jagged array implementation. From the experiment it is proved that the proposed jagged array implementation method utilizes the memory efficiently when compared with the traditional multidimensional array.

***Index Terms* —** Apriori, Array, Eclat, Frequent Pattern Mining, FPGrowth, Jagged Array, and Vertical Data Format.

I. INTRODUCTION

Now-a-days, volumes of data are exploding both in scientific and commercial domains. Data mining techniques are used to extract unknown information from the huge amount of data and became popular in many applications. Association Rule Mining (ARM) is one of an important core data mining techniques to discover patterns/rules among items in a large database of variable-length transactions. Its goal is to identify the groups of items that most often occurs together i.e., it focuses on finding frequent itemsets each occurring at more than a minimum support frequency (min_sup) among all transactions. It is widely used in market basket transaction data analysis, graph mining applications like substructure discovery in chemical compounds, pattern finding in web browsing, word occurrence analysis in text documents, and so on [1].

The major risks associated with finding frequent itemsets are i) computational time and ii) memory needed for the task because even with a moderate sized dataset, the search space and memory utilization of FPM is enormous, which is exponential to the length of the transactions in the dataset. Therefore, it is essential to perform FPM analysis in a space-and-time efficient way. Many researchers in this area focused on reducing computational time to find frequent patterns and this work focuses on reducing the memory utilization using jagged array storage structure in the vertical data mining algorithms.

Rest of the paper is organized as follows. Section 2 describes the review of literature. The proposed implementation method of Vertical Data Format (VDF) is illustrated in section 3. The comparison of existing and the proposed implementation methods are discussed in section 4 and finally section 5 ends with conclusion.

II. REVIEW OF LITERATURE

Improving the computational time and memory is always an issue in ARM and this section briefs the research contributions made by different researchers in this line which pawed way for the proposed implementation.

In [2], the authors have presented a VDSRP method to generate complete set of regular patterns over a data stream at a user given regularity threshold using sliding-window and VDF. It has been proved that the proposed method outperforms both in execution and memory consumption.

Ravikiran, D., et. al, have proposed a new model called RCP to mine regular sort of crimes in crime database using VDF which requires only one database scan. From the experimental results they proved that RCP is more efficient than the existing RPtree[3]. In [4], the authors have focused

on the various FPM techniques, their challenges in static and stream data environment.

The authors in [6] have presented a new algorithm, which mine frequent itemsets with vertical format. They proved that the new algorithm needs a single database scan and finds new frequent item sets through 'and operation' between item sets. The new algorithm requires less storage space, and improves the efficiency of data mining.

An enhanced Apriori and Eclat has been introduced in [8], in which individual thresholds for each itemset has been used and proved that that the enhanced-Apriori algorithm outperforms Enhanced-Eclat Algorithm.

In [9], the authors have presented an improved version of Eclat called Eclat-growth algorithm based on increased search strategy. For reducing the runtime in generating an intersection of two itemsets and support degree calculation, a BSRI (Boolean array Setting and Retrieval by Indexes of transactions) method has been introduced. It has been proved by them that the Eclat-growth outperforms Eclat, Eclat-diffsets, Eclat-opt and hEclat in mining association rules.

In [10], a VFFM algorithm has been developed which represents the transaction database in vertical format in the form of binary, where the attribute presence and absence is represented by 1 and 0 respectively. After one scan of transaction database for transformation it generates candidate sets and subsets similar to Apriori algorithm. The support value of each candidate itemsets is counted by intersection of every pair of frequent single items instead of database scan and proved that the VFFM outperforms Apriori.

Compressed bit vectors of frequent itemsets based on Boolean algebra named Vertical Boolean Mining (VBM) has been presented in [11] and it performs the intersection of two compressed bit vectors without making any costly decompression operation. They proved from the experiments that the VBM is better than Apriori and the classical vertical association rule mining algorithms in terms of mining time and memory usage.

A novel VDF representation called Diffset has been developed by the authors in [12], which keep track of the differences in the tid's of a candidate pattern and from which it generates frequent patterns. The method cut down the size of memory required to store intermediate results and also increased performance significantly.

From the existing literatures, it is noted that no authors have proposed a jagged array implementation of VDF approach for enhancing the memory requirement of VDF. Thus, this work implements VDF using the jagged array for efficient utilization of memory.

III. JAGGED ARRAY IMPLEMENTATION OF VERTICAL DATA FORMAT APPROACH

Frequent patterns are itemsets [set of items, such as milk and bread, that appear frequently together in a transaction data set], subsequences [buying first a PC, then a digital camera, and then a memory card, if it occurs frequently in a shopping history database], or substructures [subgraphs, subtrees or sublattices] that appear in a dataset with frequency no less than a user-specified threshold (min_sup)[7]. Finding frequent patterns plays an essential role in mining associations, correlations and many other interesting relationships among data. ARM is one of the data mining techniques to discover the hidden patterns/rules among items in a large database of variable-length transactions that help in making decision and predictions [4].

Apriori Algorithm, FP-Growth and Eclat [4] are the popularly available static data mining techniques for finding frequent patterns. Apriori is the basic algorithm for mining frequent patterns which suffers from space complexity due to large number of candidate generation and also requires multiple scans of database. FP-growth uses a tree structure for mining frequent itemsets. Due to limited number of database scans and zero candidates, it is efficient as compared to Apriori. Both the Apriori and FP-growth algorithms mine frequent patterns in Horizontal Data Format (HDF) (i.e., {TID: itemset}), where TID is a transaction-id and itemset is the set of items in TID and it is shown in Table I.

TABLE I. TRANSACTION DATABASE D IN HDF

TID	List of item IDS
T1	A,B,E
T2	B,D
T3	B,C
T4	A,B,D
T5	A,C
T6	B,C
T7	A,C
T8	A,B,C
T9	A,B,C,E

But the data can also be presented in {item: TID-set} format where item is an item name and TID-set is the set of transactions containing the item called VDF. The VDF is used in Eclat algorithm that minimizes the database scan and uses set intersection of Tid's for finding the support count for *k*-itemsets where *k*=2,3,...,n. The VDF of the transaction database D is shown in Table II. The comparisons between

the Apriori, FP-Growth and Eclat with different parameters are shown in Table III. From Table III and in [4] it is observed that the FP mining algorithms which use VDF are very fast and requires less memory space when compared with HDF approaches. But, the VDF approaches use array storage structure for storing the database in memory.

TABLE II. VDF OF D

itemset	TID_set
A	T1,T4,T5,T7,T8,T9
B	T1, T2, T3, T4, T6, T8,T9
C	T3, T5, T6, T7, T8,T9
D	T2,T4
E	T1,T9

To reduce memory space further, this research work implements the VDF using jagged array. It is a special case of 2-D array and it is an array of array in which the length of each array can differ. This concept is available in JAVA, VB.NET and C#.NET. This implementation helps to reduce the memory needed considerably because in the real life grocery datasets the customers will not purchase all the items in the shop. Thus, this implementation utilizes the memory effectively.

A. An Example

The first part of this section shows the memory requirement for the array implementation of VDM. Let the grocery shop sells *n* (5) items viz., A, B, C, D and E and consider the transaction database *D* shown in Table I. It contains *t* (9) transactions and it is scanned first to generate VDF. The VDF of Table I is shown in Table II.

TABLE III. COMPARISON BETWEEN STATIC DATA MINING TECHNIQUES FOR FINDING FREQUENT PATTERNS [5]

Comparison Parameters	Apriori	FP-Growth	ECLAT
Technique	Breadth first search and Apriori property (for pruning)	Divide and conquer	Depth first search & intersection of T-id's
Database Scan	scanned for each time a candidate item set is generated	Two times	Few times
Drawback(s)	1. Requires large memory space. 2. Too many candidate item set.	FP-tree is expensive to build and consumes more memory	It requires the virtual memory to perform the transaction.
Advantage(s)	1. Easy to implement. 2. Use large item set property	Database is scanned two times	1. No need to scan the database each time 2. fast
Data format	Horizontal	Horizontal	Vertical
Storage structure	Array	Tree (FP-tree)	Array
Time	More execution time	Execution time is less than Apriori	Execution time is less than Apriori

The support count (SC) for each item is the number of transaction-id's that it contains i.e. the SC of A, SC_A=count(A)=6. Similarly, SC_B=7, SC_C=6, SC_D=2 and SC_E=2. Let the min_sup be 2. The frequent 1-itemset contains {A, B, C, D, E}. The VDF is actually stored in the memory as 2-D array, where number of rows (*r*) = items in the grocery shop and number of columns(*c*) = *t*. Here *r*=5 and *c*=9. The memory required for storing 1-itemset in VDF format is

$$TM_1 = (r \times c \times sizeof(tid)) + (sizeof(item_{11}) \times r) \quad (1)$$

Where $item_{11}$ is the first item in the frequent 1-itemset, tid is the transaction-id and sizeof is a built-in function which says the number of bytes required for the argument.

Here each tid requires 2 bytes and $item_{11}$ requires 1 byte of memory respectively. All items say A, B, C, D and E sold in the grocery shop are frequent 1-itemsets. Therefore the VDF requires (5×9×2)+(5×1) = 95 bytes of memory i.e., $TM_1 = 95$ bytes. Suppose if there are some in-frequent items in 1-itemsets, they can be removed which saves memory considerably. The number of bytes of memory removed from 1-itemset is computed as

$$rbytes_1 = (rr_1 \times c \times sizeof(tid)) + (rr_1 \times sizeof(item_{11})) \quad (2)$$

Where, rr_1 is the number of rows to be removed as in-frequent. Therefore the total bytes of memory for frequent 1-itemset is

$$M_1 = TM_1 - rbytes_1 \quad (3)$$

Here M_1 = 95 - 0 = 95 bytes. Similarly, in iteration 2, the possible 2-itemsets combinations are generated from frequent 1-itemsets and it is {AB, AC, AD, AE, BC, BD, BE, CD, CE, DE}. Suppose if there are n items in 1-itemset, the possible two item combinations are $(n \times n\text{-}1)/2$ say tc_2. Among them, the numbers of itemset combinations say x may be in-frequent which need not be placed in VDF. Therefore, the memory required for frequent 2-itemset shown in Table IV is

$$TM_2 = ((tc_2 - x) \times c \times sizeof(tid)) + (sizeof(item_{21}) \times (tc_2 - x)) \quad (4)$$

Where, $item_{21}$ is the first item in the frequent 2-itemset. In this example, the combinations viz., AD,CD,CE and DE are in-frequent and based on equation (4), the VDF requires ((10 - 4) × 9 × 2) + (2 × (10 - 4)) = 108 + 12 = 120 bytes. Similarly from Table IV, the 3-itemset combinations are {ABC, ABD, ABE, ACE, BCD, BCE, BDE} and the combinations ABD, ACE, BCD, BCE and BDE are in-frequent, therefore the frequent 3-itemset requires ((7-5)×9×2)+(7-5)×3)=42 bytes of memory and the VDF of 3-frequent itemsets is shown in Table V. The process is repeated until no frequent itemsets are found.

TABLE IV. VDF OF 2-ITEMSETS

Itemset	TID_set
AB	T1,T4,T8,T9
AC	T5,T7,T8,T9
AE	T1,T9
BC	T3,T6,T8,T9
BD	T2,T4
BE	T1,T9

Therefore, the total memory required for VDF using 2-D array is

$$TM = M_1 + \sum_{i=2}^{itemset_i \neq \varnothing} TM_i \quad (5)$$

Where M_1 is calculated using (3) and TM_i are calculated using the equation (6) shown below.

$$TM_i = ((tc_i - x) \times c \times sizeof(tid)) + (sizeof(item_{i1}) \times (tc_i - x)) \quad (6)$$

Where, tc_i and x *are* the number of items and in-frequent items in the candidate i-frequent itemset. For the above example TM = 95+120+42 =257 bytes of memory. If the same is implemented using jagged array, the memory requirement is reduced considerably. The format of jagged array representation for candidate 1-itemset is shown in Table VI and all items in it are frequent which forms frequent 1-itemset.

TABLE V. VDF OF 3-ITEMSETS

itemset	TID_set
ABC	T8,T9
ABE	T1,T9

TABLE VI. JAGGED ARRAY REPRESENTATION OF 1-ITEMSET

itemset	TID_set						
A	T1	T4	T5	T7	T8	T9	
B	T1	T2	T3	T4	T6	T8	T9
C	T3	T5	T6	T7	T8	T9	
D	T2	T4					
E	T1	T9					

The memory required for candidate 1-itemset TM_1 is calculated as

$$TM_1 = \sum_{\forall item \in \{itemset_1\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad (7)$$

As in two-D representation, there may be x in-frequent items in candidate 1-itemset say {in-frequent} = {$item_1$, $item_2$, …,$item_x$} then the memory for {in-frequent} be saved by removing it and the amount of memory removed is computed as shown in equation (8).

$$rbytes_1 = \sum_{\forall item \in \{in-frequent\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad (8)$$

Therefore the total memory required for frequent 1-itemset in jagged representation is computed using (3) with the values computed using (7) and (8) respectively. Similarly, the jagged array representation of frequent 2-itemset shown in Table VII requires TM_2 - $rbytes_2$ memory space where TM_2 and $rbytes_2$ are calculated by using (9) and (10) respectively.

$$TM_2 = \sum_{\forall item \in \{itemset_2\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad (9)$$

$$rbytes_2 = \sum_{\forall item \in \{in-frequent\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad (10)$$

The jagged representation of frequent 3-itemset is shown in Table VIII which requires TM_3 - $rbytes_3$ memory. This process continues until no more frequent itemsets are

found. For this case the candidate 4-itemset is null and the algorithm terminates. Therefore, the total memory required for the jagged implementation is calculated using equation (11).

$$TM = \sum_{i=1}^{itemset_i \neq \varnothing} TM_i - rbytes_i \quad (11)$$

Where, TM_i and $rbytes_i$ are calculated using (12) and (13) respectively.

$$TM_i = \sum_{\forall item \in \{itemset_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad (12)$$

$$rbytes_i = \sum_{\forall item \in \{in-frequent_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \quad (13)$$

TABLE VII. JAGGED ARRAY REPRESENTATION OF 2-ITEMSET

itemset	TID_set			
AB	T1	T4	T8	T9
AC	T5	T7	T8	T9
AE	T1	T9		
BC	T3	T6	T8	T9
BD	T2	T4		
BE	T1	T9		

TABLE VIII. JAGGED ARRAY REPRESENTATION OF 3-ITEMSET

itemset	TID_set	
ABC	T8	T9
ABE	T1	T9

For this example, the jagged representation requires

TM_1 = (6×2 +1)+(7×2+1)+(6×2+1)+(2×2+1)+(6×2+1)
= 13+15+13+5+5=51 bytes

$rbytes_1$ = 0

Therefore M_1=51- 0 = bytes

TM_2 = (4×2+2) +(4×2+2)+(1×2+2)+ (2×2+2) +(4×2+2) +(2×2+2) +(2×2+2) +(0×2+2) +(1×2+2)+ (0×2+2) =10+10+4+6+10+6+6+2+4+2=60 bytes

$rbytes_2$ = (1×2+2)+(0×2+2)+(1×2+2)+(0×2+2)=12 bytes

Therefore M_2 requires = 60 - 12 = 48 bytes of memory.

