2016

Mining Frequent Sequential Patterns From Multiple Databases Using Transaction Ids

Vignesh Aravindan
University of Windsor

Mining Frequent Sequential Patterns From Multiple Databases Using Transaction Ids

By

Vignesh Aravindan

A Thesis

Submitted to the Faculty of Graduate Studies through the School of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2016

Mining Frequent Sequential Patterns From Multiple Databases Using Transaction ids

By

Vignesh Aravindan

APPROVED BY:

Dr. Animesh Sarker, External Reader

Department of Mathematics & Statistics

Dr. Boufama Boubakeur, Internal Reader

School of Computer Science

Dr. Christie I. Ezeife, Advisor

School of Computer Science

12/12/2016

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication. I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Mining frequent sequential patterns from multiple databases to discover more complex patterns from multiple data sources such as multiple E-Commerce (B2C) web sites for comparative, historical and derived analysis, poses the additional challenge of integrating mined patterns from multiple sources during various levels of mining. A few existing work on mining frequent patterns from multiple databases (MDB's) are the ApproxMap algorithm and the TidFP algorithm. The ApproxMap algorithm breaks its input sequences (e.g., the 2-column sequence <(123)(45)>) into columns so it can find the collection of approximate frequent sequences of all the columns as the approximate sequence of the database. The same method is used to integrate frequent sequences from each MDB that must have the same table structure. The TidFP algorithm mines frequent item_sets from multiple sources of different table structures and related through foreign key attributes using transaction ids for integrating patterns through set operations (e.g., intersect, union) in order to answer global queries involving multiple sources. The limitations of existing work on multiple database *sequential pattern mining* such as the ApproxMap algorithm is that they are not able to mine frequent sequences to answer exact and historical queries from MDB's of different structure; while the TidFp algorithm can only answer queries from MDB's on item_sets but not for sequences.

This thesis proposes the Transaction id frequent sequence pattern (TidFSeq) algorithm which uses the techniques of the TidFP algorithm for mining item sets on the problem of mining frequent sequences from diverse MDB's. The challenges with mining frequent sequences from MDBs that is solved by this thesis are that the TidFSeq algorithm first computes the element (ie. Sequence item position id) in which each item in each sequence (ie. sequence id) occurs by replacing the <frequent items, transaction id list> tuple used in the TidFp with a <frequent sequences, sequence column position id list> tuple. For every item 'i' in the kth sequence of n-sequence of length 'n', the TidFSeq algorithm first transforms it into a tuple that specifies (a) it's transaction id (Tid) and (b) the list of the kth sequence in this transaction that item 'i' occurs (called it's position id list). Next the GSP-like candidate generation approach is used on our transformed sequences to generate frequent sequences with transacion ids which can be used to answer complex queries from MDB's through set operations. The proposed TidFSeq algorithm, PrefixSpan algorithm and ApproxMap algorithm are compared with respect to the results obtained for a given query, processing speed and memory requirement. Experiments show that the proposed TidFSeq algorithm mines the exact frequent sequences (ie. 100% accuracy) from multiple sequence tables, when compared to the ApproxMap algorithm that has an accuracy of 79%. The TidFSeq algorithm has faster processing time for mining frequent sequences from multiple tables than the PrefixSpan and ApproxMap algorithms.

Keywords: Sequence database, Multiple Databases, Frequent Patterns, Candidate generation, Frequent itemsets, Frequent sequences, Data mining, Association rule mining, Support, Global database.

DEDICATION

This thesis is dedicated to my father Aravindan Somasundaram, Mother Suba Aravindan and my brother Vishnuvardhan Aravindan. Without their patience, understanding, support, and most of all love, the completion of this work would not have been possible.

ACKNOWLEDGEMENT

I would like to give my sincere appreciation to all of the people who have helped me throughout my education. I express my heartfelt gratitude to my parents and brother for their support throughout my graduate studies.

I am very grateful to my supervisor, Dr. Christie Ezeife for her continuous support throughout my graduate study. She always guided me and encouraged me throughout the process of this research work, taking time to read all my thesis updates and for providing financial support through research assistantship.

I would also like to thank my external reader, Dr.Animesh Sarker, my internal reader, Dr.Boufama Boubakeur, and my thesis committee chair, Dr. Ngom for making time to be in my thesis committee, reading the thesis and providing valuable input. I appreciate all your valuable suggestions and the time, which have helped improve the quality of this thesis.

Finally, I would express my appreciations to all my friends and colleagues at University of Windsor, especially Bindu Peravali for her support and encouragement. Thank you all

TABLE OF CONTENTS

DECLARATION OF ORIGINALITY iii

ABSTRACT iv

DEDICATION v

ACKNOWLEDGEMENT vi

TABLE OF FIGURES.....ix

TABLE OF TABLES.....xi

CHAPTER 1: INTRODUCTION 1

1.1 Data Mining Techniques 6

1.2 Existing work on mining multiple databases 12

1.2.1 Mining frequent patterns from multiple tables having same structure 15

1.2.2 Mining frequent patterns from multiple tables having different structure 16

***1.4 Thesis Contributions*:** 18

1.5 Outline Of Thesis: 23

CHAPTER 2: RELATED WORK 24

2.1 Frequent itemset mining algorithms 24

2.1.1 Apriori algorithm 24

2.1.2 FP-tree Algorithm 26

2.2 Frequent Sequence mining algorithms 31

2.2.1 GSP Algorithm 31

2.2.2 SPAM algorithm 34

2.2.3 PrefixSpan algorithm 39

2.2.4 PLWAP algorithm 42

2.3 Mining frequent patterns from multiple databases 45

2.3.1 ApproxMap algorithm 45

2.3.2 TidFp Algorithm 49

2.3.3 Kernel estimation to identify valuable customers 52

2.3.4 Mining Stable patterns from multiple correlated databases 55

2.3.5 Identifying relevant databases 58

2.3.6 Mining multiple databases using pipeline feedback technique 62

2.3.7 Clustering local frequent items in multiple databases 63
CHAPTER 3 : PROPOSED TIDFSEQ ALGORITHM 65
3.1 Problems Addressed 68
3.2 Definitions used in proposed algorithm 68
3.3 The TidFseq Algorithm 69
3.4 Example Application of TidFSeq Algorithm 75
3.5 Flowchart for TidFSeq algorithm 83
CHAPTER 4: EVALUATION OF TIDFSEQ ALGORITHM 87
4.1 Comparison of algorithms over the query handled 87
4.2 Comparison of execution speed (speed of processing) of the algorithms 90
4.2.1 Execution times (in secs) for different datasets of different sizes at MinSupport of 40% 90
4.2.2 Execution times (in secs) for small-sized dataset (250K) for different support counts 91
4.2.3 Execution times (in secs) for Medium-sized dataset (500K) for different support counts 92
4.2.4 Execution times for large-sized dataset (2M) for different support count. 93
4.2.5 Memory Usages (in terms of MB) for Different Data Sizes at Minsupport of 40% 93
4.3 Comparison of algorithms based on Acuracy of the frequent sequences obtained for datasets of different sizes 95
4.3.1 Acuracy of the frequent sequences obtained for datasets conatining 250K, 500K, 750K sequences at Minsupport of 40%. *95*
4.4 Comparison of algorithms based on execution speed for (250K) dataset having longer sequences at Minsupport of 20% 95
CHAPTER 5: CONCLUSION AND FUTURE WORK 97
5.1 Future Work 97
REFERENCES 98
VIVA AUCTORIS 101

TABLE OF FIGURES

Figure 1: Classification dataset……07

Figure 2: Decision tree……...……..……..08

Figure 3: Clustering Dataset…….…….…..…….09

Figure 4: Initial clusters…...…….…….…..…….09

Figure 5: Grouping data into clusters……...…….…….…10

Figure 6: Local branch sequence databases having same structure ……..…….13

Figure 7: Attribute relationship for local branch sequence databases having same structure..13

Figure 8: Attribute relationship for sequence databases having different structure...…..……14

Figure 9: Constucrtion Fp-tree…….……..……..……..29

Figure 10: Final frequent sequences…….…..…34

Figure 11: Vertical Bitmap of items in transaction db…….36

Figure 12: Lexicographic tree…….…....37

Figure 13: S-step……..……..…...38

Figure 14: I-step……38

Figure 15: PrefixSpan algorithm……..……41

Figure 16: Construction of PLWAP tree……..……...44

Figure 17: Support Counts of Candidate Items in Each Sequence Column……47

Figure 18: Local databases…….....53

Figure 19: Database 1……59

Figure 20: Database 2……....59

Figure 21: RF values of Db1…….…61

Figure 22: RF values of Db2…….…61

Figure 23: Multiple databases (Db1, Db2)…….……...62

Figure 24: local db1 and local db2……..…… 62

Figure 25: TidFSeq Algorithm…….……..70

Figure 26: I-step pruning ()…….…….……..72

Figure 27: S-step pruning ()…….…….…….74

Figure 28 (a)(b): Final output for TidFSeq algorithm…….……..……..82

Figure 29: Proposed system flowdiagram…….……84

Figure 30: Input 1…….……..…….87

Figure 31: Input 2……..……..87

Figure 32: Execution times for different datasets of different sizes at MinSupport of 40%.....90

Figure 33: Execution times (in secs) for small-sized dataset (250K) for different support counts..91

Figure 34: Execution times (in secs) for Medium-sized dataset (500K) for different support counts…….……...92

Figure 35: Execution times (in secs) for large-sized dataset (2M) for different support counts……93

Figure 36: Memory Usages for Different Data Sizes at Minsupport of 40%.....94

TABLE OF TABLES

Table 1: Transaction table……. 11

Table 2: Sequence Table…… 12

Table 3: Customer/sequence of items purchased……14

Table 4: Discounts / sequence of customers who avail it……14

Table 5: Summary of existing systems in multiple database mining.....19

Table 6: Transaction db……...25

Table 7: Transaction db……..28

Table 8: Transaction db sorted according to descending order of support counts……...28

Table 9: Final result……...……30

Table 10: GSP sequence table……..……..32

Table 11: Frequent-1 items……...32

Table 12: Candidate sequences gnerated using GSP-gen join…….……...33

Table 13: Spam transaction db……..……..…….35

Table 14: Web access sequence database……..……..……..43

Table 15: Windsor branch sequence table……..……..……....46

Table 16: Hamilton branch sequence table……..……..……....46

Table 17: Preprocessed Windsor table…….……..……..46

Table 18: Preprocessed Hamilton table……..……..47

Table 19: Finding global approximate patterns……...48

Table 20: Transaction Table Drug/Side-effect……..……...50

Table 21: Transaction Table Patient/Drug……..…….....50

Table 22: Transaction table 1……..…….……..55

Table 23: Transaction table 2…….……56

Table 24: Drug/side-effects……...…….……76

Table 25: Patient/Drugs ……..…..……..…...76

Table 26: Transformed 1-candidate item sequences for MDB_1…….……77

Table 27: Transformed 1-candidate item sequences for MDB_2……..……....77

Table 28: Comparison of query handling done by TidFSeq, ApproxMap and PrefixSpan algorithms…….……..……..89

Table 29: Execution times (in secs) for different datasets of different sizes at MinSupport of 40%.....90

Table 30: Execution times (in secs) for small-sized dataset (250K) for different support counts……..…….…..91

Table 31: Execution times (in secs) for Medium-sized dataset (500K) for different support counts……..……...92

Table 32: Execution times for large-sized dataset (2M) for different support counts……..……....93

Table 33: Memory Usages for Different Data Sizes at Minsupport of 40%.....94

Table 34: Acuracy of the frequent sequences obtained for datasets conatining 250K, 500K, 750K sequences at Minsupport of 40%.....95

Table 35: Comparison of algorithms based on execution speed for (250K) dataset having longer sequences at Minsupport of 20%.....96

CHAPTER 1: INTRODUCTION

Data mining helps end users extract useful business information from large databases. Data mining is knowledge mining from data (Han and *et al*. (2012)). Mining frequent sequential patterns from single databases (for eg, frequent sequence <ac (abc)>, meaning items a, c and itemset (abc) are purchased separately at different times by a customer frequently) has many application areas such as finding frequent sequence of products purchased by customers in a customer- purchase sequence database of a retail store, finding the set of frequent gene sequences in a biologocal sequence dataset, finding the frequent sequences of words/symbols that appear in a textual/sign-language document. Mining frequent sequences from multiple sequence databases helps to discover more complex patterns, for example, in an application having related sequence databases such as Patient/Drugs and Drugs/side-effects sequence databases, the mined frequent sequences from the related sequence tables can help answer queries such as, find the frequent sequence of side effects that patients p1 and p3 suffer from. However, mining multiple databases poses the additional challenge of integrating mined patterns from multiple sources during various levels of mining to answer user queries (for eg, by using set operations like intersect and union, the mined patterns from the related sequence tables, Drug/side-effects sequence table and Patient/Drug sequence table, can help answer queries such as, find the frequent sequence of side effects that patients p1 and p3 suffer from). There has been a lot of existing work on mining frequent itemsets/sequences from single databases, existing work on mining frequent itemsets from transaction tables, can be classified into Apriori based (such as Apriori algorithm (Agrawal and Srikant (1994))) and non-apriori based algorithms (such as Fp-tree algorithm (Pei and *et al*. (2001)), H-Mine algorithm by Pei and *et al.* (2001)). Some prominent frequent sequence mining algorithms on single database that are Apriori based are: GSP (Srikant and Agrawal (1996)), SPADE (Zaki (2001)) and frequent sequence mining algorithms that are non-Apriori based are: SPAM (Ayres and *et al*.(2002)), Prefix-span (Pei and *et al*.(2004)). Algorithms specifically for mining web sequential patterns include WAP-tree algorithm (Pei et al. 2000), PLWAP-tree algorithm (Ezeife and Lu 2005; Ezeife et al. 2005; Lu and Ezeife 2003) and FS-Miner algorithm (El-Sayed et al. 2004).