Similarly, M_3 requires 14 bytes and therefore, the jagged representation for this example requires

$TM=M_1+M_2+M_3$ =51+48+14=113 bytes of memory which is less than 50% in the original array representation.

IV. RESULTS AND DISCUSSION

From the example discussed in section 3.1, the jagged implementation has several advantages. They are

1. No memory space is wasted as in 2-D array because jagged array allocates space only to the transactions in which the items occurs.
2. Minimizes the memory space required than the original array implementation because for the above example the array implementation requires 257 bytes of memory, where as it is 113 bytes when using jagged implementation i.e., it requires less than 50% of memory when compared with the array representation.

Thus, it is finalized that the jagged implementation saves memory significantly and also fast when compared with the horizontal data format approaches.

V. CONCLUSION

From the literatures, it is observed that there is always a trade-off between the computational time and memory in generating frequent itemsets. It is also found that the vertical data format approaches reduces the database scans and finds the support counts by intersection. Though it is best, the array storage structure implementation used by VDF requires more memory because it takes the assumption that each item may fall almost in all transactions. But in real world grocery datasets, each transaction will not contain all items and each item may not present in all transactions. So to reduce the memory consumption, this research work used the jagged array representation for efficient usage of memory and from the experiments it is proved that the proposed implementation approach reduces more than 50% of memory when compared with original 2-D array implementation. In future, this work can be extended to the test real world grocery datasets of more dimensions.

REFERENCES

[1]. Liu, Y., Liao, W. K., Choudhary, A. N., & Li, J. (2008). Parallel Data Mining Algorithms for Association Rules and Clustering, In Intl. Conf. on Management of Data, pp.1-25.

[2]. Kumar, G. V., Sreedevi, M., & Kumar, N. P. (2012). Mining Regular Patterns in Data Streams Using Vertical Format. *International Journal of Computer Science and Security (IJCSS)*, *6*(2), pp.142-149.

[3]. Ravikiran, D., & Srinivasu, S. V. N. (2016). Regular Pattern Mining on Crime Data Set using Vertical Data Format. *International Journal of Computer Applications*, *143*(13).

[4]. Singla, V. (2016). A Review: Frequent Pattern Mining Techniques in Static and Stream Data Environment. *Indian Journal of Science and Technology*, *9*(45), pp.1-7.

[5]. Ishita, R., & Rathod, A. (2016). Frequent Itemset Mining in Data Mining: A Survey. *International Journal of Computer Applications*, *139*(9).

[6]. Guo, Y. M., & Wang, Z. J. (2010, March). A vertical format algorithm for mining frequent item sets. In *Advanced Computer Control (ICACC), 2010 2nd International Conference on* (Vol. 4, pp. 11-13). IEEE.
[7]. Han, J., Kamber, M. Data Mining Concepts and Techniques, *Morgan Kaufmann Publishers*, 2006.
[8]. S.Sharmila, Dr. S.Vijayarani. (2017). Frequent Itemset Mining and Association Rule Generation using Enhanced Apriori and Enhanced Eclat Algorithms, International Journal of Innovative Research in Computer and Communication Engineering, 5(4), pp. 679- 6804.
[9]. Zhiyong Ma, Juncheng Yang, Taixia Zhang and Fan Liu. (2016). An Improved Eclat Algorithm for Mining Association Rules Based on Increased Search Strategy, *International Journal of Database Theory and Application*, 9(5), pp.251-266.
[10]. C.Ganesh, B.Sathiyabhama and T.Geetha. (2016). Fast Frequent Pattern Mining Using Vertical Data Format for Knowledge Discovery, International Journal of Emerging Research in Management &Technology, 5(5), pp.141-149.
[11]. Hosny M. Ibrahim, M.H. Marghny and Noha M.A. Abdelaziz. (2015). Fast Vertical Mining Using Boolean Algebra, *International Journal of Advanced Computer Science and Applications*, 6(1), pp.89-96.
[12]. Mohammed J. Zaki amd Karam Gouda. (2003), Fast Vertical Mining Using Diffsets SIGKDD '03, ACM.

Authors Profile

P.Sumathi received her B.Sc and M.Sc degrees in Computer Science from Seethalakshmi Ramaswami College, affiliated to Bharathidasan University, Tiruchirappalli, India in 2001 and 2003 respectively. She received her M.Phil degree in Computer Science in 2008 from Bharathidasan University. She is presently working as an Assistant Professor in the Department of Computer Science, Vysya College, Salem, India. She is currently pursuing Ph.D., degree in Computer Science in Bharathidasan University. Her research interests include Data structures, Database and Data Mining techniques.

S.Murugan received his M.Sc degree in Applied Mathematics from Anna University in 1984 and M.Phil degree in Computer Science from Regional Engineering College, Trichirappalli in 1994. He is an Associate Professor in the department of Computer Science, Nehru Memorial College (Autonomous), affiliated to Bharathidasan University since 1986. He has 32 years of teaching experience in the field of Computer Science. He has completed his Ph.D., degree in Computer Science with the specialization in Data Mining from Bharathiyar University in 2015. His research interest includes Data and Web Mining. He has published many research articles in the National and International journals.

A MULTITHREAD, NOVEL PATTERN BASED ALGORITHM FOR FINDING FREQUENT PATTERNS WITH JAGGED ARRAY AND VERTICAL DATA FORMAT

P.Sumathi
Research Scholar, PG & Research Department of Computer Science, Nehru Memorial College (Autonomous) (Affiliated to Bharathidasan University), Puthanampatti-621 007, Tiruchirappalli-Dt, Tamil Nadu, India
sumiparasu@gmail.com

Dr.S.Murugan
Associate Professor, PG & Research Department of Computer Science, Nehru Memorial College (Autonomous) (Affiliated to Bharathidasan University), Puthanampatti-621 007, Tiruchirappalli-Dt, Tamil Nadu, India
murugan_nmc@hotmail.com

Dr.V.Umadevi
Assistant Professor, PG & Research Department of Computer Science, Nehru Memorial College (Autonomous) (Affiliated to Bharathidasan University), Puthanampatti-621 007, Tiruchirappalli-Dt, Tamil Nadu, India
yazh1999@gmail.com

Abstract

Frequent pattern mining is essential for discovering hidden items from a database with more than a prescribed threshold. Knowing frequent patterns helps us to determine the relationship between the items. Many researchers narrated novel algorithms for sequential frequent itemset mining using a single thread, but still, there is a need for time, memory efficient and scalable one. Therefore, the research study proposed an approach for finding frequent patterns, namely TB-NPF-VDF (Thread Based, Novel Pattern Formations with Vertical Data Format), which uses a new way of generating candidate items to minimize the time. Also, it employs a multithread concept and runs several threads simultaneously, one for each frequent 1-itemset to generate the remaining frequent itemsets for that item. Further, it also employs a jagged array to store the frequent patterns to reduce the memory requirement. The research work has been implemented and tested using four real-time datasets. Further, it has been compared with Matrix-Apriori, VDF and NPF-VDF (without multithread), and the experimental results reveal that TB-NPF-VDF outperforms significantly in terms of runtime and storage.

***Keywords*: Frequent Patterns; Jagged Array; Multithread; Novel Pattern Formation; Vertical Data Format.**

1. Introduction

Data Mining (DM) is the fastest growing field [1], whose primary goal is to discover or extract information or patterns from large datasets. It is a multidisciplinary field comprising Computer Science and Statistics. It is an analysis step of Knowledge Discovery from Databases (KDD) [2]. Several DM techniques are available, such as Association Rule Mining (ARM), sequential pattern analysis, classification, and clustering. ARM is one of the most widely used techniques for knowledge discovery in the mining domain [3]. ARM is used in several applications such as inventory control, mobile mining, educational mining, market basket analysis, risk management, telecommunication networks and graph mining, etc. [4]. Frequent patterns are the patterns that occur frequently in a dataset whose frequency is more than that of a threshold value specified by the user. For instance, a set of items viz., pen and paper appears frequently together in a transactional dataset is a frequent itemset [1]. Mining frequent patterns is an essential sub-task of ARM [5]. It generates qualitative knowledge, which helps the decision-makers for making valuable business insights [2].

Apriori is a classical algorithm for finding frequent patterns which uses a horizontal format approach proposed by Agrawal and Srikant in 1993 [6] for Boolean association rules. The algorithm begins with generating a 1-itemset, recursively produces a frequent 2-itemset, frequent 3-itemset, and so on until all frequent itemsets are produced [4]. The main drawback of the algorithm is that it generates numerous candidate itemset,

especially for huge frequent 1-itemset and needs to scan the database many times. Many algorithms have evolved over the years to overcome these drawbacks viz., FP-growth, Direct Hashing and Pruning (DHP), Matrix-Apriori and maximal association rule mining, so on. In this line, this research work also introduces a paradigm for finding frequent patterns with a multithreaded approach.

The remaining article is organized as follows. The relevant work related to the proposed work is illustrated in Section 2. Section 3 elaborates the proposed methodology with an analogy. Section 4 discusses the results and section 5 summarizes the conclusion.

2. Related Work

The problem of mining frequent patterns is an essential task in ARM. Several studies have been carried out in this domain to improve the time to generate frequent itemsets and reduce the memory space over the years. This section presents a brief overview of them, providing a strong impetus to the proposed method.

Y. M. Guo et al. [1] have initiated a VDF algorithm for mining frequent itemsets. The new algorithm only needs a single scan of the entire database and uses AND operation for finding the frequent itemsets. Additionally, it proved that the algorithm requires less storage and also improves the mining efficiency. Subashini et al. [4] have studied ARM methods in horizontal and vertical data format approaches viz., Apriori, APRIORITID, APRIORI_RARE and APRIORIRARE_TID. They analyzed the pros and cons of each technique.

Judith Pavón et al. [7] have introduced a method called Matrix-Apriori to increase the speed of finding frequent itemsets. It first generates a Boolean matrix MFI which holds the frequent 1-itemset by traversing the transaction database. The vector STE stores the support count of the candidate itemset for each row in MFI. To accelerate the search of frequent patterns, the first row of MFI writes the indexes. It used a conditional pattern generation method for generating frequent patterns and proved that it performs better than Apriori and FP-Growth algorithms. Sumathi, P and Murugan, S [8] have designed a memory-efficient VDF approach using a jagged array and developed a memory usage model. They demonstrated that memory usage was reduced significantly when compared with multidimensional arrays.

A fast GPU-based frequent itemset mining algorithm for massive datasets called GMiner has been introduced in [9]. It has been developed to overcome the limitations of various parallelism methods viz., multi-core CPU, multiple machines and many-core GPU, particularly the workload skewness. It extracts the patterns from the enumeration tree and uses the computational power of GPU. From the experimentation, they showed that the GMiner is better than the existing ones. Authors in [10] have suggested a novel algorithm, namely Accelerating Parallel Frequent Itemset Mining on Graphics Processors with Sorting (APFMS). This parallel frequent itemset mining utilizes GPU's to accelerate the mining process. GPUs speed-up process using the OpenCL platform and proved that the APFMS outperforms the previous computation time-based methods.

A new multi-core based parallel mining algorithm for finding frequent itemsets has been presented in [11] using LINQ queries. It divides the transactional database into sub-datasets known as conditional patterns. Many threads ran concurrently on a multi-core computing system, one for each conditional pattern. They proved that the algorithm is faster by 2*x* and 4*x* times than the fast Eclat and FP-growth algorithms, respectively. A compressed bit matrix-based parallel algorithm for exploring frequent itemsets has been introduced by Zong-Yu et al., which uses both bottom-up and top-down approaches for efficient pruning [12]. It also uses OpenMP's parallel multithreaded, dynamic scheduling approach to extract frequent itemsets. Finally, they demonstrated that this approach reduces memory space, I/O overhead with a single database scan compared to the Apriori algorithm.

In [13], the authors have proposed a VDF approach for finding frequent itemsets using a Boolean matrix (FPMBM), where the presence of an item for the TID's is 1 and 0 for absence. It uses logical AND operation for finding support count from frequent 2-itemset to frequent *n*-itemsets until it is not empty. To control the number of iterations for candidate generation, it also uses additional information in the Boolean matrix, namely "number of iterations". Further, they demonstrated from the experiment that the FPMBM is efficient and more scalable than the existing ones.

Jen, T. Y., et al. have created a novel vertical format based parallel method for finding frequent patterns called Apriori_V with MapReduce platform. They proved that it provides a significant improvement in reducing the number of operations and decreasing computational complexity [14]. The authors in [15] have introduced a Parallel Regular Frequent Pattern (PRF) method to find out the regular-frequent patterns from large databases using VDF format and proved from the experiments that the algorithm reduced the number of database scans, I/O cost and inter-process communication.

In [16], the authors have reviewed the works related to Parallel Sequential Pattern Mining (PSPM), viz., partition-based, Apriori-based, pattern growth-based, and hybridized algorithms for PSPM. They also reviewed the open-source software's utilized in PSPM. Further, they summarized the issues and uses of PSPM in big data. In [17], the authors have proposed an FPM algorithm with a multi-core processor and Multiple Minimum Support called MMS-FPM. It quickly generated frequent patterns. It has been designed mainly to solve rare item problems. They have proved that the MMS-FPM is more superior to MSApriori and also scalable one. In [18], the authors have designed a Spark-based parallel Apriori algorithm called YAFIM (Yet Another Frequent Itemset Mining). The experimental result revealed that the proposed method is faster than the Apriori's MapReduce implementation by 18 times.

The existing literature found that no authors proposed parallel algorithms using a multithreaded approach with uni-processor systems. Thus, the research work focuses on a multithreaded approach with jagged array representation for VDF and novel pattern formation in finding frequent patterns, namely TB-NPF-VDF. It also compares the proposed work with the methods viz., Matrix-Apriori, VDF and NPF-VDF.

3. Proposed Methodology

The proposed work's main idea is to find frequent patterns for the transaction database TD. It contains four phases. Phase one scans TD first and converts it into VDF, in which a set of TIDs represents each item as in Eclat [19]. The second phase determines the frequent 1-itemset from VDF. The third phase sorts the frequent 1-itemset in ascending order based on the min_sup(δ) threshold, and it is stored in a matrix using the jagged array format. The δ of an itemset X is calculated by dividing the total transactions in which X occurs by the total number of transactions [20]. The fourth phase creates n-1 threads, one for each frequent 1-itemset except for the last one; where n represents the total items in frequent 1-itemset (L_1). Let $L_1=\{I_1, I_2,\ldots, I_n\}$, each thread generates frequent itemsets starting from frequent 2-itemset to frequent k-itemset until it is non-empty, where $k \geq 2$.

For finding frequent i-itemset, $i \geq 2$, each thread ($t_{x,1\leq x\leq n-1}$) uses the following procedure.

(1) When i=2, the thread forms the candidate patterns by combining I_x with I_{x+1} and finds the transactions in which the combination I_xI_{x+1} occur by intersecting the transactions in I_x and I_{x+1}. The item combinations whose support count $\geq \delta$ is selected as frequent i-itemset for item x.

(2) For i>2, each item in frequent (i-1)-itemset is combined with each frequent 1-itemset starting from the next item in the last item of frequent(i-1)-itemset and the transactions in which the combination exists is determined by intersecting the item in frequent (i-1)-itemset and the appropriate item in frequent 1-itemset. This procedure will be repeatedly performed as far as the frequent k-itemset is not null.

The proposed method uses multithreads and novel pattern formation with VDF to find frequent patterns is named TB-NPF-VDF. The main benefit of this method is that it generates fewer candidate itemsets than the classical Apriori and VDF because it avoids the items whose support count is lesser than the item at any instance of time for generating the patterns. As threads are used, the CPU is effectively utilized, and it is faster compared to processes. This method avoids checking the pattern for the Apriori property because the candidate patterns generated satisfies the Apriori property by default. Further, the time required for TB-NPF-VDF is less when compared to VDF. The memory requirement is minimized since the algorithm uses the matrix notation using a jagged array [8].

The algorithm for the proposed method is shown below, and the workflow of TB-NPF-VDF is illustrated in Fig.1.

TB-NPF-VDF: Algorithm to discover the frequent patterns

Input: TD - Transactional database;
δ - min_sup threshold;

Output: Frequent itemsets;

1: $vdf \leftarrow$ scan TD and store it in <$itemset$, TID_{list}> format;

2: $C_1 \leftarrow \emptyset$;

3: **for** each $item_i$ in vdf do

4: $SC \leftarrow$ count($TID_{list}(item_i)$); //determines the number of transactions in $item_i$

5: $C_1 \leftarrow C_1$.append ({$itemset$, TID_{list}, SC})// adds a row into C_1

6: **endfor**

7: **for** each $item_i$ in C_1 do

8: $L_1 \leftarrow \{item_i \mid SC(item_i) \geq \delta\}$

endfor
L_1←jagged(sort(L_1)) //sorts L_1 and converts it into a jagged matrix format
no_freq1_itemset←count(L_1) //determines the number of itemset in L_1
for (x=1; $x \leq$ (*no_freq1_itemset-1*); x++)
    t_x←create(thread) //create t_x for the $L_1[x]$
endfor
for each thread t_x **do**
    for (k=2; $L_k \neq \emptyset$; k++)
        if k==2 **then**
            new_pattern←<$I_x I_{x+1}$>;
            new_TID_list←Transactions(I_x)∩Transactions(I_{x+1});
        else if $k \geq 2$ **then**
            for each $item_j$ in L_{k-1} **do**
                new_item←last item in $item_j$
                new_pattern←{<$item_j I_y$>|I_y←next(*new_item*)}
                new_TID_list←Transactions($item_j$)∩Transactions(I_y);
            endfor
        endif
        SC←count(*new_TID_list*);
        C_k←C_k.append({*new_pattern*,*new_TID_list*});
        L_k←{C_k | $SC(C_k) \geq \delta$}
    endfor
endfor

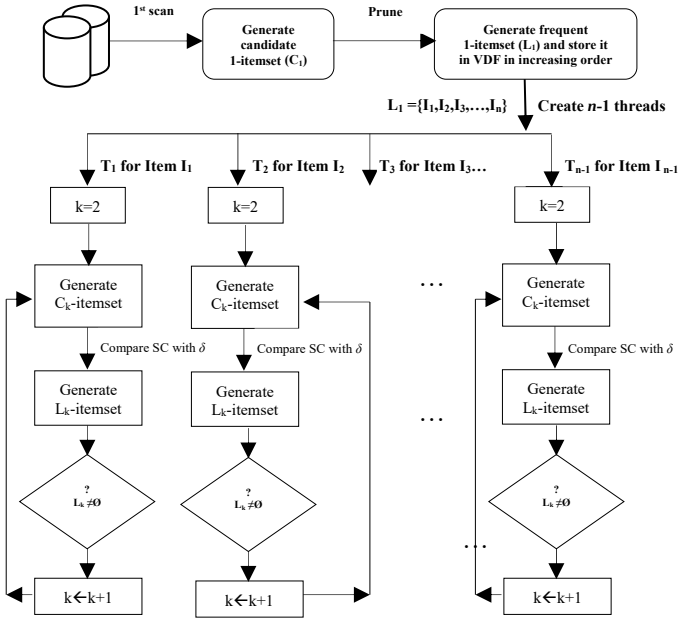


Fig. 1. Workflow of TP-NPF-VDF

3.1. *Example*

To understand the relevance of the proposed work, the Transactional Database (TD) shown in Table 1 has been considered. It consists of 12 items, namely A,B,C,D,E,F,G,H,I,K,M and P. The vertical representation of TD is shown in Table 2. Each row represents an item consisting of the item name and the TID's in which the item belongs. Assume the min_sup (δ) as 6. The candidate 1-itemset (C_1) consists of all the items in TD, the transaction IDs in which the items occurred, and the support count (*SC*), i.e. the total transactions in which the item appears. The C_1 for TD is shown in Table 3. Among them, the items viz., A,C,D,E,F,I,M and P satisfy the δ and form the frequent 1-itemset(L_1). The jagged array representation of the same is shown in Table 4 [21].

TID	Items Purchased
0	D,C,G,E,I,H,P,K,M
1	E,B,G,F,I,H,M,P
2	E,C,M
3	B,A,D,C,F,E,I,G,P
4	B,A,D,C,P,E
5	B,A,D,C,H,F,P
6	E,B,H,F,P,I,M
7	C,A,E,D,P,K,M
8	C,A,E,D,I,F,M,P
9	C,A,E,D,H,F,P,I,M

Table 1. Transactional Database (TD)

Item	Transaction ID's (TID's)
A	{3, 4, 5, 7, 8, 9}
B	{1, 3, 4, 5, 6}
C	{0, 2, 3, 4, 5, 7, 8, 9}
D	{0, 3, 4, 5, 7, 8, 9}
E	{0, 1, 2, 3, 4, 6, 7, 8, 9}
F	{1, 3, 5, 6, 8, 9}
G	{0, 1, 3}
H	{0, 1, 5, 6, 9}
I	{0, 1, 3, 6, 8, 9}
K	{0, 7}
M	{0, 1, 2, 6, 7, 8, 9}
P	{0, 1, 3, 4, 5, 6, 7, 8, 9}

Table 2. Transactional Database in VDF

C_1

Itemset	TID's	SC
A	{3, 4, 5, 7, 8, 9}	6
B	{1, 3, 4, 5, 6}	5
C	{0, 2, 3, 4, 5, 7, 8, 9}	8
D	{0, 3, 4, 5, 7, 8, 9}	7
E	{0, 1, 2, 3, 4, 6, 7, 8, 9}	9
F	{1, 3, 5, 6, 8, 9}	6
G	{0, 1, 3}	3
H	{0, 1, 5, 6, 9}	5
I	{0, 1, 3, 6, 8, 9}	6
K	{0, 7}	2
M	{0, 1, 2, 6, 7, 8, 9}	7
P	{0, 1, 3, 4, 5, 6, 7, 8, 9}	9

Table 3. Candidate 1-Itemset

L_1

1-Itemset	TID's								
A	3	4	5	7	8	9			
C	0	2	3	4	5	7	8	9	
D	0	3	4	5	7	8	9		
E	0	1	2	3	4	6	7	8	9
F	1	3	5	6	8	9			
I	0	1	3	6	8	9			
M	0	1	2	6	7	8	9		
P	0	1	3	4	5	6	7	8	9

Table 4. Jagged Array Representation of L_1

To generate fewer candidate itemsets, this research work uses a novel pattern generation method rather than the natural join used in the Apriori algorithm. For that, the L_1 is sorted in ascending order based on *SC* and replaced with L_1 as illustrated in Table 5.

The sorted L_1 contains 8 items, and this work creates 7 threads because the frequent 1-itemset contains 8 items. Thread-1 is for the item <A>, Thread-2 is for item <B>, etc. The Thread-1 first generates the following patterns.

<AF>, <AI>, <AD>, <AM>, <AC>, <AE> and <AP> and for each pattern, set intersection is calculated by using the TID's in each item of the pattern. For example, for the pattern <AF> the set intersection is calculated as {3, 4, 5, 7, 8, 9} ∩ {1, 3, 5, 6, 8, 9} = {3, 5, 8, 9} and *SC*=4.

L_1

1- Itemset	TID's								
A	3	4	5	7	8	9			
F	1	3	5	6	8	9			
I	0	1	3	6	8	9			
D	0	3	4	5	7	8	9		
M	0	1	2	6	7	8	9		
C	0	2	3	4	5	7	8	9	
E	0	1	2	3	4	6	7	8	9
P	0	1	3	4	5	6	7	8	9

Table 5. Sorted L_1

Similarly, the *SC* for other patterns viz., <AI>, <AD>, <AM>, <AC>, <AE> and <AP> is calculated as stated above. The patterns whose $SC \geq \delta$ will be considered as the frequent 2-itemset for the item <A> and are represented in Table 6. For this case, the patterns <AD>, <AC> and <AP> satisfies the δ.

Item	TID's					
<AD>	3	4	5	7	8	9
<AC>	3	4	5	7	8	9
<AP>	3	4	5	7	8	9

Table 6. Frequent 2-Itemset for <A> by Thread-1

Next, the method generates the candidate 3-itemsets for each frequent 2-itemset in Table 6 as follows.

(1) For the frequent 2-item <AD>, the items viz., <M>, <C>, <E> and <P> are considered from frequent 1-itemset because <M> is the next item after <D> where, <D> is the last item in frequent 2-itemset <AD>. The patterns generated are <ADM>, <ADC>, <ADE> and <ADP> and for them, the transactions in which the pattern occurs and *SC* is calculated as follows.
From Table 6, the TID's of <AD> is {3, 4, 5, 7, 8, 9} and from Table 5 the TID's of <M> is {0, 1, 2, 6, 7, 8, 9}. Therefore, {3, 4, 5, 7, 8, 9} ∩ {0, 1, 2, 6, 7, 8, 9} = {7, 8,T} and *SC*=3. Similarly, for <ADC>, <ADE> and <ADP> is also calculated.

(2) For the frequent 2-item <AC>, the items from <E> i.e. <E> and <P> are considered. The patterns generated are <ACE> and <ACP> and *SC* is calculated as above.

(3) For the frequent 2-item <AP>, there is no candidate 3-itemset because there is no next item after <P>.

The candidate 3-itemset generated by Thread-1 are <ADM>, <ADC>, <ADE>, <ADP>, <ACE> and <ACP>. Among them the patterns viz., <ADC>, <ADP> and <ACP> satisfies δ forms frequent 3-itemset and represented by Table 7.

Itemset	TID's					
<ADC>	3	4	5	7	8	9
<ADP>	3	4	5	7	8	9
<ACP>	3	4	5	7	8	9

Table 7. Frequent 3-Itemsets for <A> By Thread-1

The frequent 3-itemset for <A> is not empty, so the method generates the candidate 4-itemset. They are <ADCE> and <ADCP>. The TID's for <ADCE> is calculated as {3, 4, 5, 7, 8, 9} ∩ {0, 1, 2, 3, 4, 6, 7, 8, 9}={3, 4, 7, 8, 9} and *SC* of <ADCP> is 5. Similarly, for <ADCP>, the TID's are {3, 4, 5, 7, 8, 9}∩{0, 1, 3, 4, 5, 6, 7, 8, 9} = {3, 4, 5, 7, 8, 9}. The SC of <ADCP> is 6 and it is illustrated in Table 8.