But these single sequence/itemset database mining algorithms cannot mine frequent patterns from multiple databases such as Drug/side-effects sequence table and Patient/Drug sequence table and integrate the results such that it can answer queries related to multiple databases. For example, for multiple related sequence tables, such as Drug/side-effects sequence table and Patient/Drug sequence table, a standard frequent sequence mining algorithm such as GSP (Srikant and Agrawal (1996)) will only mine the frequent sequence of side-effects from Drug/Side effects sequence table and mine the frequent sequence of drugs purchased from Patient/Drugs sequence table, the final set of frequent sequences obtained from each sequence table cannot be used to answer queries such as, find the frequent sequence of side effects that patients p1 and p3 suffer from. Some of the reasons for the need to mine frequent patterns from multiple databases and the example queries for each category is listed below:

1) ***Comparative analysis:*** In applications such as E-Commerce sites, the information of products (such as product name, price) sold by each online store (such as Bestbuy) are constantly stored in multiple databases. For example, the laptop products sold in Bestbuy and Walmart are stored in two product databases corresponding to Bestbuy and Walmart respectively. By mining such multiple databases, apart from answering historical queries, comparative queries can also be answered, such as:
    - Find the E-commerce websites, from the product information tables of all the E-commerce websites that sell 'Apple' products with discount.
    - Find the set of frequently bought laptop product names from the product information tables corresponding to the E-commerce websites that provide highest discounts.
    - Find the E-commerce website which sells the cheapest 'samsung' TV products.
2) ***Frequent local and global product pattern analysis***: There is a need to find the frequent local and global patterns of product purchased from customer transaction databases of retail stores having same structure (ie. local customer transaction databases of a retail store such as Walmart having customer id and the products purchased by the customer in each of its local database). Queries useful for the retail

store (such as Walmart having many local branches) can be answered using the local and global frequent patterns of prodcut purchased, such as:

- Find the global frequent patterns that consist of milk products in them.
- Find the frequent local product patterns that are also frequent on global scale.
- Find the frequent pattern of products purchased in branches located in the western region of US.

3) ***Mining frequent patterns from multiple tables with different structures:*** Not all multiple tables have the same structure, there is also a need to mine frequent itemsets/sequences from multiple related databases having different structure (ie. Patient/Drugs and Drugs/side-effects that are related by the foreign/primary key attribute: Drugs) to answer queries related to the multiple tables. For example, in an application having related sequence databases such as Patient/Drugs and Drugs/side-effects sequence databases, the mined patterns from the related sequence tables can help answer queries such as:

- Find the frequent sequence of side effects that patients p1 and p3 suffer from.
- Find all the patients that are affected by frequent sequence of side effects that have side effect 's1' in their pattern.

4) ***Mining alternate types of information:*** Patterns such as stable patterns, finding important customers in a retail store are alternate types of information that help find useful information such as the set of standard and stable products found in all the local branches of a retail store like walmart. Example query can be answered by mining such alternate information such as:

- Find the set of stable patterns from all the local branches.
- Find the important customers that come to the retail store
- Find the local frequent products that are similar to each other.

Mining frequent patterns from multiple databases is of two types:

1) ***Mining frequent patterns from multiple tables having same structure*** (for eg, local customer transaction databases of a retail store such as Walmart having

customer id and the products purchased by the customer in each of its local database)

2) ***Mining frequent patterns from multiple tables having different structure*** (for eg, Patient/Drugs and Drugs/side-effects that are related by the foreign/primary key attribute: Drugs).

Some existing systems that mine frequent patterns from multiple databases having same structure are:

1) ApproxMAP algorithm by Hyue and *et al*. (2006), the ApproxMap algorithm breaks its input sequences (e.g., the 2-column sequence < (123) (45) >) into columns so it can find the collection of approximate frequent sequences of all the columns as the approximate sequence of the database. The same method is applied on the local frequent sequences obtained from each local cutomer transaction database that must have the same table structure, to get the frequent global approximate sequence patterns. The main limitation of this algorithm is that it does not generate exact frequent sequence patterns and does not work for multiple tables with different structure.
2) The hierarchial Gray clustering algorithm by Yaojin and et al (2013), introduced a new concept of stable patterns. It defines an item 'a' as stable, if the item satisfies the minimum support count 's' in each of the local transaction table (T1,T2..TN, where Ti, 1<=i<=N, is a local transaction table) that it occurs and also the variation of the support count of that item 'a' is less than or equal to a user defined variation value 'v'. The algorithm clusters stable items found from multiple transaction tables into stable patterns according to the similarity of their timestamps (ie. the time at which the stable item 'a' was purchased by a cutomer). The use of stable patterns helps us to identify the standard set of constantly purchased items on a global scale in a retail store having local branches. Since this algorithm is specific to mining stable patterns, it is not useful to mine frequent sequences from multiple tables with different structures.
3) The kernel estimation algorithm by Zhang and *et al* (2009), introduced the concept of generating the local and global important cutomers (ie. customers who regularly

purchase high value products from the store) from a retail store having local branches. The local databases contain cutomer information, such as the expenditure of the customer in that local branch and the number of visits and transactions made by the customer in that local branch. Using this information the kernel estimation algorithm uses a mathematic approach to first find the local importance of each customer to the local branch. And then the global importance of the customer to the retail store. Since this algorithm is specific to mining local and global important customers, it is not useful to mine frequent sequences from multiple tables with different structures.

4) The pipeline feedback technique by Adhikari.A et al (2010) aims to give an efficient methodology to mining large sets of local databases. The main concept is to sort the databases based on decreasing order of their sizes and then mine each database using a standard frequent itemset mining algorithm such as FP tree algorithm (Han et al (2001)) and finally synthesize the large collection of patterns obtained. This technique was proposed to make the process of mining frequent itemsets from large sets of multiple transaction databases having same structure faster and is not useful to mine frequent patterns from multiple related databases.
5) Clustering local frequency items in multiple databases by Animesh.A (2013), gives a similarity measure that is derived from the common items/products in multiple databases (for example, milk product is common item in local transaction tables corresponding to the local branches of a retail store), to cluster the frequent itemsets obtained in each local database into groups based on similarity of the frequent itemsets (for example, frequent itemsets (Ice cream, Milk) and (Milkshake, cheese) belong to same group, since all are milk based products). This method is useful to find the groups of similar frequent items in each local branch of a retail store, but it is not designed to mine frequent sequences from multiple tables with different structures.

Some existing systems that mine frequent patterns from multiple databases having different structure are:

1) In some application areas where there is a very large set of databases and the user needs to find specific databases from the entire set of databases, in order to query specific information, there is a need to first find the specific databases that can be queried later, to find information that the user is looking for. For example, if the query is to find the eating habits of people belonging to Sri Lanka and we are given a huge dataset of 1000 databases, then we must first find the relavant databases that can be queried to find the answer. One prominent existing work to solve this type of problem was given by Liu, Lu and Yao (2001) (belongs to category 2). They introduced a mathematical factor known as relevance factor, which identifies how close the attribute in a database is to the given query. For example, if the query is find the eating habits of people belonging to Sri Lanka and the database being checked has attributes 'habits' and 'Country', the relevance factor will be high for that database (close to 1), meaning this database can be chosen to be queried in order to find the answer for the input query. Since this paper is focused on finding the relevant databases that needs to be mined rather than mining the frequent patterns from the multiple databases, it cannot mine frequent sequences from multiple tables with different structures.
2) TidFP algorithm by Ezeife and Zhang (2009).The TidFp algorithm mines frequent item_sets from multiple sources using transaction ids for integrating patterns through set operations (e.g., intersect, union) in order to answer global queries involving multiple sources. But the main limitation is that it only mines frequent itemsets from multiple transaction tables and not frequent sequences.

It is clear that none of the existing systems are able to mine multiple related sequence tables having different structure for frequent sequences and answer queries using the result sets. The main purpose of this thesis is to develop a multiple related sequence database mining algorithm to mine the frequent sequence patterns and answer valuable user queries.

1.1 Data Mining Techniques

In this section, the three main techniques of data mining are discussed. Data mining uses three techniques, Classification, Clustering and Association rule mining.

Classification:

The goal of classification is to accurately predict the target class for each case in the data. For example, a classification model could be used to identify loan applicants as low, medium, or high credit risks. Classification involves a training data and a test data. In the training data, the target class is known and in the test data the target class in not known. A classification algorithm finds the relationship between the target class and the predictors (ie. the rest of the attributes in the dataset) to find the value of the target class. This relationship is known as a classification model. Some prominent classification algorithms are Naive Bayes algorithm (Russell and Norvig (1995)), k-nearset neighbour algorithm (Altman (1992)). The training dataset is used to find the best classification algorithm that provides the best classification model to predict the known target class values with good accuracy. Then the same classification model is applied on the test data having unknown target class values, to check for its prediction accuracy. Figure 1 shows a classification dataset. The goal is to predict which of the customer ids (case id) are likely to increase spending if given an affinity card (target class attribute). The target class attribute affinity_card has two values '0' (customer does not increase spending) and '1' (customer does increase spending). The dataset is split into training dataset and test dataset. An efficient classification model (ie. the relationship between the predicates (cust_gender,education,occupation,age) and the affinity card (target class attribute)) is found for the training set and the same model is applied on the test set to predict the values for the target class attribute in test set with efficient accuracy.

Cust_id	gender	Age	occupation	education	Affinity_card
1	M	20	Clerk	Bach	1
2	M	21	IT	Masters	1
3	F	33	IT	Bach	0
4	F	22	Doctor	Bach	0

Figure 1: Classification dataset

For the training dataset shown in Figure 1, a decision tree classification algorithm (Rokach and Maimon (2008)) is used to get a decision tree classification model shown in Figure 2, for the dataset. The decision tree model, gives the rules to predict the values of the target class attribute. For example, in Figure 2, when attribute education of an incoming record

is either ‘Bachelors’ or ‘Masters’ and the gender of the customer is Male, the value of target class attribute (ie. affinity_card) is 1, else if the conditions are not met, value is ‘0’.

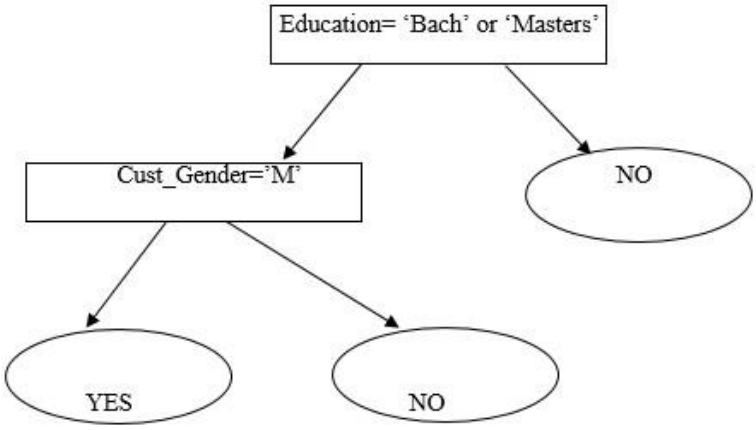

Figure 2: Decision tree

Such a classification model is learned from historical data and the rules from it are used to predict the affinity_card of incoming customers.