Itemset	TID's					
<ADCP>	3	4	5	7	8	9

Table 8. Frequent 4-Itemsets for <A> by Thread-1

Now, candidate 5-itemset for the item <A> is Ø. So Thread-1 stops its execution and returns <AD>, <AC>, <AP>, <ADC>, <ADP>, <ACP> and <ADCP> as frequent items for <A>. Similarly, the other threads generate frequent itemsets for other frequent 1-itemset in parallel as shown from Table 9 to Table 19.

Itemset	TID's					
<FP>	1	3	5	6	8	9

Table 9. Frequent 2-Itemset for <F> by Thread-2

Itemset	TID's					
<IE>	0	1	3	6	8	9
<IP>	0	1	3	6	8	9

Table 10. Frequent 2-Itemset for <I> by Thread-3

Itemset	TID's					
<IEP>	0	1	3	6	8	9

Table 11.frequent 3-Itemset for <I> by Thread-3

Itemset	TID's						
<DC>	0	3	4	5	7	8	9
<DE>	0	3	4	7	8	9	
<DP>	0	3	4	5	7	8	9

Table 12. Frequent 2-Itemset for <D> by Thread-4

Itemset	TID's						
<DCE>	0	3	4	7	8	9	
<DCP>	0	3	4	5	7	8	9
<DEP>	0	3	4	7	8	9	

Table 13. Frequent 3-Itemset for <D> by Thread-4

Itemset	TID's					
<DCEP>	0	3	4	7	8	9

Table 14. Frequent 4-Itemset for <D> by Thread-4

Itemset	TID's						
<ME>	0	1	2	6	7	8	9
<MP>	0	1	6	7	8	9	

Table 15. Frequent 2-Iemset for <M> By Thread-5

Itemset	TID's					
<MEP>	0	1	6	7	8	9

Table 16. Frequent 3-Itemset for <M> by Thread-5

Itemset	TID's						
<CE>	0	2	3	4	7	8	9
<CP>	0	3	4	5	7	8	9

Table 17. Frequent 2-Itemset for <C> by Thread-6

Itemset	TID's					
<CEP>	0	3	4	7	8	9

Table 18. Frequent 3-Itemset for <C> by Thread-6

Itemset	TID's							
<EP>	0	1	3	4	6	7	8	9

Table 19. Frequent 2-Itemset for <E> by Thread-7

Table 20 depicts the candidate and frequent items, the total number of candidates and frequent items generated by the TB-NPF-VDF for the given TD. The total number of candidate items generated using TB-NPF-VDF is 56, and it is less when compared to VDF.

Itemset	Candidate Items	Total[#]	Frequent Items	Total[$]
1-itemset	{A,B,C, D, E, F,G, H, I, K, M,P,M}	13	{A,C,D,E,F, I,M, P}	8
2-itemset	{AF,AI,AD,AM,AC,AE,AP,FI,FD,FM,FC,FE,FP,ID,IM, IC,IE,IP,DM,DC,DE,DP,MC,ME,MP,CE,CP,EP}	28	{AD,AC,AP,FP,IE,IP,DC,DE,DP, ME,MP,CE,CP,EP}	14
3-itemset	{ADM,ADC,ADE,ADP,ACE,ACP,IEP,DCE,DCP,DEP, MEP, CEP}	12	{ADC, ADP, ACP, IEP,DCE, DCP, DEP,MEP, CEP}	9
4-itemset	{ADCE, ADCP,DCEP}	3	{ADCP, DCEP}	2
Total		**56**		**33**

[#]Number of candidate items [$]Number of frequent items

Table 20. Details of Itemsets for TD

4. Experimental Results and Discussion

The algorithms viz., Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF were implemented using the Python programming language (version 3.8.2). To estimate the performance of TB-NPF-VDF, the research work used four real-time datasets downloaded from the FIMI repository and an open-source Data Mining Library. Table 21 describes the characteristics of datasets. The purpose of using these datasets is that they have been used as a reference by researchers primarily for FPM and ARM-based research. To do a uniform and fair comparison, the experiments for all the datasets of all algorithms were conducted using the same software and hardware configurations. The experiments were performed using 8.00GB RAM, Intel Core i7 with 2.40GHz 64-bit processor and Windows 8.1. All algorithms' runtime performance (Matrix-Apriori [7], VDF, NPF-VDF, TB-NPF-VDF) for the four datasets with different min_sup percentages ranging from 20% to 70% were tabulated in Table 22.

Datasets	Transaction count	Item count	Average item count/transaction
chess	3196	75	37.00
mushrooms	8416	119	23.00
T25i10d10k	9976	929	24.77
c20d10k	10000	192	20.00

Table 21. Characteristics of Datasets

min_sup (%)	Runtime (in Sec.)			
	Matrix -Apriori	VDF	NPF-VDF	TB-NPF-VDF
chess				
20	20.7578	16.8578	13.3578	6.5267
30	19.6365	16.0452	12.1455	5.0325
40	17.7750	14.0750	10.0720	4.5635
50	16.3028	13.3017	9.0017	3.2634
60	15.3625	12.7943	8.2934	2.4571
70	14.8546	11.9825	7.4822	2.0012
mushroom				
20	23.2135	21.1215	18.0016	12.1024
30	21.3426	20.0462	17.0642	11.5642
40	20.0035	19.7083	14.1038	10.7869
50	19.2002	18.2058	13.2044	10.0063
60	18.0805	17.7898	12.7240	8.5698
70	17.5652	15.9575	11.4530	7.9586
t25i10d10k				
20	25.2145	23.3254	20.3325	15.1267
30	23.9625	21.4578	19.4258	13.9568
40	21.5467	20.0025	17.9857	12.0127
50	20.3859	18.7621	16.2456	11.6321
60	19.5321	18.0056	15.0012	10.5212
70	18.4521	16.0527	13.7564	9.2451
c20d10k				
20	26.0014	24.4253	22.8342	17.7586
30	24.9532	22.6752	21.5062	15.9802
40	22.4251	21.9546	20.0412	13.7542
50	21.5621	19.4316	18.8562	11.9892
60	20.1425	19.0012	17.0124	11.0016
70	19.1478	17.5242	15.9351	10.0142

Table 22. Performance Results

Figures 2 to 5 show the graphical representation of the runtime comparison between the algorithms viz., Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for the datasets, namely chess, mushroom, t25i10d10k and c20d10k, respectively. From Table 22 and from figures 2 to 5, it was observed that the runtime performance of TB-NPF-VDF outperforms than Matrix-Apriori, VDF and NPF-VDF. On an average, the runtime performance is improved from 20.3092 to 9.9094.

Further, to prove statistically, a Welch two-sample *t*-test is being performed between the runtimes of Matrix-Apriori and TB-NPF-VDF. The test was done to determine whether the mean runtimes of Matrix-Apriori and TB-NPF-VDF are equal to each other or not. The null hypothesis is taken as that the two mean runtimes are equal, and the alternative is that they are not equal. The test is performed using the R tool for each dataset, and the results are tabulated in Table 23.

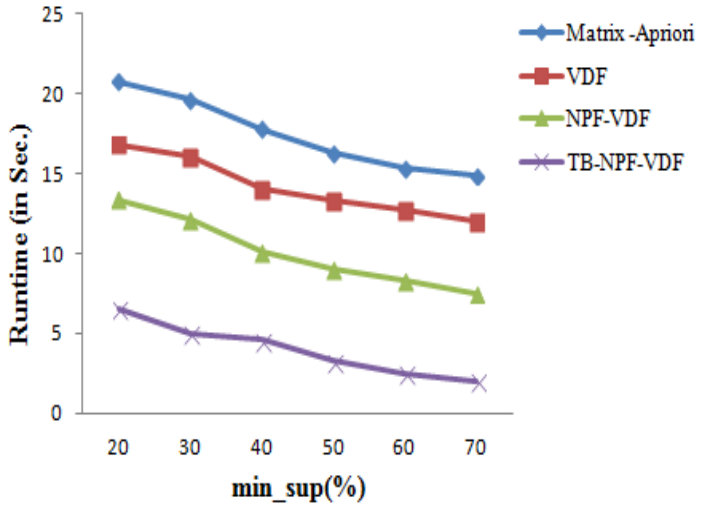


Fig. 2. The execution time of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for chess dataset

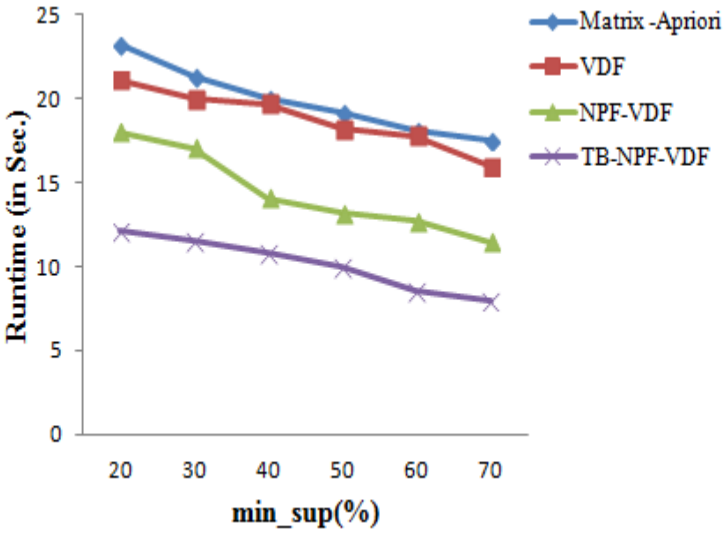


Fig. 3. The execution time of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for mushroom dataset

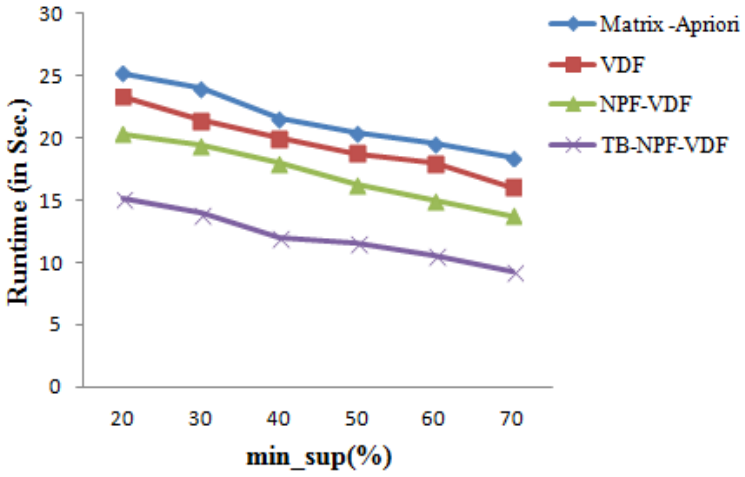


Fig. 4. The execution time of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for t25i10d10k dataset

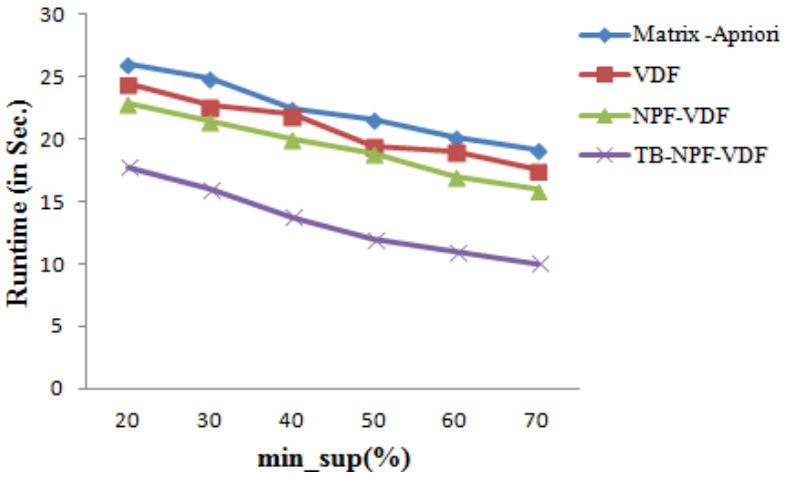


Fig. 5. The execution time of Matrix-Apriori, VDF, NPF-VDF and TB-NPF-VDF for c20d10k dataset

Dataset	p-value
chess	1.207×10^{-06}
mushroom	6.785×10^{-06}
t25i10d10k	5.611×10^{-05}
c20d10k	0.0002914

Table 23. Results of *t*-Test

From the observation of *t*-test results, it is noted that for all datasets, the p-value is ≤ 0.05 (5%) which concluded that the two means are not equal, which means that there are significant differences between the runtimes. Therefore, the proposed method TB-NPF-VDF is more efficient in terms of runtime than the others.

The reason for enhancing the performance is that the concurrent execution of the tasks using a multithreaded approach speeds applications up and reduced the time required for execution by utilizing the CPU effectively. With novel pattern generation, the set of candidate elements generated is less than the existing ones. Further, it scans the database only once during the entire process.

5. Conclusion

Many FPM algorithms were introduced in the field of data mining. Each algorithm has its own merits and demerits and is unsuited for all real-life situations. A new approach called TB-NPF-VDF has been introduced in this research article to discover the frequent patterns that efficiently combine the power of VDF, NPF, and multithread concepts. Experiments were carried out on real-time datasets using python implementation for the existing and proposed methods. TB-NPF-VDF has been proven to be superior to other sequential approaches through memory usage and run time. The main advantage is that it discovers frequent patterns with less time and saves memory with jagged array representation for the VDF matrix. In future, the work can be improved by applying new and efficient optimization techniques.