Clustering:

Clustering is a task of grouping a set of objects in a way such that objects of the same group are similar to each other than to objects of other groups. Clustering is also known as unsupervised classification, since clustering involves grouping together instances of unlabeled data. For example, clustering a dataset of customer locations in a city into groups based on the similarity of the data values, were all the customers belonging to a group or cluster belong to same locality in the city. Some prominent clustering algorithms are K-means clustering algorithm (MacQueen (1967)), BIRCH (Zhang (1996)). As a simple illustration of a k-means algorithm, consider the following data set consisting of the scores of two variables A and B on each of seven subjects, in Figure 3.

Subject	A	B
1	1.0	1.0
2	1.5	2.0
3	3.0	4.0
4	5.0	7.0
5	3.5	5.0
6	4.5	5.0
7	3.5	4.5

Figure 3: Clustering Dataset

The first step is to find the initial clusters. The initial clusters are taken as minimum (A, B) pair values and maximum (A, B) pair values from Figure 3. (ie. (1.0, 1.0) and (5.0, 7.0)) in our example. In our example, the initial clusters will be (1.0, 1.0) and (5.0, 7.0), belonging to individuals 1 and 4 respectively, which is shown in (Figure 4):

	Individual	Mean Vector(Centroid)
Group 1	1	(1.0,1.0)
Group 2	4	(5.0,7.0)

Figure 4: Initial clusters

The mean vector is the mean of A and B values of all the individuals in the cluster. Since initially only one indivudual is in the cluster, the mean vectors of the clusters (group 1 and group 2) will be (A, B) values of the individuals 1 and 4 in group 1 and group 2 respectively. The mean vector is recalculated each time a new member is added. The remaining individuals are now examined in sequence and allocated to the cluster to which they are closest, in terms of Euclidean distance to the cluster mean (Mean vector of the cluster).

The formula for eucladiean distance between (A, B) value of each indivdual and (X, Y) value of the mean vector of the cluster is:

sqrt ((B-A)^2 + (Y-X)^2). This leads to a series of steps (shown in Figure 5), until every individual is assigned to a cluster that has the most similarity to the individual.

	Cluster 1		Cluster 2	
Step	individual	Mean Vector(centroid)	Individual	Mean Vector(centroid)
1	1	(1.0,1.0)	4	(5.0,7.0)
2	1,2	(1.2,1.5)	4	(5.0,7.0)
3	1,2,3	(1.8,2.3)	4	(5.0,7.0)
4	1,2,3	(1.8,2.3)	4,5	(4.2,6.0)
5	1,2,3	(1.8,2.3)	4,5,6	(4.3,5.7)
6	1,2,3	(1.8,2.3)	4,5,6,7	(4.1,5.4)

Figure 5: Grouping data into Clusters

Individuals 1, 2, 3 belong to cluster 1 and 4, 5, 6, 7 belong to cluster 2. Hence, from the example we can see that, items which are similar to each other are clustered together by using clustering algorithms such as K-means clustering algorithm.

Association Rule mining:

Association rule: Association rule is of the form X->Y, X and Y belongs to candidate set I, where I= {i1, i2....in} is a set of n items belonging to a retail store. In the rule X->Y, the itemset on lefthand side (X) of the rule is called antecedent of the rule and itemset on righthand side (Y) of the rule is called consequent. Association rule finds the relationship between the items in the rule. For example, Bread->Milk implies that if product Bread is bought customers also buy product Milk.

TID	Items purchased
10	Bread,Milk
20	Bread,Diaper,Beer,eggs
30	Milk,Diaper,Beer,coke
40	Bread,Milk,Diaper,Coke

Table 1: Transaction table

Table 1 shows a customer transaction table of a retail store, where there are 4 transaction ids (10, 20, 30, 40) and their corresponding itemsets (items that are purchased together). For example, in TID 10, itemset (Bread, Milk) consists of items Bread and milk that are purchased together. Support of an itemset (i) = (number of occurences of i) / (number of database transactions). In example database Table 1 support of item Bread (i.e.) supp (Bread) = 3/4, since it occurs in three out of a total of four transactions. Confidence of an itemset (i) = (number of occurences of i) / (number of occurence of the antecedent of the itemset). For example, in Table 1, it can be seen that itemset (Bread, Milk) appears '2' times and antecedent item -Bread appears '3' times. Thus, confidence of (Bread->Milk) = 2/3. Hence, given a set of transactions T, the goal of association rule mining is to first find all the frequent itemsets having support greater than or equal to a user defined minimum support count. From the computed frequent itemsets, only strong association rules with support greater than or equal to a given minimum confidence threshold are retained. For example, if the minimum support threshold is 0.5 and minimum confidence threshold is 0.5. The association rule, Bread->Milk is a strong association rule in Table 1, since Bread->Milk has a support of 0.5, which is greater than minimum support threshold and confidence of 0.6, which is greater than minimum confidence threshold. Table 1 shows a transaction table from which frequent itemsets can be mined using algorithms such as the Apriori algorithm (Agrawal and Srikant (1994)) and Fp-tree algorithm (Pei and *et al*. (2001)). Table 2 shows an example of a customer purchase sequence table, having candidate items: a, b, c and d.

CID	Customer purchase sequences
1	<bcbd>
2	<ac(abc)>
3	<(abc)ac>
4	<abc>

Table 2: Sequence Table

In Table 2, each customer id (1, 2, 3, 4) is associated with a sequence of products/items purchased by the customer (cid), for example in Table 2, cid 2 consists of sequence <ac (abc)>, meaning items a, c and itemset (abc) are purchased separately at different times by the customer 'cid 2'. Using a sequence table, frequent sequences can be generated using algorithms such as: GSP (Srikant and Agrawal (1996)), SPADE (Zaki (2001)), SPAM (Ayres and *et al*.(2002)), Prefix-span (Pei and *et al*.(2004)), PLWAP-tree algorithm (Ezeife and Lu 2005).

1.2 Existing work on mining multiple databases

Existing systems in mining frequent patterns from multiple databases can be categorized into:

1) ***Mining frequent patterns from multiple tables having same structure:*** Local customer transaction databases of a retail store such as Walmart having customer id and the products purchased by the customer in each of its local database is an example of this type. An example of local transaction databases having same structure is shown in Figure 6 (it shows Customer_id/product_purchase_sequences, local sequence databases belonging to branches London and Hamilton). The Figure 7 shows the attribute relationship between the local databases. The primary key for both London database and Hamilton database is Customer_id attribute. The foreign key for both London database and Hamilton database is Product_purchase_sequences attribute.

Customer_id	Product_purchase_sequences
1	<(123) (1)>
2	<(123)>
3	<(3) (4)>
4	<(1) (3)>

Local branch: London db

Customer_id	Product_purchase_sequences
10	<(123)>
20	<(3) (4)>
30	<(1 2) (1) (3)>
40	<(1) (3)>

Local branch: Hamilton db

Figure 6: Local branch sequence databases having same structure

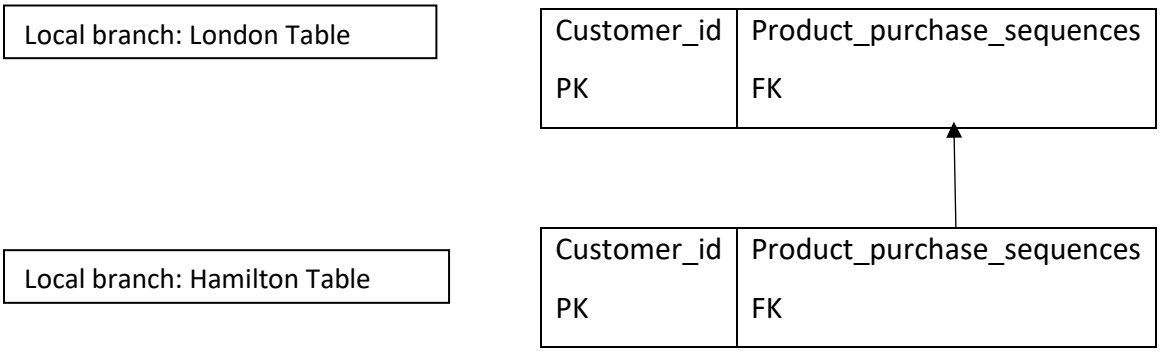

Figure 7: Attribute relationship for local branch sequence databases having same structure

2) ***Mining frequent patterns from multiple tables having different structure:*** Patient/Drugs and Drugs/side-effects that are related by the foreign/primary key attribute: Drugs, is an example of this type. For example, Table 3 (Customer/Sequence of items purchased by the customer) and Table 4 (Discounts/sequence of customers who use the discounts) are two sequence tables which are related through the customer attribute. Figure 8 shows attribute relationship between Table 3 and Table 4. In Table 3, the customer attribute is primary/foreign key and in Table 4, Discounts attribute is primary key and Customers attribute is foreign key.

Customers	Sequence of items
C1	<(1 2 3)>
C2	<(1) (3)>
C3	<(1 2 3)>
C4	<(2) (3)>

Table 3: Customer/sequence of items purchased

Discounts	Customers
D1	<(C1) (C4)>
D2	<(C1 C2 C3)>
D3	<(C1 C2 C3) (C3)>
D4	<(C2) (C3)>

Table 4: Discounts / sequence of customers who avail it

Table 5: Customer/sequence of items purchased

Customers PK/FK	Sequence of items

Table 6: Discounts / sequence of customers who avail it

Discounts PK	Customers FK

Figure 8: Attribute relationship for sequence databases having different structure

The following subsections describe the related systems that belong to the above two categories of mining multiple databases, their application area in that category and their limitations.

1.2.1 Mining frequent patterns from multiple tables having same structure

Mining the local customer transaction tables that are of the same structure (ie. having the same attributes customer_id and items_purchased in all the local customer transaction databases), will only give the frequent purchases made at each branch of a store. But mining the set of local frequent patterns (frequent itemsets or frequent sequences) will give us global frequent product purchase patterns, that help the store to assess the global impact of their products and find the most sought after products on a global scale. Some existing work related to this type of mining is ApproxMAP algorithm by Hyue and *et al*. (2006). The ApproxMap algorithm finds approximate frequent sequences from multiple customer purchase sequence databases, where each local database corresponds to the local branch of a retail store (eg. Windsor branch of Walmart). It breaks its input sequences (e.g., the 2-column sequence < (123) (45) >) into columns so it can find the collection of approximate frequent sequences of all the columns as the approximate sequence of the database. The same method is applied on the local frequent sequences obtained from each local cutomer transaction database that must have the same table structure, to get the frequent global approximate sequence patterns. The main limitation of this algorithm is that it does not generate exact frequent sequence patterns, that will be gnerated by a standard frequent sequence mining algorithm such as GSP (Srikant and Agrawal (1996)) for a single sequence table. And also the ApproxMap algorithm does not work for multiple tables with different structure.

The hierarchial Gray clustering algorithm by Yaojin and et al (2013), is an existing algorithm that introduced a new concept of stable patterns. The hierarchial Gray clustering algorithm (belongs to category 1) defines an item ‘a’ as stable, if the item satisfies the minimum support count ‘s’ in each of the local transaction table (T1,T2..TN, where Ti, 1<=i<=N, is a local transaction table) that it occurs and also the variation of the support count of that item ‘a’ is less than or equal to a user defined variation value ‘v’. The algorithm clusters stable items found from multiple transaction tables into stable patterns according to the similarity of their timestamps (ie. the time at which the stable item ‘a’ was purchased by a cutomer). The use of stable patterns helps us to identify the standard set of constantly purchased items on a global scale in a retail store having local branches. Since

this algorithm is specific to mining stable patterns, it is not useful to mine frequent sequences from multiple tables with different structures.

The kernel estimation algorithm (belongs to category 1) by Zhang and *et al* (2009), introduced the concept of generating the local and global important cutomers (ie. customers who regularly purchase high value products from the store) to a retail store having local branches. This algorithm makes the assumption that a customer 'A' visting local branch X will also visit local branch Y. The local databases contain cutomer information, such as the expenditure of the customer in that local branch and the number of visits and transactions made by the customer in that local branch. Using this information the kernel estimation algorithm uses a mathematic approach to first find the local importance of each customer to the local branch. And then the global importance of the customer to the retail store. Since this algorithm is specific to mining local and global important customers, it is not useful to mine frequent sequences from multiple tables with different structures.

Clustering local frequency items in multiple databases by Animesh.A (2013), gives a similarity measure that is derived from the common items/products in multiple databases (for example, milk product is common item in local transaction tables corresponding to the local branches of a retail store), to cluster the frequent itemsets obtained in each local database into groups based on similarity of the frequent itemsets (for example, frequent itemsets (Ice cream, Milk) and (Milkshake, cheese) belong to same group, since all are milk based products). This method is useful to find the groups of similar frequent items in each local branch of a retail store, but it is not designed to mine frequent sequences from multiple tables with different structures.

1.2.2 Mining frequent patterns from multiple tables having different structure

In some application areas where there is a very large set of databases and the user needs to find specific databases from the entire set of databases, in order to query specific information, there is a need to first find the specific databases that can be queried later, to find information that the user is looking for. For example, if the query is to find the eating habits of people belonging to Sri Lanka and we are given a huge dataset of 1000 databases, then we must first find the relavant databases that can be queried to find the answer. One prominent existing work to solve this type of problem was given by Liu, Lu and Yao (2001)

(belongs to category 2). They introduced a mathematical factor known as relevance factor, which identifies how close the attribute in a database is to the given query. For example, if the query is find the eating habits of people belonging to Sri Lanka and the database being checked has attributes 'habits' and 'Country', the relevance factor will be high for that database (close to 1), meaning this database can be chosen to be queried in order to find the answer for the input query. Since this paper is focused on finding the relevant databases that needs to be mined rather than mining the frequent patterns from the multiple databases, it cannot mine frequent sequences from multiple tables with different structures.

Mining frequent itemsets and frequent sequences from multiple tables with different structure is a major application in the area of multi database mining. For example, Table A (Customer/Sequence of items purchased by the customer) and Table B (Discounts/sequence of customers who use the discounts) are two sequence tables which are related through the customer attribute. The candidate items in Table A are: 1, 2, 3. The candidate items in Table B are: C1, C2, C3, C4. A query that can be answered by mining frequent sequences from Table A and Table B is, what are the discounts associated with items 1 and 3? The TidFP algorithm by Ezeife and Zhang (2009), is an existing algorithm in the area of mining tables with different structures. The TidFp algorithm mines frequent item_sets from multiple sources using transaction ids for integrating patterns through set operations (e.g., intersect, union) in order to answer global queries involving multiple sources. But the main limitation is that it only mines frequent itemsets from multiple transaction tables and not frequent sequences. The proposed TidFSeq algorithm mines multiple related sequence tables having different structure and can help in answering queries (such as what are the discounts associated with items 1 and 3?) for Table A and Table B.

1.4 Thesis Contributions:

Mining frequent sequential patterns from single databases has many application areas such as finding frequent sequence of products purchased by customers in a customer- purchase sequence database of a retail store, finding the set of frequent gene sequences in a biologocal sequence dataset. There has been a lot of existing work on mining frequent sequences from single databases, such as GSP (Srikant and Agrawal (1996)) and SPADE (Zaki (2001)). But these single database sequence mining algorithms cannot mine frequent sequence patterns from multiple related sequence databases such as Drug/side-effects sequence table and Patient/Drug sequence table and integrate the results such that it can answer queries related to multiple databases. For example, for multiple related sequence tables, such as Drug/side-effects sequence table and Patient/Drug sequence table, a standard frequent sequence mining algorithm such as GSP (Srikant and Agrawal (1996)) will only mine the frequent sequence of side-effects from Drug/Side effects sequence table and mine the frequent sequence of drugs purchased from Patient/Drugs sequence table, the final set of frequent sequences obtained from each sequence table cannot be used to answer queries such as, find the frequent sequence of side effects that patients p1 and p3 suffer from. There are some existing systems in the area of mining frequent patterns from multiple databases (shown in Table 5), that are closely related to the proposed algorithm. Each system has its own specific use and limitation that makes it unable to mine multiple related sequence tables (such as Drug/side-effects sequence table and Patient/Drug sequence table) for frequent sequences and answer queries using the result sets.

Existing System	Technique used	It's application	Limitation
ApproxMap algorithm by Hyue and *et al*. (2006)	Breaks its input sequences into columns so it can find the collection of approximate frequent sequences of all the columns as the approximate sequence of the database. The same method is applied on the local frequent sequences obtained from each local cutomer transaction database that must have the same table structure, to get the frequent global approximate sequence patterns.	Give the global frequent product purchase patterns of a Retail store such as Walmart, that help the store to assess the global impact of their products and find the most sought after products on a global scale	Does not generate exact frequent sequence patterns and does not answer queries for multiple related sequence tables.
The hierarchial Gray clustering algorithm by Yaojin and et al (2013)	Introduces a new concept of stable patterns, it defines an item 'a' as stable, if the item satisfies the minimum support count 's' in each of the local transaction table that it occurs and also the variation of the support count of that item 'a' is less than or equal to a user defined variation value 'v'. The algorithm clusters stable items found from multiple transaction tables into stable patterns according to the similarity of their timestamps (ie. the time at which the stable item 'a' was purchased by a cutomer).	Helps us to identify the standard set of constantly purchased items on a global scale in a retail store having local branches. Example: Milk, Eggs, Cereals is a stable pattern, since it is bought by most people on a regular basis in all local branches of a retail store.	Since this algorithm is specific to mining stable patterns, it is not useful to mine frequent sequences from multiple related tables.
Clustering local frequency items in multiple databases by Animesh.A (2013)	Gives a similarity measure that is derived from the common items/products in multiple databases (for example, milk product is common item in local transaction tables corresponding to the local branches of a retail store), to cluster the frequent itemsets obtained in each local database into groups based on similarity of the frequent itemsets (for example, frequent itemsets (Ice cream, Milk) and (Milkshake, cheese) belong to same group, since all are milk based products).	This method is useful to find the groups of similar frequent items in each local branch of a retail store.	Algorithm is not designed to mine frequent sequences from multiple related tables.
The kernel estimation algorithm by Zhang and *et al* (2009)	Finds locally and globally important customers using kernel estimation that finds the customer lifetime value on local and global scale.	Finds customers who regularly purchase high value products from the store.	It is specific to finding the importance of the customers and not suited to find sequential patterns from MDB's.
TidFP algorithm by Ezeife and Zhang (2009)	The TidFp algorithm mines frequent item_sets from multiple sources using transaction ids for integrating patterns through set operations (e.g., intersect, union) in order to answer global queries involving multiple sources.	Answers queries from multiple related transaction tables.	It only mines frequent itemsets from multiple transaction tables and not frequent sequences.

Table 5: Summary of existing systems in multiple database mining

Problem Definition: Given multiple related sequence tables where each table consists of sequence id and corresponding sequence of items and a minimum support count ‘s’, the problem of mining frequent sequences from multiple related sequence databases is to mine the exact frequent sequences with support counts greater or equal to the given minimum support count ‘s’ from each sequence table to be able to answer complex queries on the multiple related sequence tables. For example, what are the frequent side effects that affect the patients p1 and p3? , is a query that can be answered by the proposed algorithm from Patient/sequence of drugs table and Drug/sequence of side effects table.

Feature contributions:

The following are the thesis features:

1. ***Mining frequent sequences from multiple related sequence databases having different structure as well as from multiple related sequence database having similar structure for comparative mining:*** The problem of answering complex sequence database queries involving related data from more than one table or database is solved. Example query that can be answered is “Find the discounts associated with frequent sequence of customers in the all named retail stores”.
2. ***Finding records that are related to frequent sequences in multiple databases faster:*** The frequent sequences along with their associated sequence ids (transaction ids) are mined from the multiple related sequence tables with less database scans and this is used to find the records that share the frequent sequences, unlike existing systems that present only the frequent sequences with more table scans. For example, for multiple related sequence tables, such as Drug/side-effects sequence table and Patient/Drug sequence table, the proposed algorithm will mine the frequent sequence of side-effects and its corresponding drugs from Drug/Side effects sequence table and will mine the frequent sequence of drugs and the corresponding patients from Patient/Drugs sequence table. Example query is finding the list of customers that purchase a sequence of products most frequently in all branches or all named retail stores.

3. ***Mining alternate types of information from competitive databases:*** Patterns such as stable patterns and their customers, frequent, trending patterns and their customers in a retail store can be a tool for finding important customers to be targetted for attrition by a competitive retail store as alternate types of information to advance competitor's business. Example queries that can be answered by mining such alternate information include:

    - Find the set of stable patterns from all the local branches.
    - Find the important customers that come to the retail store.
    - Find the local frequent products that are similar to each other.

Procedure contributions:

This thesis solves the unsolved problems identified in the feature contribution section above by proposing the following solutions.

1. To solve the problem of mining from multiple databases with similar or different schema structures in order to answer more complex sequence queries, this thesis proposes the an algorithm called the Transaction id frequent sequence pattern (TidFSeq) algorithm for mining frequent sequences from multiple related sequence databases having similar or different structures. The candidate item sets, multiple related sequence tables and minimum support count are used as input to the proposed algorithm. The thesis algorithm is an extension of the TidFP algorithm (Ezeife and Zhang (2009)) for mining frequent itemsets from multiple tables, which now mines frequent sequences from multiple tables using an extension of the ApproxMap sequential pattern algorithm (Hyue and *et al*. (2006)) for mining sequential patterns from only multiple tables having similar table schemas. While the TidFP and the ApproxMap algorithm approaches are used to pre-process and compute the format of the fequent sequences with transaction ids found, the ApproxMap approach for finding only approximate patterns are replaced by the GSP algorithm Srikant and Agrawal (1996) algorithm approach for finding exact sequences for general sequence databases. The item set sequences (ie. item set sequence-(abc), meaning items a, b and c are purchased during same transaction) were termed as I-step sequences and the multi set sequences (ie. multi set sequence-(a)(b)(c), meaning items a, b and c are

purchased during different transactions) were termed as S-step sequences by the SPAM algorithm (Ayres and et al (2002)). The definitions of I-step sequences and S-step sequences are used in the thesis algorithm. Thus, the main six steps of the proposed TidFSeq algorithm are:

I. ***Linking the frequent sequences to their Database transaction record ids:*** This is done by replacing the <frequent items, transaction id list> tuple used in the TidFp (Ezeife and Zhang (2009)) with a <sequence id, position id list> tuple. Each <sequence id, position id list > tuple consists of the sequence id and the sequence id attribute position of the items (sequences).

II. ***Find the frequent k sequences for k=1:*** Count the support of the candidate items from step 1 above, as the number of sequence ids in the position list. Then, keep the items with support greater or equal to minimum support as the frequent items.

III. ***Find the k+1 candidate sequences:*** This is done by joining the F_k frequent sequences by F_k frequent sequences using GSP_gen join approach, if the k+1 candidate sequence is not an empty set. If k+1 candidate sequence is an empty set, algorithm terminates and jumps to step VI.

IV. ***Find the frequent k+1 sequences for k>1: If the candidate k sequences is not an empty set, then the following steps are used:***

   1) If the candidate sequence is of the I-step form of (ab) which is single item sequence then, the following conditions, condition 1: items 'a' and 'b' have same sequence ids and condition 2: the positions ids corresponding to the sequence ids are same, must be satisfied. If the number of times both the conditions are satisifed is greater or equal to min-support count, then the single item sequence of form (ab) is frequent.

   2) If the candidate sequence is of the S-step form of (a)(b) which is multi set sequence then, the following conditions, condition 1: items 'a' and 'b' have same sequence ids and condition 2: position id of item 'a' is lesser than position id of item'b', must be satisfied. If the number of

times both the conditions are satisifed is greater or equal to min-support count, then the multi set sequence of form (a)(b) is frequent.

V. ***Continue finding higher order frequent sequences:*** Go to step 3, to continue finding all frequent sequences with longer items until either the candidate set generation step or frequent sequence generation step yields an empty set.

VI. ***Collect all found frequent sequences (FP):*** This is done as the union of all F1 to Fn, ie. the frequent sequences found so far.

2) The proposed algorithm uses the concept of <sequence id, position_id list> tuple for each frequent sequence, which enables to store the sequence ids in which the candidate sequences occur inside the <sequence id, position_id list> tuple, while the algorithm checks if the candidate sequence is frequent or not. This helps to avoid the costly method of first finding the frequent sequences and then mapping it to the corresponding sequence ids. Thus finding the frequent sequences along with their corresponding sequence ids in parallel, helps in finding records that are related to frequent sequences faster.

1.5 Outline Of Thesis:

The remainder of the thesis is organized as follows. Chapter 2 consists of related work in mining frequent itemsets, mining frequent sequences and mining frequent patterns from multiple databases. We have identified problems which are related to the problem studied in this thesis. Each has its own advantages and disadvantages. Next, in chapter 3, a detailed discussion of the problem addressed and new algorithm is proposed. In chapter 4, the performance analysis and comparative experiments are conducted in detail. Chapter 5 gives the conclusion and the future enhancements that can be done in the proposed algorithm.