References

[1] Guo, Y. M.; Wang, Z. J. (2010): A vertical format algorithm for mining frequent item sets. Proceedings of 2nd International Conference on Advanced Computer Control (IEEE Xplore), **4**, pp. 11-13.
[2] Han, J.; Kamber, M.; Pei, J. (2011): *Data mining concepts and techniques*, 3rd edn. Morgan Kaufmann.
[3] Aqra, I.; Herawan, T.; Ghani, N. A.; Akhunzada, A.; Ali, A.; Razali, R. B.; Choo, K. K. R. (2018): A novel association rule mining approach using TID intermediate itemset. PloS one, **13**(1), pp. 01-32.
[4] Subhashini, A.; Karthikeyan, M. (2019): Itemset Mining using Horizontal and Vertical Data Format, International Journal for Research in Engineering Application & Management. **5**(3) pp. 534-539.
[5] Gawwad, M. A.; Ahmed, M. F.; Fayek, M. B. (2017): Frequent itemset mining for big data using greatest common divisor technique. Data Science Journal, **16**(25), pp. 1-10.
[6] Usha, D.; Rameshkumar, K. (2014): A Complete Survey on application of Frequent Pattern Mining and Association Rule Mining on Crime Pattern Mining. International Journal of Advances in Computer Science and Technology, **3**(4), pp. 264-275.
[7] Pavón, J.; Viana, S.; Gómez, S. (2006): Matrix Apriori: Speeding up the Search for Frequent Patterns. Databases and Applications, pp. 75-82.
[8] Sumathi, P.; Murugan, S. (2018): A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach. International Journal of Computer Sciences and Engineering, **6**(11), pp. 152-157.
[9] Chon, K.W.; Hwang, S. H.; Kim, M. S. (2018): GMiner: A fast GPU-based frequent itemset mining method for large-scale data. Information Sciences, **439**, pp. 19-38.

[10] Huang, Y. S.; Yu, K. M.; Zhou, L. W.; Hsu, C. H.; Liu, S. H. (2013): Accelerating parallel frequent itemset mining on graphics processors with sorting. Proceedings of IFIP International Conference on Network and Parallel Computing, pp. 245-256.
[11] Huang, C. H.; Leu, Y. (2015): A LINQ-based conditional pattern collection algorithm for parallel frequent itemset mining on a multi-core computer. Proceedings of ASE BigData & Social Informatics, pp. 1-6.
[12] Zong-Yu, Z.; Ya-Ping, Z. (2012): A parallel algorithm of frequent itemsets mining based on bit matrix. Proceedings of IEEE International Conference on Industrial Control and Electronics Engineering, pp. 1210-1213.
[13] Tanna, P.; Ghodasara, Y. (2015): Analytical Study and Newer Approach towards Frequent Pattern Mining using Boolean Matrix. IOSR Journal of Computer Engineering, **17**(3), pp. 105-109.
[14] Jen, T. Y.; Marinica, C.; Ghariani, A. (2016): Mining frequent itemsets with vertical data layout in MapReduce. Proceedings of International Workshop on Information Search, pp. 66-82.
[15] Vijay Kumar, G.; Valli Kumari, V. (2013): Parallel Regular-Frequent Pattern Mining in Large Databases. International Journal of Scientific & Engineering Research, **4**(6).
[16] Gan, W.; Lin, J. C. W.; Fournier-Viger, P.; Chao, H. C.; Yu, P. S. (2019): A survey of parallel sequential pattern mining. ACM Transactions on Knowledge Discovery from Data (TKDD), **13**(3), pp. 1-34.
[17] Huynh, B.; Trinh, C.; Dang, V.; Vo, B. (2019): A parallel method for mining frequent patterns with multiple minimum support thresholds, International Journal of Innovative Computing. Information and Control, **15**(2), pp. 479-488.
[18] Qiu, H.; Gu, R.; Yuan, C.; Huang, Y. (2014): YAFIM: a parallel frequent itemset mining algorithm with spark. Proceedings of IEEE International Parallel & Distributed Processing Symposium Workshops, pp. 1664-1671.
[19] Shruti, I.; Abhay, K. (2018): Parallel Eclat with Large Data Base Parallel Algorithm and Improve its Effectiveness. International Journal of Engineering Trends and Technology, **60**(3), pp. 180-183.
[20] D. Kalpana, Data Mining Apriori Algorithm Implementation Using R, International Research journal of Engineering and Technology. **4**(11), pp. 1810- 1815.
[21] Sumathi, P.; Murugan, S. (2021): GNVDF: A GPU-accelerated Novel Algorithm for Finding Frequent Patterns Using Vertical Data Format Approach and Jagged Array. International Journal of Modern Education and Computer Science (IJMECS), **13**(4), pp. 28-41.

Authors Profile

P.Sumathi received her B.Sc and M.Sc degrees in Computer Science from Seethalakshmi Ramaswami College (affiliated to Bharathidasan University), Tiruchirappalli, India in 2001 and 2003 respectively. She received her M.Phil degree in Computer Science in 2008 from Bharathidasan University. She is presently working as an Assistant Professor in the Department of Computer Science, Vysya College, Salem. She is currently pursuing Ph.D, a degree in Computer Science in Bharathidasan University. Her research interests include Data Mining, Data structures and Database concepts.

S.Murugan received his M.Sc degree in Applied Mathematics from Anna University in 1984 and M.Phil degree in Computer Science from Regional Engineering College, Tiruchirappalli in 1994. He is an Associate Professor in the Department of Computer Science, Nehru Memorial College (Autonomous), affiliated to Bharathidasan University since 1986. He has 32 years of teaching experience in the field of Computer Science. He has completed his Ph.D degree in Computer Science with a specialization in Data Mining from Bharathiyar University in 2015. His research interest includes Data and Web Mining. He has published more than 25 research articles in reputed National and International journals.

V.Umadevi obtained her M.Sc degree in Computer Science & Information Technology and M.Phil degree in Computer Science from Madurai Kamaraj University. She has completed her Ph.D degree in Computer Science from CMJ University. Besides, she has received M.Tech and MBA degrees. She has 15 years of teaching experience in Computer Science. Her area of teaching and research interests include Management Information Systems, Project Management and Wireless Sensor Networks. She has published 28 research papers in National and International journals and authored three books. Also produced one Ph.D candidate. She has received National Award for "South Indian Achiever" in March 2020 and a "Lifetime Achiever" award from International Lions Club in March 2021. She has published a patent entitled "AI abetted material synthesising for hybrid metal rubber composite and 3D Printing" in August 2021.

I.J. Modern Education and Computer Science, **2021, 4, 28-41**
Published Online August 2021 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2021.04.03

GNVDF: A GPU-accelerated Novel Algorithm for Finding Frequent Patterns Using Vertical Data Format Approach and Jagged Array

P. Sumathi
Research Scholar, Nehru Memorial College (Affiliated to Bharathidasan University), Puthanampatti, Tiruchirappalli-Dt, Tamil Nadu, India - 621 007
Email:sumiparasu@gmail.com

S.Murugan
Associate Professor, Nehru Memorial College (Affiliated to Bharathidasan University), Puthanampatti, Tiruchirappalli-Dt, Tamil Nadu, India - 621 007
Email:murugan_nmc@hotmail.com

Received: 01 June 2021; Accepted: 24 July 2021; Published: 08 August 2021

Abstract: In the modern digital world, online shopping becomes essential in human lives. Online shopping stores like Amazon show up the "Frequently Bought Together" for their customers in their portal to increase sales. Discovering frequent patterns is a fundamental task in Data Mining that find the frequently bought items together. Many transactional data were collected every day, and finding frequent itemsets from the massive datasets using the classical algorithms requires more processing time and I/O cost. A GPU accelerated Novel algorithm for finding the frequent patterns using Vertical Data Format (GNVDF) has been introduced in this research article. It uses a novel pattern formation. In this, the candidate *i*-itemsets is divided into two buckets viz., Bucket-1 and Bucket-2. Bucket-1 contain all the possible items to form candidate-(i+1) itemsets. Bucket-2 has the items that cannot include in the candidate-(i+1) itemsets. It compactly employs a jagged array to minimize the memory requirement and remove common transactions among the frequent 1-itemsets. It also utilizes a vertical representation of data for efficiently extracting the frequent itemsets by scanning the database only once. Further, it is GPU-accelerated for speeding up the execution of the algorithm. The proposed algorithm was implemented with and without GPU usage and compared. The comparison result revealed that GNVDF with GPU acceleration is faster by 90 to 135 times than the method without GPU.

Index Terms: Frequent Patterns, GNVDF, Graphical Processing Unit, Novel Pattern Formation, Vertical Data Format, and Jagged Array.

1. Introduction

Data Mining (DM) is a part of Knowledge Discovery in Databases (KDD) [1] and explores the hidden patterns for business people. It is associated with many fields such as database systems, data warehousing, statistics, machine learning, information retrieval, and high-level computing [2,3]. It is also supported by other sciences like neural networks, pattern recognition, spatial data analysis, image databases and signal processing [2,3]. There are several techniques in data mining like classification, clustering, association rule mining and regression [4]. Frequent Pattern Mining (FPM) is a computationally crucial step in data mining [5]. It is used to determine the frequent patterns and associations from databases such as relational and transactional databases and other data repositories. The Apriori is one of the most important algorithms for finding frequent itemsets. It has many problems such as more database scan and I/O cost, a large amount of time etc., for finding frequent itemsets. So the researchers have made several refinements to Apriori in the last two decades.

However, enhancing speed and reducing memory requirements are the essential parameters while determining the frequent patterns nowadays because of the rise of big data in various domains and sources in human endeavour. Also, when the transactional database size increases, demand for storage is increased and requires high-speed algorithms to find frequent patterns. But with a single-threaded approach, it's tough to minimize time. The GPU accelerated computing employs GPUs along with CPUs. It enables superior performance by supporting a parallel programming paradigm with multiple cores. It saves time and cost in scientific and other high computing tasks [6]. So, researchers

were utilized GPUs in FPM based research. Some research works based on GPUs that motivate this article's proposed work were discussed here.

W. Fang et al. [7] have introduced two implementations for Apriori using GPUs with Single Instruction, Multiple Data (SIMD) architectures. Both methods use a bitmap data structure. They executed the first one on the GPU, avoiding the intermediate data transfer between the GPU and CPU memory. The second one uses both the CPU and GPU for processing with trie structure. They proved that both implementations speed up the processing than the classical Apriori algorithm. S. M. Fakhrahmad et al. [8] have developed different parallel versions of a novel sequential mining algorithm for finding frequent itemsets. The methods are i) assigning each partition to a processor, ii) assigning each column to a processor, and iii) devoting the k^{th} processor to mine the k^{th}-itemsets. These methods were compared experimentally using time complexity, communication rate, and load balancing and proved that the proposed methods outperformed the existing sequential algorithms.

The authors J. Zhou et al. have designed [9] a GPU-based Apriori algorithm with OpenGL to accelerate association rules mining. The experiment proved that the proposed algorithm provides better performance than the classical algorithms. A new pattern-based algorithm called HSApriori has been suggested by D. William Albert et al. [10], and it is based on the parallel processing nature of GPU. In this, the proposed method was tested using both the tidset and bitset representation of the dataset and found that the bitset is more appropriate for parallel processing. Further, they proved from the experiment that the speed of HSApriori is substantially more when compared with traditional HorgeltAprirori.

To solve the limitations of Apriori, a parallel Apriori Map Reduce model has been presented by M. Tiwary et al. [11] using high-performance GPU. They have attached a GPU with every node in a Hadoop cluster. Also, they have used NVIDIA's GPU and JCUDA and JNI for the integration process. From the experiments, it has been proved that it provides better performance in terms of execution time. The downside of the algorithm is that the extra hardware charge is associated with the GPUs in each node in the Hadoop cluster. To overcome the drawbacks in the traditional cluster-based map-reduce, J. Li et al. [12] have designed a multi-GPU based parallel Apriori algorithm to accelerate the calculation process of Apriori. It has been initiated especially to mine association rules in medical data. The analytical results have proved that the proposed method significantly improves the execution speed with a lower cost for medical data.

A novel method called CGMM to suit both sparse and dense datasets has been proposed to mine frequent patterns has been introduced by L. Vu et al. [13]. To increase the speed of the FPM process, it combines both the CPU and GPU. In this method, the CPU uses the FP-tree data structure to perform mining, and the GPU converts the data to bit vectors. The experiments with AMD CPUs and NVIDIA GPU have proved that the performance evaluation of CGMM is faster than the existing sequential FPM and GPApriori. Y. Li et al. [14] have developed a GPU-based algorithm called Multi-level Vertical Closed FIM. In this, a multi-layer vertical data structure has been used to minimize the usage of storage. The implementation is being accelerated with GPU to achieve high-speed computation, mainly on large and sparse datasets.

K.W. Chon et al. [15] have proposed a novel algorithm called GMiner. It is a GPU-based method for finding frequent itemsets on large-scale datasets. It determines the patterns from the first level of the enumeration tree rather than storing and utilizing the patterns at the intermediate levels of the tree. With the computational power of GPUs, the method achieved fast performance and outperformed significantly than the existing sequential and parallel methods. The method also eliminates the skewness problem that the parallel algorithms suffer. A Dynamic Queue and Deep Parallel (D2P) Apriori algorithm was generated by Y. Wang et al. in [16]. In this, the candidate generation process has been parallelized by using the Graph-join and dynamic bitmap queue. It also uses a vertical bitmap structure with low-latency memory on GPU. The experiments have explored that the D2P-Apriori obtained high-speed up, i.e. a 23×speed up ratio compared to the modern CPU methods.

The authors Y. Djenouri et al. [17] have created three High-Performance Computing (HPC)-based versions of Single Scan (SS) for frequent itemset mining viz., GSS, CSS, and CGSS. The GSS, CSS, and CGSS implement SS with GPU, cluster architecture, and GPU with multiple cluster nodes. They have also presented three approaches to reduce cluster load balancing and GPU thread divergence. The experiments have proved that the CGSS performs best in speed than SS, GSS and CSS.

The authors P.Sumathi et al. [18] have developed a memory-efficient implementation for a vertical data format approach in finding frequent patterns using jagged array matrix representation. They have formulated mathematical equations for memory requirements and proved that it reduces the memory requirement than the traditional multidimensional array.

The numerous GPU based FPM algorithms found in the literature have their own merits. But they have some performance, data size and scalability issues [19], which provides a more vital lead to the proposed work. The research article has introduced GNVDF, a novel GPU-accelerated FPM algorithm. It uses a novel pattern generation method to avoid generating many candidate itemsets as classical algorithms and uses a compact jagged array structure to minimize storage space [18]. Further, it uses the VDF format of transactional data to reduce the number of disk accesses.