CHAPTER 2: RELATED WORK

2.1 Frequent itemset mining algorithms

2.1.1 Apriori algorithm

The Aprori algorithm was introduced by Agrawal and Srikant (1994). The Apriori algorithm follows the Apriori principle: An itemset is frequent only if all of its sub-itemsets are frequent. The input to the algorithm is a transaction database, the user defined support count and the candidate-1 set of items. In the first step, the support counts of each item in the candidate-1 set is calculated. All the items that satisfy the minimum support count are retained and those that do not satisfy the minimum support count are pruned (ie. removed). The items that are retained form the frequent 1 (F1) set of items. Next step is to find the candidate 2 set from the frequent 1 set using the Apriori gen-join method. The Apriori gen-join is a self join of the F1 items with itself, where an item in the F1 set is joined only with the items that come after it in the F1 set (eg. {a,b,c} ap-gen join {a,b,c} is (a,b) (a,c) (b,c), c is not joined with the previous items 'a' and 'b' in the F1 set, since candidate itemsets (a,c) and (b,c) are same as (c,a) and (c,b)). Once, the (C2) candidate-2 itemsets are generated, the frequent-2 set is generated by pruning the infrequent candidate-2 itemsets as explained above. This process of genrating candidate sets and finding frequent sets from it keep continuing until there are no more candidate sets that can be generated. The working of the Apriori algorithm is shown through an example:

Input: Transaction db (Table 6), min-support count=3, Candidate -1 set= {1, 2, 3, 4}

Output: frequent itemset List (L).

Tid	Items
T1	1,2,3,4
T2	1,2,4
T3	1,2
T4	2,3,4
T5	2,3
T6	3,4
T7	2,4

Table 6: Transaction db

Step 1: Find frequent-1 items

The first step of Apriori is to count up the number of occurrences, called the support, of each item in the Candidate set (C1) separately, by scanning the database a first time. All the items that have support count greater than or equal to the input support count (ie. 3) are frequent items and are retained. The items that do not meet the minimum support count are pruned. The frequent 1 items in our example are: 1, 2, 3, 4.

Step 2: Candidate-2 itemset generation

The next step is to generate candidate-2 itemsets. Candidate itemsets (C2) are generated by doing Apriori-gen join of frequent 1 items (L1) with itself. The Apriori gen-join is a self join of the F1 items with itself, where an item in the F1 set is joined only with the items that come after it in the F1 set (eg. {a,b,c} ap-gen join {a,b,c} is (a,b) (a,c) (b,c), c is not joined with the previous items 'a' and 'b' in the F1 set, since candidate itemsets (a,c) and (b,c) are same as (c,a) and (c,b)). For example, apriori-gen join of frequent 1 items found in step 1 is:

C2={1,2},{1,3},{1,4},{2,3},{2,4},{3,4}

Step 3: Finding frequent-2 itemsets

From the set of candidate -2 itemsets obtained in the previous step, we can see that the pairs {1, 2}, {2, 3}, {2, 4}, and {3, 4} all meet or exceed the minimum support of 3, so

they are frequent. The pairs {1, 3} and {1, 4} are not, hence they are pruned. The frequent 2 itemsets for our example will be:

F2: frequent 2 itemsets = {1, 2}, {2, 3}, {2, 4}, and {3, 4}

Next, the candidate-3 itemsets are generated from the frequent-2 itemsets. The C3 set obtained by doing ap-gen join of F2 with itself is: {1,2,3} {1,2,4} {2,3,4}. Since none of the candidate-3 itemsets satisfy the minimum support count, there will be no frequent-3 itemsets. Since no more candidate itemsets can be generated, the Apriori algorithm stops.

The final fequent itemset list is the union of all the frequent itemssets found so far:

F=F1 U F2 = 1, 2, 3, 4, {1, 2}, {2, 3}, {2, 4}, {3, 4}.

Key features and shortcomings:

Candidate generation generates large numbers of subsets (the algorithm attempts to load up the candidate set with as many as possible before each scan). Apriori algorithm has slow processing times for huge datasets.

2.1.2 FP-tree Algorithm

The FP-Growth Algorithm, proposed by Pei and *et al* (2001) finds frequent itemsets without any candidate generation. It uses a tree structure known as the FP- growth tree, to compute the frequent itemsets. The input to the algorithm is a transaction database, the user defined support count and the candidate-1 set of items. In the first step, the support counts of each item in the candidate-1 set is calculated. All the items that satisfy the minimum support count are retained and those that do not satisfy the minimum support count are pruned (ie. removed). The items that are retained form the frequent 1 (F1) set of items. The set of frequent-1 items are sorted in the descending order of their support counts. And each itemset corresponding to the transaction id is ordered according to the descending order of their support counts. Next step is to construct the FP tree. An FP-tree is then constructed as follows. First, create the root of the tree, labeled with “null.”

A branch is created for each transaction. For example, for the transaction (sorted in descending order of support count) “T100: I1, I2, I5,” which contains three items

(I2, I1, I5), leads to the construction of the first branch of the tree with three nodes (where each node has a corresponding count specifying number of occurences of the node in the fp-tree), I2:1, I1:1, I5:1, where I2 is linked as a child to the root, I1 is linked to I2, and I5 is linked to I1. And then if the second transaction, T200, contains the items I2 and I4, it would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common prefix, I2, with the existing path for T100. Therefore, we instead increment the count of the I2 node by 1, and create a new node, I4: 1, which is linked as a child to I2: 2. In general, when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly. However, if the first item in the transaction does not have any common prefix with existing nodes in the fp tree, a new branch is created for the transaction, by connecting first item in the transaction to the root. The next step is to find conditional pattern base from the fp tree. For each item in the fp tree, all the prefix paths of the item (with the item as suffix) connecting the item to the root are found by following the node links (eg. The two prefix paths or branches of item I5 is I1:1, I2:1, I3:1 and I1:1 I2:1). These prefix paths now form the conditional pattern base for the suffix item (I5). From the conditional pattern base the infrequent items that do not satisfy the support count are pruned to get the conditional fp tree for the suffix item (conditional fp tree for I5 is I1:2 I2:2). Next all the frequent itemsets are generated by combining with the items in the conditional fp tree with the suffix item ({I1, I5} {I2, I5} {I1, I2, I5}). Similary, the rest of the frequent patterns are found for each suffix item. The final frequent itemset list is the union of all frequent itemsets found so far. The following example, shows the detailed explanation of the algorithm.

Input: Transaction db (Table 7), min-support count=2, Candidate items= {I1, I2, I3, I4, I5}

Output: frequent itemset list.

Step 1: frequent 1 itemsets in descending order of support count

The first scan of the database derives the set of frequent items (frequent 1-items) and their support counts (frequencies), by retaining only those items that have support count greater than or equal to the minimum support count (ie. 2).

TID	Items
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Table 7: Transaction db

The set of frequent items is sorted in the order of descending support count. This resulting set is denoted by L. Thus, we have L = {{I2: 7}, {I1: 6}, {I3: 6}, {I4: 2}, {I5: 2}}. Each transaction in the input transaction table, Table 7, is now sorted in the descending order of the support counts of the items in the transaction. Table 8 shows the sorted items in each transaction.

TID	Items
T100	I2, I1, I5
T200	I2, I4
T300	I2, I3
T400	I2, I1, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I2, I1, I3, I5
T900	I2, I1, I3

Table 8: Transaction db sorted according to descending order of support counts

Step 2: Fp-tree construction

An FP-tree is then constructed as follows. First, create the root of the tree, labeled with "null." Scan database (Table 8) once. The items in each transaction are processed in L order (i.e., sorted according to descending support count), and a branch is created for each transaction. For example, the scan of the first transaction, "T100: I1, I2, I5," which contains three items (I2, I1, I5 in L (descending) order), leads to the construction of the first branch of the tree with three nodes, I2:1, I1:1, I5:1, where I2 is linked as a child to the root, I1 is linked to I2, and I5 is linked to I1. The second transaction, T200, contains the items I2 and I4 in L order, which would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common prefix, I2, with the existing path for T100. Therefore, we instead increment the count of the I2 node by 1, and create a new node, I4: 1, which is linked as a child to I2: 2. In general, when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.

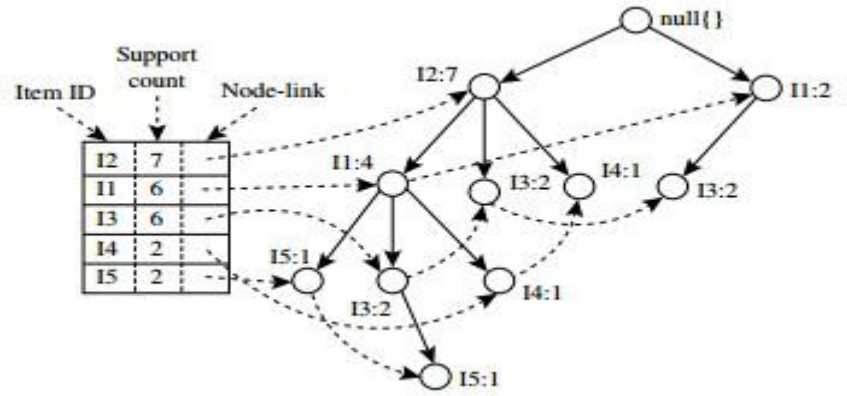

Figure 9: Construction of fp tree

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. The tree obtained after scanning all the transactions is shown in Figure 9 with the associated node-links. In this way, the problem of mining frequent patterns in databases is transformed into that of mining the FP-tree.

Step 3: Mining the fp-tree for frequent itemsets

Mining of the FP-tree is summarized in Table 9 and detailed as follows. We first consider I5, which is the last item in L. I5 occurs in two FP-tree branches of Figure 9. (The occurrences of I5 can easily be found by following its chain of node-links). The paths formed by these branches are I2, I1, I5: 1 and I2, I1, I3, I5: 1. Therefore, considering I5 as a suffix, its corresponding two prefix paths are I2, I1: 1 and I2, I1, I3: 1, which form its conditional pattern base. Using this conditional pattern base as a transaction database, we build an I5-conditional FP-tree, which contains only a single path, (I2: 2, I1: 2); I3 is not included because its support count of 1 is less than the minimum support count. The single path generates all the combinations of frequent patterns: {I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}. Similar to the preceding analysis, I3's conditional pattern base is {{I2, I1: 2}, {I2: 2}, {I1: 2}}. Its conditional FP-tree has two branches, I2: 4, I1: 2 and I1: 2, which generates the set of patterns {{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}} .This process is carried out for all the items.

Item	Conditional pattern base	Conditional FP-Tree	Frequential Pattern Generated
I_5	$[(I_2\ I_2:1),(I_2\ I_1\ I_3:1)]$	$(I_2:2, I_1:2)$	I_2I_5:2, I_1I_5:2, $I_2I_1I_5$:2
I_4	$[(I_2\ I_1:1),(I_2:1)]$	$(I_2:2)$	I_2I_4:2
I_3	$[(I_2\ I_1:I_2),(I_2:2),(I_1:2)]$	$(I_2:4, I_1:2)(I_2:2)$	I_2I_3:4, I_1I_3:2, I_1I_3:2
I_1	$[(I_2:4)]$	$(I_2:4)$	I_2I_1:4

Table 9: Final result

The final frequent itemset list is the union of all the frequent itemsets found so far. In our example, the final frequent itemset list F= I1, I2, I3, I4, I5, (I2, I5), (I1, I5), (I2, I1, I5), (I2, I4), (I2, I1), (I2, I3), (I1, I3), (I2, I1, I3).

Key features and shortcomings: The FP-tree algorithm improves Apriori to a great extent. Only two scans to the database is needed. Frequent Itemset Mining is possible without

candidate generation. It is a scalable technique for frequent pattern mining. Some disadvantages of FP tree is that tree is expensive to build.

2.2 Frequent Sequence mining algorithms

2.2.1 GSP Algorithm

The candidate generation method that is used in this algorithm will be used in the proposed algorithm. Generalized Sequential Patterns (GSP), an Apriori-based sequential pattern mining algorithm was introduced by Srikant and Agrawal (1996). The GSP algorithm makes multiple passes over the data. The first pass determines the frequent 1-item patterns (L1). Each subsequent pass starts with a seed set: the frequent sequences found in previous pass (Lk-1). The seed set is used to generate new candidate sequences (Ck). The candidate sequences are found using the GSP gen join. The GSP candidate generation method consists of 2 steps: Join phase and Prune phase. In order to obtain the k-sequence candidates (Ck), the frequent sequences found in previous step (Lk-1), joins with itself in an Apriori-gen way. This requires that every sequence - s in Lk-1 joins with other sequences - s'in Lk-1, if the last elements of s (excluding the first element of s) is same as first elements of s' (excluding the last element of s'). For example, sequence s-<(1) (2) (3)> can join with sequence s'- <(2) (35)>, since the last elements of s (ie. (2) (3)) is same as first elements of s' (ie. (2)(3)). After the join phase, in the prune phase, the candidate sequences (Ck) that do not satisfy the minimum support count are removed to get the next frequent sequences (Lk+1). This process is repeated till no more candidate sequence sets can be generated. The final frequent sequence list is union of all frequent sequences found so far. In order to understand the candidate generation method used in the algorithm, we shall run through an example from the input to the final output of the algorithm.