The remaining paper is organized as follows. Section 2 presents the basic terminologies and definitions, vertical data format, jagged array, and GPU. The description of the proposed methodology with an illustration is presented in

section 3. Section 4 illustrates the experimental results and discussion. Finally, the research article ends with a conclusion in section 5.

2. Basic Concepts

Finding frequent itemsets is essential in mining associations, correlations, and many other relationships among the data. It is used in data classification, clustering, and other data mining tasks. Thus, FPM is focused on data mining research, and this section briefs the fundamental concepts associated with FPM and the study.

A. Basic Terminology

An itemset (set of items) that contains k items is said to be a k-itemset. The set of laptop, printer is a 2-itemset. Frequent patterns are the patterns (itemsets, subsequences, or substructures) that frequently appear in a dataset [2,20]. The support count of the itemset is identified by the number of transactions that contain the itemset. A sequence is an ordered list of itemsets, i.e. set of items purchased together. A subsequence is a sequence of items bought together and frequently occurs in a transactional database known as a sequential pattern. A substructure can be represented in different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences [2].

B. Basic Definitions

Let $I=\{I_1, I_2,\ldots, I_m\}$ be an itemset, and D is a transaction database contains a set of transactions T is a non-empty itemset such that $T \subseteq I$ and each transaction T is associated with a unique identifier TID. Let A be a set of items. A transaction T is said to contain in A if $A \subseteq T$. The format of the association rule is $A \rightarrow B$, where $A \subset I$, $B \subset I$, $A \neq \emptyset$, $B \neq \emptyset$, and $A \cap B = \emptyset$ [21]. Associations rule $A \rightarrow B$ that holds in the transaction database D with support (s) and confidence(c) [1].

Support(s): The support of an association rule $A \rightarrow B$ is defined as the percentage of records that contain $A \cup B$ to the total number of records in the database [22]. It is noted that the support count is increased when an item present in numerous transactions in the database D [22].

Confidence: The confidence of a rule $A \rightarrow B$ is defined as $s(A \rightarrow B)/s(A)$. It is the ratio of the number of transactions that contain all items in the consequent (B), as well as the antecedent (A) to the number of transactions that include all items in the antecedent (A) [23].

The minimum support threshold is used to discover the frequent itemsets from the databases. In contrast, the minimum confidence constraint is applied to those frequent itemsets found previously in determining the best rules.

C. Vertical Data Format

The databases can be represented in FPM algorithms in two data formats. They are i) Horizontal Data Format (HDF) and ii) Vertical Data Format (VDF). HDF represents the items categorized into particular transactions as stored in the database. i.e. it is denoted as <TID, Itemset>, where TID is the transaction ID, and Itemset refers to the items purchased by the customer corresponding to TID. The VDF represents data as transactions categorized into particular items that mean the TIDs are grouped for each item, i.e. VDF is described by <Item, Tid_set>, where item denotes an item in the shop and Tid_set contains the TID's where the item occurs. Fig.1. and Fig.2. show the HDF and VDF of D.

TID	Itemset
0:	**{c,d,e,g,h,i,k,p,m}**
1:	**{b,e,f,g,h,i,p,m}**
2:	**{c,e,m}**
3:	**{a,b,c,d,e,f,g,i,p}**
4:	**{a,b,c,d,e,p}**
5:	**{a,b,c,d,f,h,p}**
6:	**{b,e,f,h,i,p,m}**
7:	**{a,c,d,e,k,p,m}**
8:	**{a,c,d,e,f,i,p,m}**
9:	**{a,c,d,e,f,h,i,p,m}**

Fig.1. HDF of Transaction Database D

D. Jagged Array

A jagged array data structure is an array whose elements are arrays known as "array of arrays" with varying columns in each array/row, and it is shown in Fig.3.

E. Graphical Processing Unit

It is a device specifically designed for graphics processing. It is widely used in large scale hashing and matrix computations because it supports parallelism and serves as the base for mining and machine learning. CUDA and OpenCL are two popular GPGPU programming framework tools. NVIDIA has designed a parallel computing platform and programming called Compute Unified Device Architecture (CUDA) [12,24]. The CUDA-based program can only be run on the NVIDIA-produced GPU. A typical CPU may contain four or eight cores; an NVIDIA GPU consists of thousands of CUDA cores and a pipeline that supports parallel processing on thousands of threads, increasing the speed significantly.

With Numba, the python developer can quickly enter into GPU-accelerated computing. It makes use of both GPU and CPU to facilitate processing-intensive operations viz., deep learning, analytics, and engineering applications. The CUDA Python and Numba help to enhance the speed by targeting both CPUs and NVIDIA GPUs. With this advantage of CUDA python and Numba, the implementation of this proposed work will be GPU accelerated.

Item	Tid_set
a:	{3,4,5,7,8,9}
b:	{1,3,4,5,6}
c:	{0,2,3,4,5,7,8,9}
d:	{0,3,4,5,7,8,9}
f:	{1,3,5,6,8,9}
g:	{0,1,3}
h:	{0,1,5,6,9}
i:	{0,1,3,6,8,9}
k:	{0,7}
m:	{0,1,2,6,7,8,9}
p:	{0,1,3,4,5,6,7,8,9}

Fig.2. VDF of Transaction Database D

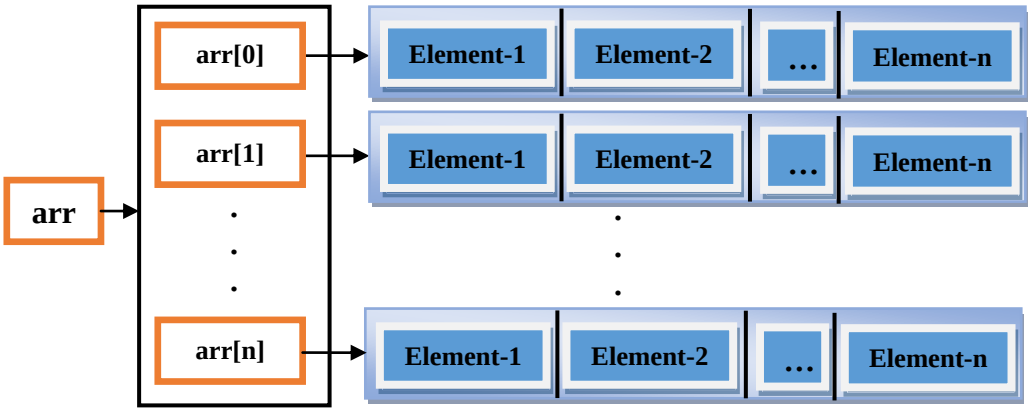


Fig.3. Jagged array representation

3. Proposed Methodology

The main objective of the proposed work is to find the essential frequent itemsets from the transaction database with less memory space and time by ignoring the least probable ones. The method used Jagged array storage structure [16] and GPU to minimize memory usage and execution time. The proposed method first removes the null/void transactions in the dataset. Null/void transactions are those which contain only one item. Then the dataset is scanned once and converted into VDF format. The support count (*SC*) for each item is calculated by counting the number of transactions that contain each item. Now the candidate 1-itemset C_1 is formed. Next, the frequent 1-itemset is formed by removing the items whose *SC*<min_sup(δ) and stored it in Jagged array representation [18] in sorted order based on *SC*. From L_1 the common transactions among all items are determined either by intersecting or ANDing the transaction in each item, and it is preserved in the Common Transaction List (C_{TID_list}). The transactions in C_{TID_list}'s are removed from each item in L_1, forming the final frequent 1-itemset. The *SC* for each item in L_1 is updated by *SC* - *n*, where *n* is the number of transactions in C_{TID_list}. Next, the new min_sup (δ_{new}) is determined as $\delta_{new} = \delta - n$, and it will be the min_sup from the 2nd iteration onwards.

Before finding the frequent 2-itemset, the final frequent 1-itemset is divided into two logical buckets, LB_1 and LB_2, respectively. LB_1 contains all the items whose $SC = \delta_{new,}$ and the rest will be placed LB_2. The itemset combinations among the items in LB_1 are least probable of being a candidate 2-itemset because the SC of each item is equal to δ_{new}. So it is not considered for generating candidate 2-itemset. The candidate 2-itemsets patterns are generated by combining each item I_x in LB_1 with each item I_y in LB_2 and each item I_z in LB_2 with I_{z+1} in LB_2 until the last item in LB_2. The itemset combination that ends with the last item in LB2 will be placed in C2_2 and the rest in C_{2_1}. From C_{2_1} and C_{2_2}, the items whose SC below the δ_{new} is removed as infrequent and formed L_{2_1} and L_{2_2}.

For generating candidate 3-itemset, each itemset I_x in L_{2_1} is combined with the next item Iy in LB_2 after the last item in I_x. Similar to the previous iteration, the combinations that end with the last item in LB2 are placed in C_{3_2} and rest in $C_{3\_1.}$ It is noted that the itemset combinations in L_{2_2} are not used in the formation of candidate 3-itemsets. The L_{3_1} and L_{3_2} were formed by removing the infrequent itemsets in C_{3_1} and C_{3_2}. The process is continued until L_{n_1} is not null. Further, to increase the execution speed of the proposed method, it is being accelerated with GPU. The proposed algorithm (Algorithm 1) is shown below, and the workflow diagram is shown in Fig.4.

Algorithm 1 A**lgorithm for finding frequent itemsets**

Input : D - a dataset with n transactions;
δ - minimum support threshold;
Output : Frequent patterns;

1: D ← eliminate_null(D);
2: vdf ← scan D and convert it in vertical data format;
3: L_1← one_frequent_itemset(vdf, δ);
4: C_{TID_list} ← find_common_TID(L_1);
5: L_1← remove the transactions in C_{TID_list} for each item in L_1;
6: δ_{new} ← δ - number of transactions in C_{TID_list};
7: LB_1 ← {∀ frequent 1-itemset | SC=δ_{new} };
8: LB_2 ← {∀ frequent 1-itemset | SC > δ_{new}};
9: L_{2_1}, L_{2_2} ← find_two_freq_itemset(LB_1,LB_2,δ_{new});
10: i=2;
11: while $L_{i\_1} \neq Ø$ do
12: L_{i+1_1},L_{i+1_2} ← n_frequent_itemset(L_{i_1},LB_2,δ_{new});
13: i=i+1;
14: end while

procedure eliminate_null(D - a dataset with *n* transactions)

1: for each $T_i \in D$ do
2: cnt←count the number of items in T_i;
3: if cnt == 1 then
4: remove T_i from D;
5: end if;
6: end for;
7: return D;

procedure one_frequent_itemset(D: Dataset after removing null transactions; δ :minimum support threshold)

1: L_1←Ø;
2: for each $item_i$ in D do
3: TID_{list}←transactions in which $item_i$ occurs;
4: SC←count the number of transactions in TID_{list}
5: if SC ≥ δ then
6: add {$item_i$, TID_{list}, SC}into L_1;
7: end if
8: end for
9: sort L_1 and store it in jagged array format;
10: return L_1;

procedure find_common_TID (L_1: frequent 1-itemset)

1: n←find the number of items in L_1;
2: C_{TID_list}←{$TID_{list1} \cap TID_{list2} \cap \ldots \cap TID_{listn}$};
3: return C_{TID_list};

procedure two_freq_itemset (LB_1: frequent 1-itemset1, LB_2: frequent 1-itemset2, δ:minimum support)

last_item←find last item in LB_2;
for each itemi in LB_1 do
    for each $item_j$ in LB_2 do
        new_pattern ← <$item_i item_j$>;
        new_tid←TIDs($item_i$)∩TIDs($item_j$);
        new_sc←count the transactions in new_tid;
        if new_pattern contains last_item then
            append{new_pattern,new_tid,new_sc} in C_{2_2};
        else
            append{new_pattern,new_tid,new_sc} in C_{2_1};
        end if
    end for
end for
L_{2_1}←{C_{2_1} | SC(C_{2_1}) ≥ δ};
L_{2_2}←{C_{2_2} | SC(C_{2_2}) ≥ δ};
return L_{2_1}, L_{2_2}

procedure n_frequent_itemset(L_{i_1}: frequent i-itemset1, LB_2: frequent 1-itemset2, δ_{new}: minimum support)

for each $item_i$ in L_{i_1} do
    last_item←find the last item in $item_i$;
    for each $item_j$ in LB_2 after last_item do
        new_item←{<$item_i item_j$>};
        new_tid←TIDs($item_i$)∩ TIDs($item_j$);
        new_sc←count the transactions in new_tid;
        if new_item contains last element in LB_2 then
            append{new_item,new_tid,new_sc}in C_{n_2};
        else
            append{new_item,new_tid,new_sc}in C_{n_1};
        end if
    end for
end for
L_{n_1}←{C_{n_1} | SC(C_{n_1}) ≥ δ};
L_{n_2}←{C_{n_2} | SC(C_{n_2}) ≥ δ};
return L_{n_1},L_{n_2}

The main advantage of the proposed method is that it reduces the number of candidate itemsets to be generated in each iteration because the itemsets in L_{i_2}, for $i \geq 3$ will not be considered for creating candidate itemsets and removal of items in *CTL* in final L_1. Additionally, GPU and Jagged array enhance the performance in terms of speed and usage of memory.

A. Memory Requirement Calculation

From [25,18], it was observed that the memory requirement using a jagged array structure for the frequent itemsets could be calculated based on the following equation.

$$TM = \sum_{i=1}^{itemset_i \neq \phi} TM_i - rbytes_i \quad (1)$$

where, TM_i is the total memory required for the candidate *i*-itemset, and $rbytes_i$ is the memory occupied by the infrequent/rare items in the candidate *i*-itemset. By subtracting $rbytes_i$ from TM_i, the memory for L_i i.e., frequent *i*-itemsets can be found.