Input: Sequence table (Table 10), minimum support count=2, Candidate items= {A, B, C, D, E, F, G}

Output: frequent sequence patterns.

Step 1: Finding frequent 1 items

From the given sequence table (Table 10), we get a frequent 1 table (Table 11) by eliminating all items that have support count less than the given min support count 2.

SID	Sequences
1	<AB(FG)CD>
2	<BGD>
3	<BFG(AB)>
4	<F(AB)CD>
5	<A(BC)GF(DE)>

Table 10: GSP sequence table

Items	Count
A	4
B	5
C	3
D	4
F	4
G	4

Table 11: Frequent-1 items

Step 2: Generating candidate sets and finding frequent sequences

The candidate sequences are found using the GSP gen join. The GSP candidate generation method consists of 2 steps: Join phase and Prune phase. In order to obtain the k-sequence candidates (Ck), the frequent sequences found in previous step (Lk-1), joins with itself in

an Apriori-gen way. This requires that every sequence - s in Lk-1 joins with other sequences - s'in Lk-1, if the last elements of s (excluding the first element of s) is same as first elements of s' (excluding the last element of s'). For example, sequence s-<(1) (2) (3)> can join with sequence s'- <(2) (35)>, since the last elements of s (ie. (2) (3)) is same as first elements of s' (ie. (2)(3)). After the join phase, in the prune phase, the candidate sequences (Ck) that do not satisfy the minimum support count are removed to get the next frequent sequences (Lk+1). For example, in Table 12, sequence ⟨(1,2) (3)⟩ joins with ⟨(2) (3,4)⟩ since the last 2 items (2,3) in the first sequence and the first 2 items (2,3) in the second sequence are the same. Equally, ⟨(1,2) (3)⟩ joins with ⟨(2) (3) (5)⟩. The join operation then produces ⟨(1,2) (3,4)⟩ and ⟨(1,2) (3) (5)⟩. In this example, Sequence ⟨(1,2) (3) (5)⟩ is pruned since its contiguous sub-sequence ⟨(1,2) (5)⟩ is not frequent (not in frequent 3-sequence).

Frequent 3-Sequences	Candidate 4-Sequences	
	after join	after pruning
⟨ (1, 2) (3) ⟩	⟨ (1, 2) (3, 4) ⟩	⟨ (1, 2) (3, 4) ⟩
⟨ (1, 2) (4) ⟩	⟨ (1, 2) (3) (5) ⟩	
⟨ (1) (3, 4) ⟩		
⟨ (1, 3) (5) ⟩		
⟨ (2) (3, 4) ⟩		
⟨ (2) (3) (5) ⟩		

Table 12: Candidate sequences generated using GSP-gen join

Using the join phase and prune phase approach described above, the candidate-k sequences and frequent-k+1 sequences are generated from Table 11, which consists of the frequent 1 items. This process is repeated till no more candidate sets can be generated. The final sets of frequent sequences is shown in Figure 10:

Sequences			
1-Item	2-Items	3-Items	4-Items
A	AB	ABD	ABFD
B	AC	ABF	ABGD
C	AD	ABG	
D	AF	ACD	
F	AG	AFD	
G	BC	AGD	
	BD	BCD	
	BF	BFD	
	BG	BGD	
	CD	F(AB)	
	FA	FCD	
	FB		
	FC		
	FD		
	GD		
	(AB)		

Figure 10: Final frequent sequences

The candidate generation method used in this algorithm will be used in the proposed TidFSeq algorithm for mining multiple sequence tables.

2.2.2 SPAM algorithm

The Sequential Pattern Mining algorithm with vertical bitmap representation was introduced by Ayres and et al (2002), abbreviated as SPAM algorithm. The implementation was done based on vertical bitmap representation of the data and Depth First Search (DFS) Strategy was used for mining sequential patterns. The DFS search strategy combines depth first traversal search with effective pruning mechanism to reduce the search space. A vertical bitmap is created for each item in the dataset, and each bitmap has a bit corresponding to each transaction in the dataset. If item i appears in transaction j, then the bit corresponding to transaction j of the bitmap for item i is set to one; otherwise, the bit is set to zero. The database used to mine with SPAM can be considered as a lexicographic tree. Each sequence in the sequence tree can be considered as either a sequence-extended sequence or an itemset-extended sequence. For example, if we have a sequence sa = ({a, b, c}, {a, b}), then ({a, b, c}, {a, b}, {a}) is a sequence-extended sequence of sa and ({a, b, c}, {a, b, d}) is an itemset-extended sequence of sa. If we generate sequences by traversing the tree, then each node in the tree can generate sequence-extended children sequences and itemset-extended children sequences. The process of generating sequence-extended sequences is known as the sequence-extension step (abbreviated, the S-step), and

the process of generating itemset-extended sequences is known as the item-extension step (abbreviated, the I-step). Let us consider an example to understand the algorithm.

Input: transaction db (Table 13), min-support count=2, Candidate items= {a, b, c, d}

Output: frequent sequences

Step 1: Converting database to vertical bitmap

Sort the Database (Table 13) of itemsets in ascending Order of Cid and Tid. Store it in the form of vertical bitmap by setting (1) or (0) for itemsets depending on whether the item is found for a given transaction belonging to a specific sequence (Figure 11). For example, item 'a' is found in transaction ids: 1, 4, 5, the bits corresponding to those positions are set to 1 and other bits are set to 0.

CID	TID	Itemset
1	1	{a,b,d}
1	3	{b,c,d}
1	6	{b,c,d}
2	2	{b}
2	4	{a,b,c}
3	5	{a,b}
3	7	{b,c,d}

Table 13: Spam transaction db

CID	TID	{a}	{b}	{c}	{d}
1	1	1	1	0	1
1	3	0	1	1	1
1	6	0	1	1	1
-	-	0	0	0	0
2	2	0	1	0	0
2	4	1	1	1	0
-	-	0	0	0	0
-	-	0	0	0	0
3	5	1	1	0	0
3	7	0	1	1	1
-	-	0	0	0	0
-	-	0	0	0	0

Figure 11: Vertical Bitmap of items in transaction db

Step 2: Constructing lexicographic tree

The root of the tree is null (part of the sequence tree for items 'a' and 'b' are shown in Figure 12). Each sequence in the sequence tree can be considered as either a sequence-extended sequence or an itemset-extended sequence. For example, if we have a sequence sa = ({a, b, c}, {a, b}), then ({a, b, c}, {a, b}, {a}) is a sequence-extended sequence of sa and ({a, b, c},{a, b, d}) is an itemset-extended sequence of sa. If we generate sequences by traversing the tree, then each node in the tree can generate sequence-extended children sequences and itemset-extended children sequences. We refer to the process of generating sequence-extended sequences as the sequence-extension step (abbreviated, the S-step), and we refer to the process of generating itemset-extended sequences as the item-extension step (abbreviated, the I-step).

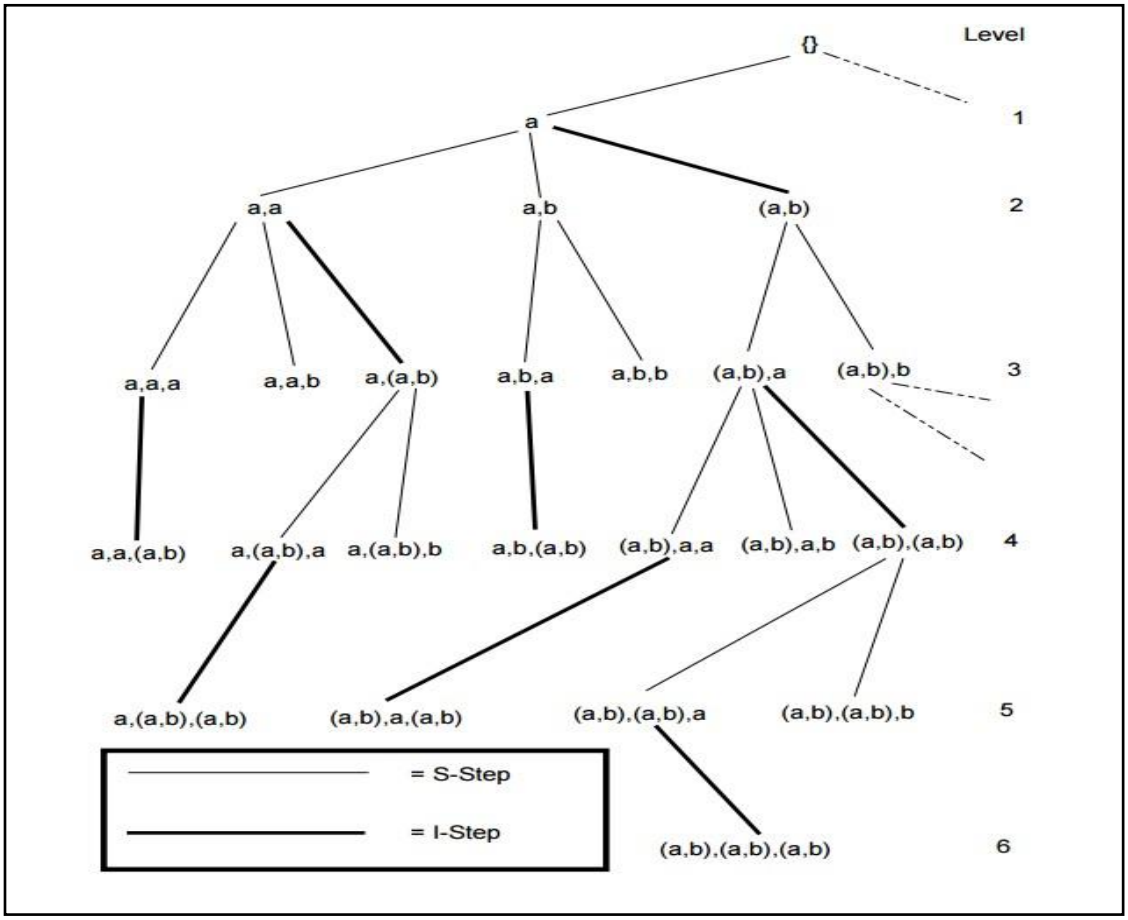

Figure 12: Lexicographic tree

Step 3: Finding frequent sequences

SPAM traverses the sequence tree described above in a standard depth-first manner. At each node n, the support of each sequence-extended child and each itemset-extended child is tested. Sequences of form (ab) -I step sequence and sequences of form a,b -S-step sequence have certain procedures to be followed for checking if they are frequent or not.

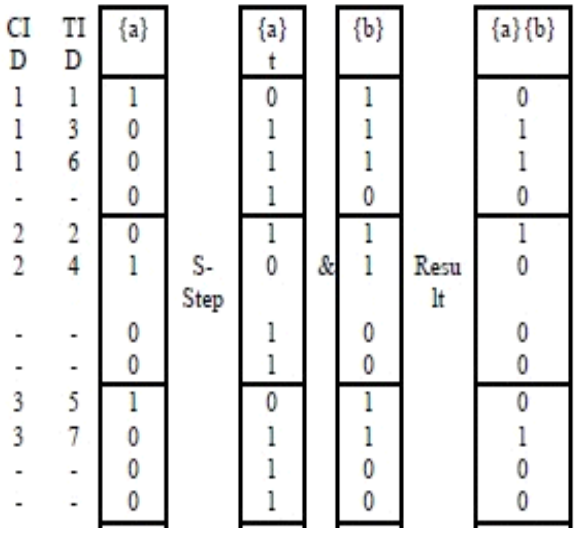

Figure 13: S-step

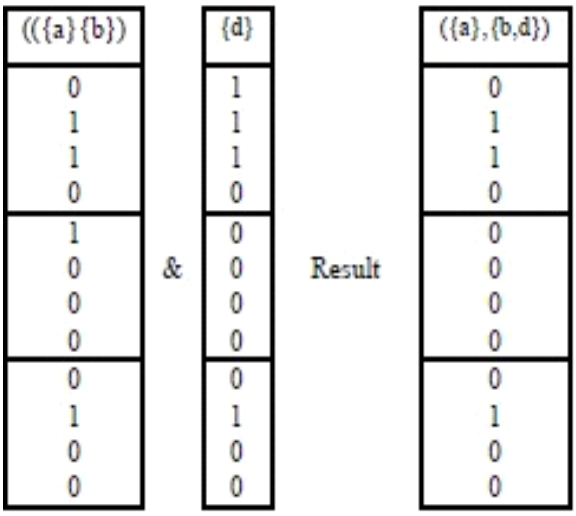

Figure 14: I-step

We demonstrate S-step and I-Step procedures of item {a} in Figures 13 and 14. Let us consider the bitmap B({a}), and suppose we want to generate B({a},{b}). Since ({a}, {b}) is a sequence-extended sequence of ({a}), we have to perform a S-step process on B({a}). For S-step, we first need to transform the bitmap of {a} into {a}s. The index of the first 1 bit in {a} should be transformed into 0. Then all the bits which are behind this bit should be set to "1". When the transformed bitmap {a}s is obtained, we do the bit-AND operation on {a}s and {b} to get the result of S-step of ({a},{b}).