TM_i and $rbytes_i$ were calculated using equations 2 and 3, respectively.

$$TM_i = \sum_{\forall item \in \{itemset_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \tag{2}$$

$$rbytes_i = \sum_{\forall item \in \{in-frequent_i\}} SC_{item} \times sizeof(tid) + sizeof(item) \tag{3}$$

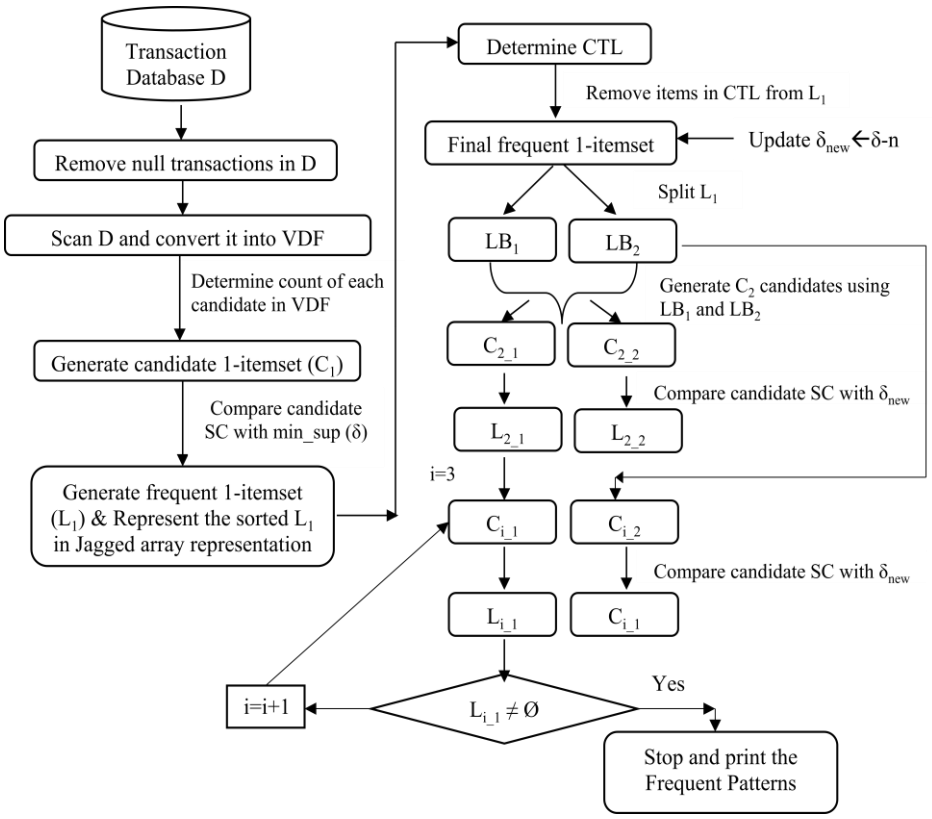


Fig.4. Workflow of GNVDF

As in [25], the GNVDF also used the same jagged storage structure for storing frequent itemsets, and the amount of memory requirement was calculated as follows. It first fetches the common transactions among items in the frequent 1-itemsets and then removes them from frequent 1-itemsets. Suppose if the frequent 1-itemset contains *n* items say $item_1$, $item_2$, $item_3$,…, $item_n$ and the corresponding *TID* lists say $TID\text{-}List_1$, $TID\text{-}List_2$, $TID\text{-}List_3$,…,$TID\text{-}List_n$, then the common *TID*s(C_{TID}) among the *n* items were found by set intersection operation using equation (4) shown below.

$$C_{TID} = \{TID - List_1\} \cap \{TID - List_2\} \cap ... \cap \{TID - List_n\} \tag{4}$$

The memory space required for C_{TID} was calculated using equation (5).

$$CM = \sum_{i=1}^{length(C_{TID})} sizeof(C_{TID_i}) \tag{5}$$

Since the method removes the C_{TID} from frequent 1-itemsets, the C_{TID} need not be repeated in the subsequent frequent itemsets, saving memory space considerably. The amount of memory saved (*MS*) for the entire dataset was calculated using equation (6).

$$MS = count(itemset_i) \times CM + \sum_{i=2}^{itemset_i \neq \phi} \{count(itemset_{i-1}) + count(itemset_{i-2})\} \times CM \tag{6}$$

where, $count(itemset_1)$, $count(itemset_{i\_1})$, and $count(itemset_{i\_2})$ refer to the number of items in frequent 1-itemset, first and the second part of frequent *i*-itemsets, respectively. Thus, the total memory required for the frequent itemsets of the entire dataset using the proposed method was calculated using equation (7).

$$TM_{final} = \{ \sum_{i=1}^{itemset_i \neq \phi} TM_i - rbytes_i \} - MS \quad (7)$$

B. Proposed Methodology: An Example

The vertical representation of transaction dataset D as shown in Fig. 2 is considered to understand the proposed methodology. It contains 12 items viz., {a, b, c, d, e, f, g, h, i, k, m, p}. Each item is represented by a row containing the name of the item and the transactions in which the item occurs (TIDs) [26]. Let δ is 6. From Fig. 2, the candidate 1-itemset is calculated. The candidate 1-itemset contains all the items in D, the TIDs in which the item occurs and the SC. It is shown in Table 1.

Table 1. Candidate 1-itemset(C_1)

Item	TIDs	SC
a	{3, 4, 5, 7, 8, 9}	6
b	{1, 3, 4, 5, 6}	5
c	{0, 2, 3, 4, 5, 7, 8, 9}	8
d	{0, 3, 4, 5, 7, 8, 9}	7
e	{0, 1, 2, 3, 4, 6, 7, 8, 9}	9
f	{1, 3, 5, 6, 8, 9}	6
g	{0, 1, 3}	3
h	{0, 1, 5, 6, 9}	5
i	{0, 1, 3, 6, 8, 9}	6
k	{0, 7}	2
m	{0, 1, 2, 6, 7, 8, 9}	7
p	{0, 1, 3, 4, 5, 6, 7, 8, 9}	9

From the table above, the items viz., b, g, h and k are removed as infrequent because the items do not satisfied δ. The frequent 1-itemset is shown in Table 2. Since the common transactions (CTL) are stored in Table 3, they are removed from each item in L_1, the final L_1 is formed, and it is shown in Table 4.

Table 2. Frequent 1-itemset(L_1)

1- Itemset	TIDs								
a	3	4	5	7	8	9			
f	1	3	5	6	8	9			
i	0	1	3	6	8	9			
d	0	3	4	5	7	8	9		
m	0	1	2	6	7	8	9		
c	0	2	3	4	5	7	8	9	
e	0	1	2	3	4	6	7	8	9
p	0	1	3	4	5	6	7	8	9

Now the new_min is calculated by removing the number of items in CTL as $\delta_{new} = \delta - n = 6-2 = 4$. The logical buckets from final L_1, i.e. LB_1 and LB_2, are shown in Tables 5 and 6.

To reduce the storage space requirement further, this method finds the common transaction in which the all items occurs either by AND operation or intersection of the TIDs of all frequent 1-itemset. i.e.{3,4,5,7,8,9} ∩ {1,3,5,6,8,9} ∩ {0,1,3,6,8,9}∩{0,3,4,5,7,8,9}∩{0,1,2,6,7,8,9}∩{0,2,3,4,5,7, 8,9}∩{0,1,2,3,4,5,6,7,8,9}∩{0,1,3,4,5,6,7,8,9} = {8,9} and it is stored in CTL. The CTL is shown in Table 5.

Table 3. Common Transaction List(CTL)

CTL	
8	9

Table 4. Final Frequent 1-itemset(L_1)

1- Itemset	TIDs						
a	3	4	5	7			
f	1	3	5	6			
i	0	1	3	6			
d	0	3	4	5	7		
m	0	1	2	6	7		
c	0	2	3	4	5	7	
e	0	1	2	3	4	6	7
p	0	1	3	4	5	6	7

Table 5. Logical Bucket-1(LB_1)

1- Itemset	TIDs			
a	3	4	5	7
f	1	3	5	6
i	0	1	3	6

Table 6. Logical Bucket-2(LB_2)

1- Itemset	TIDs						
d	0	3	4	5	7		
m	0	1	2	6	7		
c	0	2	3	4	5	7	
e	0	1	2	3	4	6	7
p	0	1	3	4	5	6	7

The 2-itemset combinations viz., ad, am, ac, ae, fd, fm, fc, fe, id, im, ic, ie, dm, dc, de, mc, me, mp, and ce are in C_{2_1} and the items viz., ap, fp, ip, dp, mp, cp and ep are stored in C_{2_2}. The possible combinations viz., af, ai and fi need not be generated. It is shown in Tables 7 and 8 respectively.

Table 7. Candidate 2-itemset - Part I

C_{2_1}	TIDs	SC
ad	3, 4, 5, 7	4
am	7	1
ac	3, 4, 5, 7	4
ae	3,4,7	3
fd	3,5	2
fm	1,6	2
fc	3,5	2
fe	1,3,6	3
id	0,3	2
im	0,1,6	3
ic	0,3	2
ie	0, 1, 3, 6	4
dm	0	1
dc	0, 3, 4, 5, 7	5
de	0, 3, 4, 7	4
mc	0,2	2
me	0, 1, 2, 6, 7	5
ce	0, 2, 3, 4, 7	5

Table 8. Candidate 2-itemset - Part II

C_{2_2}	TIDs	SC
ap	3, 4, 5, 7	4
fp	1, 3, 5, 6	4
ip	0,1,3,6	4
dp	0, 3, 4, 5, 7	5
mp	0, 1, 6, 7	4
cp	0, 3, 4, 5, 7	5
ep	0, 1, 3, 4, 6, 7	6

The items viz., am, ae, fd, fm, fc, fe, id, im, ic, dm and mc are infrequent in C_{2_1} and no item is infrequent in C_{2_2}. Therefore, the frequent 2-itemsets are stored in L_{2_1} and L_{2_2} in jagged array notation as shown in Tables 9 and 10 respectively. The candidate 3-itemsets from L_{2_1} and LB_2 viz., adm, adc, ade, ace and dce, stored in C_{3_1} and the patterns adp, acp, iep, dep, mep, dcp and cep are kept in C_{3_2} as shown in Tables 11 and 12 respectively. The L_{3_1} and L_{3_2} are shown in Tables 13 and 14, respectively. Similarly, C_{4_1} and C_{4_2} are shown in Tables 15 and 16, respectively. L_{4_1} and L_{4_2} are $L_{4\_1} = \{\}$ and L_{4_2} is shown in Table 17.

Table 9. Frequent 2-itemset - Part I

L_{2_1}	TIDs				
ad	3	4	5	7	
ac	3	4	5	7	
ie	0	1	3	6	
dc	0	3	4	5	7
de	0	3	4	7	
me	0	1	2	6	7
ce	0	2	3	4	7

Table 10. Frequent 2-itemset - Part II

L_{2_2}	TIDs					
ap	3	4	5	7		
fp	1	3	5	6		
ip	0	1	3	6		
dp	0	3	4	5	7	
mp	0	1	6	7		
cp	0	3	4	5	7	
ep	0	1	3	4	6	7

Table 11. Candidate 3-itemset - Part I

C_{3_1}	TIDs	SC
adm	7	1
adc	3, 4, 5, 7	4
ade	3,4,7	3
ace	3,4,7	3
dce	0, 3, 4, 7	4

Table 12. Candidate 3-itemset - Part II

C_{3_2}	TIDs	SC
adp	3, 4, 5, 7	4
acp	3,4,5,7	4
iep	0, 1, 3, 6	4
dep	0, 3, 4, 7	4
mep	0, 1, 6, 7	4
dcp	0, 3, 4, 5, 7	5
cep	0, 3, 4, 7	4

Table 13. Frequent 3-itemset - Part I

L_{3_1}	TIDs			
adc	3	4	5	7
dce	0	3	4	7

Table 14. Frequent 3-itemset - Part II

L_{3_2}	TIDs				
adp	3	4	5	7	
acp	3	4	5	7	
iep	0	1	3	6	
dcp	0	3	4	5	7
dep	0	3	4	7	
mep	0	1	6	7	
cep	0	3	4	7	

Table 15. Candidate 4-itemset - Part I

C_{4_1}	TIDs	SC
adce	3,4,7	3

Table 16. Candidate 4-itemset - Part II

C_{4_2}	TIDs	SC
adcp	3, 4, 5, 7	4
dcep	0, 3, 4, 7	4

Table 17. Frequent 4-itemset - Part II

L_{4_2}	TIDs			
adcp	3	4	5	7
dcep	0	3	4	7

Now, L_{4_1} is an empty list, so the algorithm terminates. It is observed from the experiment that the time needed for finding frequent items for sample dataset D in the example without the use of GPU is 0.8111 sec, whereas the wall time is 0.0073ms with GPU. The total memory requirement for the frequent itemset for the above dataset using the method in [18] is TM = 124+210+137+32=503 bytes. By using GNVDF, the memory requirement for the common transaction is CM = 2+2 = 4 bytes and the amount of memory saved using the proposed method is MS = (8×4) + {(7×4 + 7×4) + (2×4 + 7×4) + (0×4 + 2×4)} = 32 + 56 + 36 + 8 = 132 bytes. Therefore, the final memory requirement is TM_{final} = 503 - 132 = 371 which is 26.24% of memory saved for this example dataset compared to the memory requirement in [18]. It is also noted that the number of common transactions is directly proportional to the amount of memory saved.

4. Experimental Results and Discussion

The proposed algorithm was implemented using Python with CUDA Toolkit with NVIDIA GPU. An extensive experiment was conducted using four real-time datasets viz., chess, mushroom, t25i10d10k and c20d10k to evaluate the performance of GNVDF. The datasets and their details were shown in Table 18. They were obtained from the FIMI repository and an open-source Data Mining Library. The reason for choosing those datasets is that many researchers used those bench-mark datasets in Frequent Itemset Mining (FIM) and Association Rule Mining(ARM) based research. The runtime performance of the proposed method without GPU acceleration was obtained for each dataset, with the minimum threshold values ranging from 20% to 70% and is shown in Table 19. Similarly, the proposed algorithm was executed with GPU acceleration using the same minimum support range and results were tabulated in Table 20.