For I-step, we simply do the bit-AND operation on the bitmaps say for e.g. of ({a}, {b}) and {d} to get the result of ({a},{b,d}). Considering the first customer (sequence id) in the final bitmap, the second bit and the third bit for the first customer have value one. This means that the last itemset (i.e. {b}) of the sequence appears in both transaction 2 and transaction 3, and itemset {a} appears in transaction 1. After I-step or S-step is finished, we accumulate the number of sequences that have more than one "true" bits in bitmap results. As shown in Figures 13 and 14, it can be seen that the support count of ({a}, {b}) is 4 and that of ({a},{b,d}) is 3. If min sup is set to 50% (i.e., 2 sequences), both ({a}, {b}) and ({a},{b,d}) will be viewed as frequent and will not be pruned, else will be pruned. Following this procedure, SPAM can generate the complete set of frequent sequential patterns by traversing only through parent sequences which are frequent in the lexicographic tree, since by the Apriori principle, any child sequence generated from a infrequent parent sequence will not be frequent.

Key features and shortcomings: SPAM (Ayres and *et al* (2002)) outperformed SPADE (Zaki (2001)) and PrefixSpan (Pei and *et al*.(2004)) in terms of short runtime for relatively large datasets (20 transactions) size, because of vertical bitmap representation used for counting recursively many times. For the same reasons as the average number of items per transaction and the average number of transactions per customer increases, and as the average length of maximal sequences decreases, the performance of SPAM increases even further relative to the performance of SPADE and PrefixSpan. Scans the original database once and transforms it into vertical bitmap table unlike Apriori based approaches which requires multiple database scans. Storage Space**:** Because SPAM uses a depth-first traversal of the search space, it is quite space-inefficient in comparison to SPADE/PrefixSpan. SPAM uses the vertical bitmap representation to store transactional data, for each item we need one bit for transaction in the database.

2.2.3 PrefixSpan algorithm

PrefixSpan algorithm was introduced in Pei and et al (2004). The PrefixSpan algorithm first finds the frequent 1 items from the candidate items by removing all those items that do not satisfy input support count. The next step follows a pattern growth, divide and conquer principle, where for each frequent 1 item, a projected database is created, having

all sequences from the input sequence table, that has the frequent 1 item as the prefix in the sequence (eg. If there are 2 sequences <abcd> and <(ae)d>, then the projected database for item ‘a’ will contain sequences <bcd> and <ed>). Next, the frequent 1 items in each projected database is found and combined with the parent frequent 1 item (ie. prefix of the projected db) to form a frequent sequence (ie. if frequent item in projected db of item ‘a’ is ‘d’, then the frequent sequence is <ad>). Next, the projected databases for the frequent sequences found in each of the parent projected databases is created in a similar manner. And new frequent sequences are found in the each of the newly created projected databases (for eg, the projected db for frequent sequence <ad> will contain all the sequences that have <ad> as their prefix). This process continues until no more projected databases can be created.

Input: min support count=2, sequence db (Figure 15), Candidate items= {a, b, c, d, e, f}

Output: frequent sequences

Step 1: Find frequent-1 items

The first step would be to find the frequent length 1 sequential patterns as shown in Figure 15. Item 'g' is pruned out since it has count less than min support count.

Step 2: Create projected databases:

The next step is to divide search space into set of projected databases according to the prefixes. For example, for each sequence of the sequence db, the projected db of frequent 1 item 'a' would consist of all the items that appear after the item 'a' (ie.) the projected db for 'a' will consist of all the sequences with its prefix as 'a' . The Figure 15 gives the projected dbs for all the frequent 1 items.

Step 3: Finding frequent sequences from projected databases

Consider projected database of 'd'. The projected db is scanned to find the frequent items in it. In our example, only 'b 'and 'c' are frequent. Now the projected db for <db> and <dc> are constructed using step 2. Now their respective projected dbs are scanned to get the frequent items in their projected dbs. This process is carried out until no more projected dbs can be constructed. Now we get all the frequent sequences with 'd' as the prefix. In the

Figure 15, it can be seen that, <dc>,<dcb> are the frequent sequences with prefix as frequent item 'd'. This procedure is carried out for the projected dbs of rest of frequent items to get all the frequent sequences.

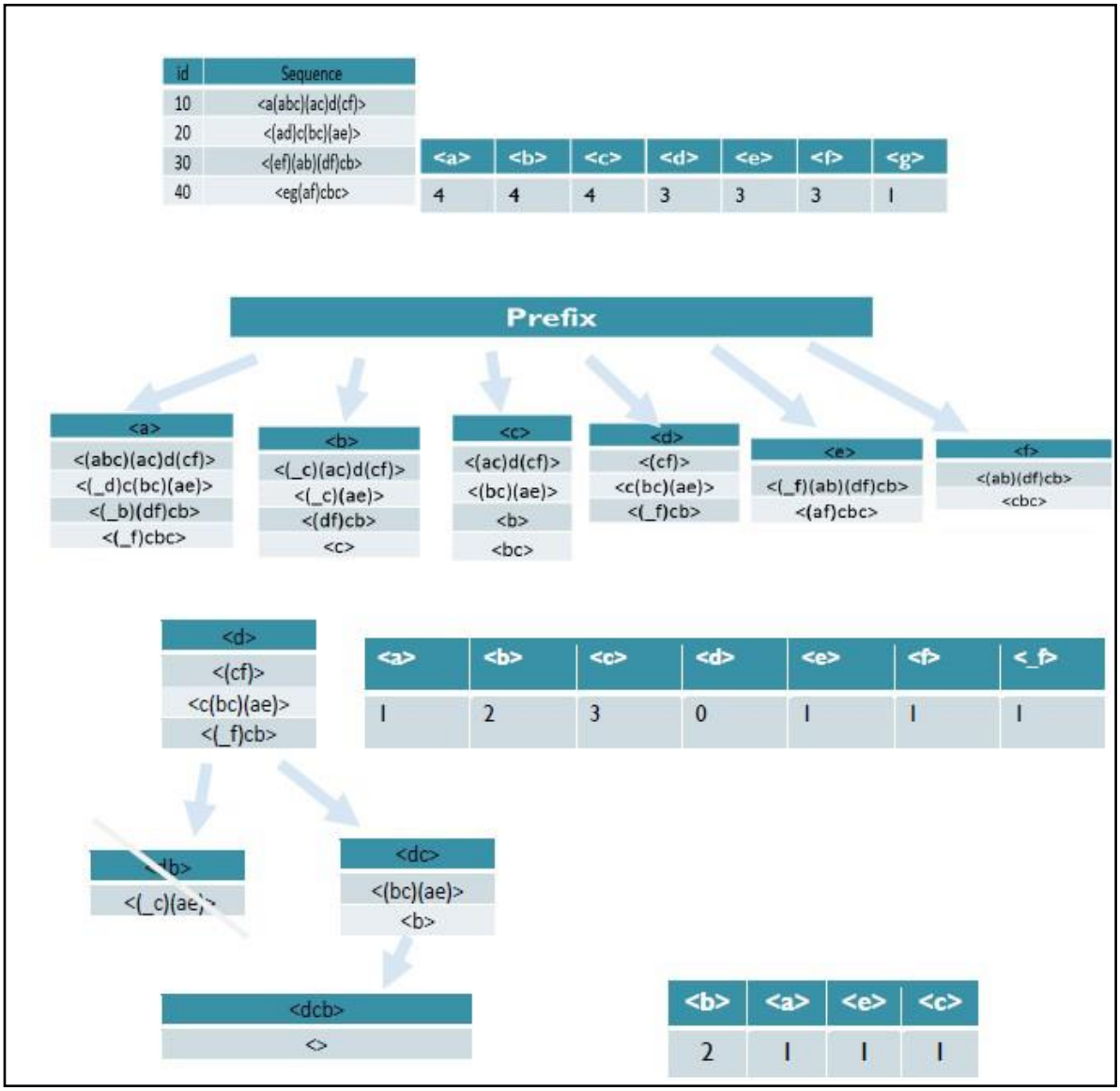

Figure 15: PrefixSpan algorithm

Key features and Shortcomings:

It is faster than its predecessor algorithms GSP (Agrawal R, Srikant R (1995)), Freespan (Han, Pei and et al (2000)) and SPADE (Zaki (2001)). It avoids multiple scans of database. Unlike traditional a priori-based approach which performs candidate generation-and test, PrefixSpan does not generate any useless candidate and it only counts the frequency of

local 1-itemsets. It also consumes less memory than GSP (Agrawal R, Srikant R (1995)) and SPADE (Zaki (2001)).

2.2.4 PLWAP algorithm

The PLWAP algorithm was introduced by Ezeife.C.I and Yi Lu (2005). The PLWAP algorithm first scans the database to get the frequent 1 items that satisfy the input support count and then removes the infrequent items in each sequence of the database. Next, the PLWAP tree is constructed. The PLWAP tree starts with an empty root. The nodes consist of the item: count and also a position code for easy tree traversal. If a new node is attached to an existing branch then position code has an extra 1 appended to the parent node's position code and if the new node is attached to the root (ie. new branch is formed) then position code has a extra 0 appended to the parent node's position code. For example, if there is a branch from root having nodes a: 1 (position code: 1), b:1 (position code: 11), then when attaching new nodes a, b, c to the branch, the counts of items a and b in the existing branch increase by 1, since when same items are inserted into an existing branch the counts of those overlapping items in their corresponding nodes are also increased. The last new item 'c' will attach as a new node (with count=1 and position code= 111) to node containing item 'b', forming a updated branch: a: 2 (position code: 1), b:2 (position code: 11), c:1 (position code: 111). If there is a new node 'd' waiting to be inserted, then a new branch is created with node 'd:1' (position code:10) attached to root. In a similar fashion the entire PLWAP tree is constructed. Next final step is to mine the tree for frequent sequences. The PLWAP mining starts from the root. It finds the total count of each sequence pattern in all the branches and if the count satisfies the minimum support count, then it is a frequent sequence. For example, if sequence ab is found in 3 different branches with counts 1, 2 and 2 respectively, then total support count of sequence ab is 5. Using this approach all frequent sequences are mined from the PLWAP Tree.

Input: Sequence database (Table 14), support count=2, Candidate items= {a, b, c, d, e, f, g}

Output: frequent sequences

TID	Web access sequences
100	abdac
200	abcae
300	babfac
400	afbacfcg

Table 14: Web access sequence database

Step 1: Frequent 1 items

First, the database (Table 14) is scanned to get the frequent items (ie. items that pass the support count) and the infrequent items are removed from the sequences showed in Table 14 to get the filtered sequences as:

{abac, abcac, babfa, afbacfc}, as well as the frequent 1-items {a:4, b:4, c:3, f:2}.

Step 2: PLWAP tree construction

The constructed PLWAP tree is as given in Figure 16. The leftmost branch of this tree shows the insertion of the first sequence abac when the first a node and b node initially had a count of '1' each. Then, when the second sequence abcac is inserted, the leftmost branch top nodes counts and position codes became a: 2:1 and b:2:11. Since the next child of this b node is not c, then, a new child node of b is created for the suffix sequence cac as shown in the Figure 16. The remaining sequence 3 and 4 of the batch are inserted into the PLWAP tree in a similar fashion. Then the tree is traversed pre-order way to link the frequent 1-item header nodes {a:4, b:4, c:3, f:2} to the PLWAP tree nodes using the dashed arrow headed links. For example, following the left subtree of "Root", we find an a:3:1 node that is linked to the frequent header of its kind 'a', then the next a:1:111 node found during pre-order traversal has a link from the first linked 'a' node to itself. Next operation is to mine this constructed PLWAP tree for frequent patterns meeting a minimum support threshold.