Table 18. Datasets used in experiments with their properties

Datasets	No. of transactions	No. of items	Average item count per transaction
chess	3196	75	37.00
mushrooms	8416	119	23.00
t25i10d10k	9976	929	24.77
c20d10k	10000	192	20.00

Table 19. Runtime (in ms) performance of the proposed algorithm without GPU

DS# → / MS* ↓	chess	mushroom	t25i10d10k	c20d10k
20	10759.6	14501.6	16332.5	16334.2
30	9845.5	13464.2	16225.8	16006.2
40	7972	11103.8	13885.7	15441.2
50	7101.7	10224.4	12645.6	14956.2
60	6293.4	9834	11101.2	13412.4
70	5082.2	8253	9256.4	12035.1

Table 20. Runtime (in ms) performance of the proposed algorithm with GPU-acceleration

DS# → / MS* ↓	chess	mushroom	t25i10d10k	c20d10k
20	119.5511	145.0160	161.7079	161.7248
30	107.0163	138.0940	156.0173	158.4772
40	83.9158	117.2770	129.7729	131.9761
50	73.2134	104.5091	108.3670	110.6496
60	64.2184	88.8096	102.4380	105.3511
70	53.4968	74.0512	83.6424	92.9924

#DS-Dataset *MS-min_sup(δ)

The graphical representation of the runtime performance of each dataset with and without GPU usage was illustrated in Fig.5. From tables 19 and 20, it was observed that when the number of items and transactions in a dataset increases, the time required for finding frequent patterns also increases. In general, there is an inverse relationship between the min_sup threshold and the time needed to determine the frequent patterns. i.e., when the min_sup threshold is increased, the number of generated candidate itemsets, followed by frequent patterns, is minimized, consuming less time for the higher threshold.

Fig.5. showed that the GPU acceleration significantly enables the execution speed of the proposed methodology, and GNVDF with GPU is faster by 90 to 135 times when compared with GNVDF without GPU acceleration. The reason for the performance enhancement is that the GPUs have many computing cores that allow the parallel execution of computation-intensive tasks. Since the GNVDF uses the VDF approach, the number of database scans is restricted to one [27] for determining each item's support count, which in turn reduces the overtime for finding the frequent patterns. But, VDF requires more memory for additional information like TID's than HDF [27], so a Jagged array has been used to minimise memory space is an advantage. Further, the elements in CTL removed from frequent 1-itemset save the memory space considerably more than the existing classical algorithms.

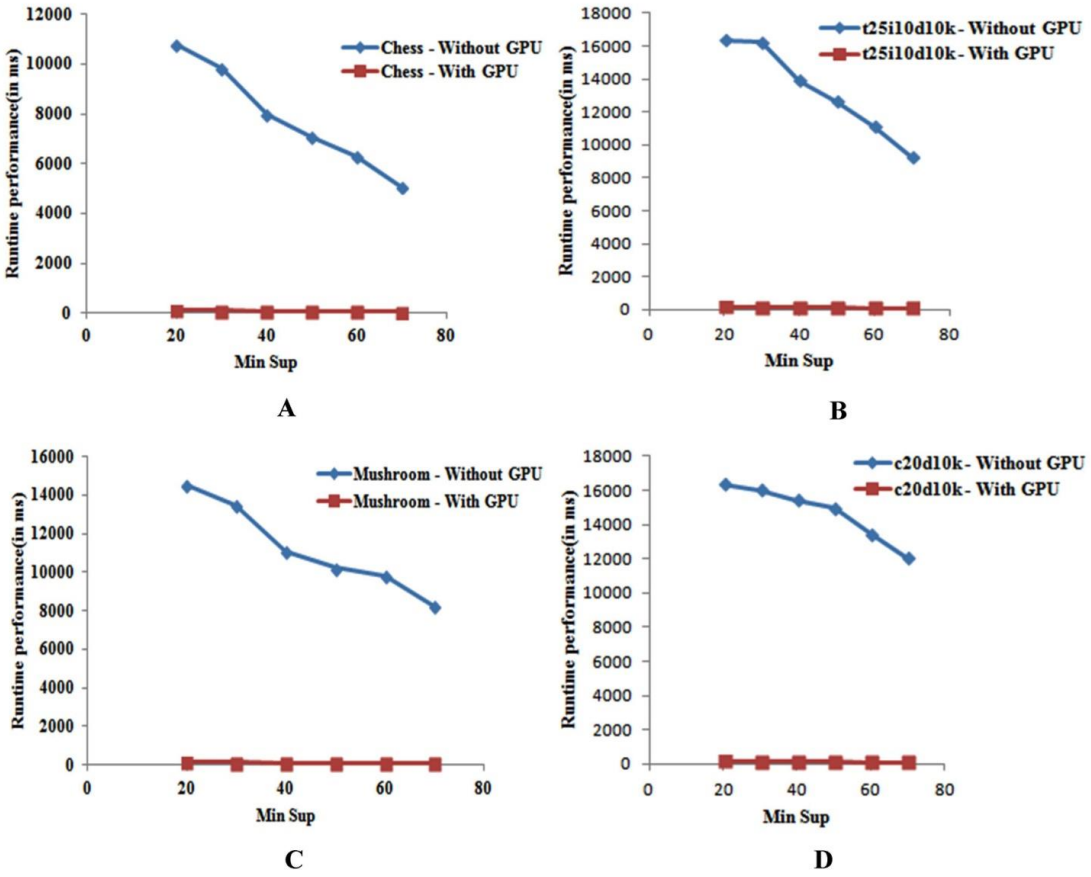


Fig.5. Runtime performance of the proposed method with and without GPU acceleration of each dataset

5. Conclusion

A GPU-accelerated novel method for finding the frequent itemset called GNVDF has been proposed in this research article. It uses an innovative approach to discover the candidate and frequent itemsets by removing unnecessary itemsets to form the subsequent itemsets. It also utilizes GPU for speeding up the process. It also empowers the use of a jagged array storage structure and removes the common elements in 1-frequent itemsets. With GPU-acceleration and innovative way of determining itemsets, the time required is significantly decreased. Similarly, with a jagged storage structure, the memory requirement is also minimized than the classical algorithms. From the extensive experiments made, it is observed that the GNVDF with GPU is 90-135 times faster than with GNVDF without GPU and also proved that it suits both sparse and dense datasets. Further, the use of the VDF approach restricts the database scan to one.

References

[1] H. Hamidi and A. Daraei, "Analysis of Pre-processing and Post-processing Methods and Using Data Mining to Diagnose Heart Diseases," *International Journal of Engineering (IJE), TRANSACTIONS A: Basics*, vol. 29, no. 7, pp. 921-930, 2016.

[2] J. Han, J. Pei and M. Kamber, Data mining: concepts and techniques, Morgan Kaufmann Publishers, 2011.

[3] H. Lisnawati and A. Sinaga, "Data Mining with Associated Methods to Predict Consumer Purchasing Patterns", *International Journal of Modern Education and Computer Science(IJMECS)*, vol. 12, no. 5, pp. 16-28, 2020.

[4] A. Sinha, B. Sahoo, S.S.Rautaray and M. Pandey, "An Optimized Model for Breast Cancer Prediction Using Frequent Itemsets Mining", *International Journal of Information Engineering and Electronic Business(IJIEEB)*, vol.11, no.5, pp. 11-18, 2019.

[5] L. Vu and G. Alaghband, "A self-adaptive method for frequent pattern mining using a CPU-GPU hybrid model," *in Proceedings of the Symposium on High Performance Computing*, 2015.

[6] D. Albert, K. William, Fayaz and D. Veerabhadra Babu, "Exploiting Parallel Processing Power of GPU for High Speed Frequent Pattern Mining", *International Journal of Computer Engineering and Applications*, vol. 7, no. 2, pp. 71 - 81, 2014.

[7] W. Fang, M. Lu, X. Xiao, B. He and Q. Luo, "Frequent itemset mining on graphics processors," *in Proceedings of International Conference on Network and Parallel Computing*, 2009.

[8] S. M. Fakhrahmad and G. Dastghaibyfard, "An Efficient Frequent Pattern Mining Method and its Parallelization in Transactional Databases," *Journal of Information Science and Engineering*, vol. 27, no. 2, pp. 511-525, 2011.

[9] J. Zhou, K. M. Yu and B. C. Wu, "Parallel frequent patterns mining algorithm on GPU", *in Proceedings of International Conference on Systems*, 2010.

[10] D. William Albert, K. Fayaz and D. Veerabhadra Babu, "HSApriori: high speed association rule mining using apriori based algorithm for GPU," *International Journal of Multidisciplinary and Current Research*, vol. 2, pp. 759-763, 2014.

[11] M. Tiwary, A. K. Sahoo and R. Misra, "Efficient implementation of apriori algorithm on HDFS using GPU," *in Proceedings of International Conference on High Performance Computing and Applications*, 2014.

[12] J. Li, F. Sun, X. Hu and W. Wei, "A multi-GPU implementation of apriori algorithm for mining association rules in medical data," *ICIC Express Letters*, vol. 9, no. 5, pp. 1303-1310, 2015

[13] L. Vu and G. Alaghband, "A self-adaptive method for frequent pattern mining using a CPU-GPU hybrid model," *in Proceedings of the Symposium on High Performance Computing*, 2015.

[14] Y. Li, J. Xu, Y. H. Yuan and L. Chen, "A new closed frequent itemset mining algorithm based on GPU and improved vertical structure," *Concurrency and Computation Practice and Experience*, vol. 29, no. 06, pp. 1-12, 2016.

[15] K.W. Chon, S. H. Hwang and M. S. Kim, "GMiner: A fast gpu-based frequent itemset mining method for large-scale data," *Information Sciences*, vol. 439-440, pp.19-38, 2018.

[16] Y. Wang, T. Xu, S. Xue and Y. Shen, "D2P-Apriori: A deep parallel frequent itemset mining algorithm with dynamic queue," *in Proceedings of 10th International Conference on Advanced Computational Intelligence*, 2018.

[17] Y. Djenouri, D. Djenouri, A. Belhadi and A. Cano, "Exploiting GPU and cluster parallelism in single scan frequent itemset mining," *Information Sciences*, vol. 496, pp. 363-377, 2019.

[18] P. Sumathi, and S. Murugan, A Memory Efficient Implementation of Frequent Itemset Mining with Vertical Data Format Approach, International Journal of Computer Sciences and Engineering. 6(2018) 152-157.

[19] W. Gan, J. C. Lin, P. Fournier-Viger, H. C. Chao and P. S. Yu, "Survey of parallel sequential pattern mining," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 13, no. 3, pp. 1-34, 2019.

[20] Y. M. Guo and Z. J. Wang, "A vertical format algorithm for mining frequent item sets," *in Proceedings of 2nd International Conference on Advanced Computer Control*, 2010.

[21] E. Hashemzadeh and H. Hamidi, "Using a Data Mining Tool and FP-growth Algorithm Application for Extraction of the Rules in Two Different Dataset," *International Journal of Engineering (IJE), TRANSACTIONS C: Aspects*, vol. 29, no. 6, pp. 788-796, 2016.

[22] M. Samoliya and A. Tiwari, "On the Use of Rough Set Theory for Mining Periodic Frequent Patterns", *International Journal of Information Technology and Computer Science (IJITCS)*, vol.8, no.7, pp.53-60, 2016.

[23] P. Prithiviraj and R. Porkodi, "A comparative analysis of association rule mining algorithms in data mining: a study," *American Journal of Computer Science and Engineering Survey*, vol. 3, pp. 98-119, 2015.

[24] F. Wang, J. Dong and B. Yuan, "Graph-based substructure pattern mining using cuda dynamic parallelism," *in Proceedings of International conference on intelligent data engineering and automated learning*, 2013.

[25] B. De Alwis, S. Malinga, K. Pradeeban, D. Weerasiri and S. Perera, "Horizontal format data mining with extended bitmaps," *in International Conference of Soft Computing and Pattern Recognition*,2011.

[26] P. Suresh, K. N. Nithya and K. Murugan, "Improved Generation of Frequent Item Sets using Apriori Algorithm," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 4, no. 10, pp. 25-27, 2015.

[27] A.Subashini and M. Karthikeyan, "Itemset Mining using Horizontal and Vertical Data Format," *International Journal for Research in Engineering Application & Management*, vol. 05, no.03, pp. 534-539, 2019.

Authors' Profiles

P.Sumathi received her B.Sc and M.Sc degrees in Computer Science from Seethalakshmi Ramaswami College, affiliated to Bharathidasan University, Tiruchirappalli, India in 2001 and 2003 respectively. She received her M.Phil degree in Computer Science in 2008 from Bharathidasan University. She is presently working as an Assistant Professor in the Department of Computer Science, Vysya College, Salem. She is currently pursuing a Ph.D. degree in Computer Science at Bharathidasan University. Her research interests include Data Mining, Data structures and Database concepts.

Dr.S.Murugan received his M.Sc degree in Applied Mathematics from Anna University in 1984 and M.Phil degree in Computer Science from Regional Engineering College, Tiruchirappalli in 1994. He is an Associate Professor in the Department of Computer Science, Nehru Memorial College (Autonomous), affiliated to Bharathidasan University since 1986. He has 32 years of teaching experience in the field of Computer Science. He has completed his Ph.D. degree in Computer Science with a specialization in Data Mining from Bharathiyar University in 2015. His research interest includes Data and Web Mining. He has published many research articles in reputed National and International journals.

How to cite this paper: P. Sumathi, S.Murugan, " GNVDF: A GPU-accelerated Novel Algorithm for Finding Frequent Patterns Using Vertical Data Format Approach and Jagged Array ", International Journal of Modern Education and Computer Science(IJMECS), Vol.13, No.4, pp. 28-41, 2021.DOI: 10.5815/ijmecs.2021.04.03