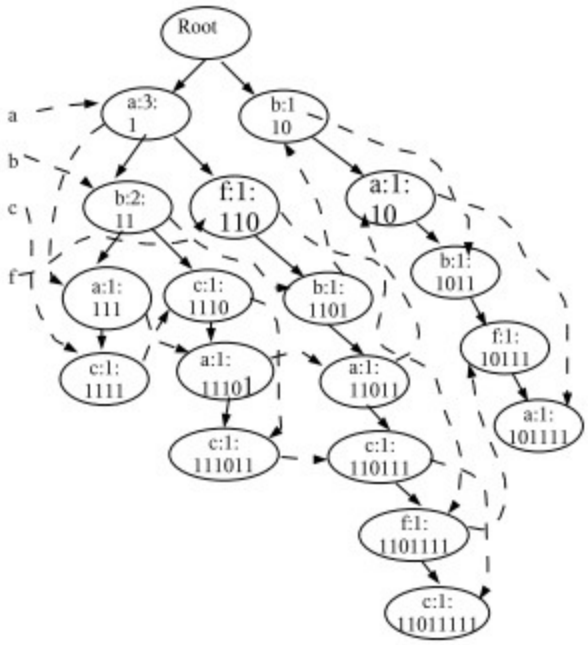

Figure 16: Construction of PLWAP tree

Step 3: Mining PLWAP Tree

The PLWAP mining method starts from PLWAP Root to examine sub tree suffix root sets and extract all patterns with total count on all different branches in the suffix root sets at that height of the tree, greater or equal to minimum tolerance support. For example, the left branch rooted at a:3:1 and right branch rooted at b:1:10 form the first suffix root sets. It can be seen that a:3:1, a:1:10 mean that pattern a has a total count of 4 and is frequent. Then, ab:2:11, ab:1:1101 and ab:1:1011 mean ab has a count of 4 and is frequent. Also, aba:1:111, aba:1:11101, aba:1:11011 and aba:1:101111 mean a count of 4 for pattern aba. The algorithm uses position codes of nodes to quickly determine if they are on different branches of the tree and their counts can be added. The final frequent patterns are {a:4, aa:4, aac:3, ab:4, aba:4, abac: 3, abc: 3}.

Key features and shortcomings:

The PLWAP algorithm eliminates the need to store numerous intermediate WAP trees during mining. Since only the original PLWAP tree is stored, it drastically cuts off huge memory access costs, which may include disk I/O cost in a virtual memory environment, especially when mining very long sequences with millions of records. The performance of PLWAP seems to get degraded with very long sequences having sequence length more than 20 because of the increase in the size of position code that is required to build and process for very deep PLWAP tree.

2.3 Mining frequent patterns from multiple databases

2.3.1 ApproxMap algorithm

The ApproxMap algorithm introduced by Hyue and *et al*. (2006) breaks its input sequences (e.g., the 2-column sequence < (123) (45) >) into columns so it can find the collection of approximate frequent sequences of all the columns as the approximate sequence of the database. The same method is used to integrate frequent sequences from each MDB that must have the same table structure. The first step is to convert the input tables into equal length sequence tables. For each row in the table, empty padded elements are placed so that all the sequences are of the same length. Since all sequences are of same length the sequences can be divided into columns. The next step is to generate weighted sequence using multiple alignment (ie. for each column, in each table, the number of occurences of each item in that column is counted and the total count for each item is called the weighted sequence). Next, retain all the items in the weighted sequence that satisfy the minimum support count. The retained items form the local approximate patterns for each table. After the local approximate patterns are obtained they are grouped into a table, where each local pattern is associated with the local table id from which it was obtained. The ApproxMap algorithm is applied to the set of local patterns in a similar way as explained before to get the global patterns. The following shows the explanation of the algorithm with example.

Input: Two multiple local cutomer sequence tables, Table 15 (Windsor) having candidate items {1, 2, 3, 4} and Table 16 (Hamilton) having candidate items {1, 2, 3, 4} as inputs and also a minimum support count of ‘2’.

Output: Approximate frequent patterns obtained from each table.

sid	sequences
1	<(123) (1)>
2	<(123) (3) (2)>
3	<(3) (4)>
4	<(1) (3) (2)>

Table 15: Windsor branch sequence table

sid	sequences
1	<(123) (34)>
2	<(3) (4)>
3	<(1 2) (1) (3)>
4	<(1) (3)>

Table 16: Hamilton branch sequence table

Step 1: The first step is to convert the input tables into equal length sequence tables. For each row in the table, empty padded elements are placed so that all the sequences are of the same length. Since all sequences are of same length the sequences can be divided into columns. Following the above rule, the input tables, Table 15 and Table 16 are converted into Table 17 and Table 18.

sid	sequences		
1	<(123)	(1)	>
2	<(123)	(3)	(2)>
3	<(3)	(4)	>
4	<(1)	(3)	(2)>

Table 17: Preprocessed Windsor table

sid	sequences		
1	<(123)	(34)	>
2	<(3)	(4)	>
3	<(1 2)	(1)	(3)>
4	<(1)	(3)	>

Table 18: Preprocessed Hamilton table

Step 2: The next step is to generate weighted sequence using multiple alignment. For each column, in each table, the number of occurences of each item in that column is counted and the total count for each item is called the weighted sequence. For example, in Figure 17, we can see the weighted sequence (ie. item count for each column) for both the preprocessed input tables. In the first column for the first table in Figure 17, we can see that item '1' occurs three times (occurs in sid 1,2 and 4), the weighted sequence will have the item '1' and it's count (ie. 3). Likewise, the weighted sequence has the item: count for all the items in that column.

Step 3: The next step is to get the approximate patterns from the weighted sequence. All the items that have count greater than or equal to the input minumum support count (ie. 2) are taken to be a part of aproximate sequence. For the first table in the Figure 17, the approximate frequent pattern is < (123) (3) (2) >, since only items 1, 2 and 3 satisfy the support count. Similarly, < (123) (34) > is frequent approximate pattern for second table in Figure 17.

sid	sequences		
1	<(123)	(1)	>
2	<(123)	(3)	(2)>
3	<(3)	(4)	>
4	<(1)	(3)	(2)>

1:3 2:2 3:3 1:1 3:2 4:1 2:2

Approx pattern: <(123) (3) (2)>

sid	sequences		
1	<(123)	(34)	>
2	<(3)	(4)	>
3	<(1 2)	(1)	(3)>
4	<(1)	(3)	>

1:3 2:2 3:2 3:2 4:2 1:1 3:1

Approx pattern: <(123) (34) >

Figure 17: Support Counts of Candidate Items in Each Sequence Column

Output: Approximate frequent sequences for table 17: < (1 2 3) (3) (2)>

Approximate frequent sequences for table 18: < (1 2 3) (34)>

Step 4: After the local approximate patterns are obtained they are grouped into a table, Table 19, where each local pattern is associated with the local table id from which it was obtained. The ApproxMap algorithm is applied to the set of local patterns in a similar way as explained before to get the global patterns. For example, in Table 19, the global pattern obtained from the local result sets will be < (1 2 3) (3)>.

Local table no.	Approx.pattern
Table 17	<(1 2 3) (3) (2)>
Table 18	<(1 2 3) (34)>
Global pattern	<(1 2 3) (3)>

Table 19: Finding global approximate patterns

Limitations:

1) The major drawbacks of this algorithm is that though it mines multiple databases, it does not generate exact sequence patterns to be able to answer exact queries for multiple sequence tables.
2) The ApproxMap algorithm was designed with a goal of mining multiple customer sequence tables. That is, the algorithm was used on local customer purchase sequence tables (such as Windsor, Hamilton, London branches) of a store to find the local approximate frequent sequences of products purchased by the customers. The ApproxMap algorithm cannot handle multiple related sequence tables that have different table structure with different column names, for example consider two tables, table a: (Drug/side-effects) and table b: (Patient/drugs), the ApproxMap algorithm will mine the frequent side-effect patterns from table a and frequent drug patterns from table b, but the output generated will not be able to handle queries such as: What are the possible frequent side effects that the patient p1 and p3 suffer from?.

2.3.2 TidFp Algorithm

The next notable algorithm for mining multiple databases is the TidFP algorithm by Ezeife, C.I., Zhang, D (2009) which mines frequent item_sets from multiple sources using transaction ids for integrating patterns through set operations (e.g., intersect, union) in order to answer global queries involving multiple sources. The first step of TidFP algorithm, scans the tables and obtains all items with their transaction IDs in the format of a <item,tid-list> tuple for each table. Next, for each table, if the number of transaction ids in which each item occurs is less than support count, they are pruned. The items that are retained are the frequent 1 items. Next step is to find the candidate itemsets. Candidate 2-item itemsets (C2) will be found by performing F1 Apriori *map -gen join with itself*: The candidate itemsets will be obtained in a slightly different way when compared to the apriori-gen join (Agrawal and Srikant (1994)) ie. the transaction ids in which the itemsets occur will also be derived by intersecting transaction ids. For example, apriori *map -gen join* of two tuples*:* <i1,(tid1,tid2)> (Apriori *map -gen join)* <i2,(tid1,tid5)> = <i1 i2,(tid1)> ,where item i1 and its tid-list joins with item i2 and its corresponding tid-list, the resulting candidate itemset is (i1 i2) and its tid-list is the intersection of (tid1,tid2) and (tid1,tid5) which is tid1. The frequent-2 itemsets are obtained by pruning the infrequent itemsets just like how frequent 1 items were found. The process of candidate generation and frequent itemset generation from the candidates continues until no more candidates can be generated. The final frequent itemset list is the union of all the frequent itemsets found so far.

A detailed example of the TidFp algorithm is given below:

Input: Two multiple related transaction tables, Table 20 (Drug/side-effects) having candidate items {1, 2, 3, 4, 5} and Table 21 (Patient/drugs) having candidate items {D1, D2, D3, D4} as inputs and also a minimum support count of ‘3’.

Output: Frequent itemsets and their corresponding transaction ids

Tid(drug)	Items(Side Effect)
D1	1 3 4
D2	2 3 5
D3	1 2 3 5
D4	2 5

Table 20: Transaction Table Drug/Side-effect

Patient	Drug
P1	D1D2
P2	D1D2D3
P3	D3D4
P4	D1D2D4

Table 21: Transaction Table Patient/Drug

Step 1: The first step of TidFP scans the tables once and obtains all items with their transaction IDs in the format of a <item,tid-list> tuple in each table. The tuples for Table 20 and Table 21 are :

Tuples for Table 20: <1,(D1, D3)> <2,(D2,D3,D4)> <3,(D1,D2,D3)> <4,(D1)> <5,(D2,D3,D4)> and tuples for Table 21: <D1,(P1,P2,P4)> <D2,(P1,P2,P4)> <D3,(P2,P3)> <D4,(P3,P4)>.

Step 2: Next, if the number of transaction ids in which each item occurs is less than support count (ie. 3), they are pruned. For example, in Table 20, item '1' appears only in tids D1 and D3 (ie. <1,(D1,D3)>), which does not satisfy the support count and hence it is pruned. Likewise, once all the infrequent items are pruned, the frequent-1 (F1) items for Table 20 and Table 21: (Frequent-1) <item,tid-list> tuples for Table 20: <2,(D2,D3,D4)>

<3,(D1,D2,D3)> <5,(D2,D3,D4)> and (Frequent-1) <item,tid-list> tuples for Table 21: <D1,(P1,P2,P4)> <D2,(P1,P2,P4)>

Step 3: Next step is to find the candidate itemsets. Candidate 2-item itemsets (C2) will be found by performing F1 Apriori *map -gen join with itself*: The candidate itemsets will be obtained the same way as apriori-gen join (Agrawal and Srikant (1994)). But the only modification is that the transaction ids in which the itemsets occur will also be derived by intersecting transaction ids. For example, apriori *map -gen join* of two tuples*:*

<i1,(tid1,tid2)> (Apriori *map -gen join)* <i2,(tid1,tid5)> = <i1 i2,(tid1)> ,where item i1 and its tid-list joins with item i2 and its corresponding tid-list, the resulting candidate itemset is (i1 i2) and its tid-list is the intersection of (tid1,tid2) and (tid1,tid5) which is tid1.

Following the above candidate generation method, the (C-2) candidate-2 itemsets for Table 20 and Table 21: C-2 <itemset,tid-list> tuples for Table 20: <2 3,(D2,D3)> <2 5,(D2,D3,D4)> <3 5,(D2,D3)> and C-2 <itemset,tid-list> tuples for Table 21: <D1 D2,(P1,P2,P4)>. The frequent-2 itemsets are obtained by pruning the infrequent itemsets using same pruning method as explained before. The process of candidate generation and frequent itemset generation from the candidates continues until no more candidates can be generated. The final frequent itemset list is the union of all the frequent itemsets found so far. The final outputs (Frequent itemsets and corresponding transaction ids) for Table 20 and Table 21:

<2,(D2,D3,D4)> <3,(D1,D2,D3)> <5,(D2,D3,D4)> <2 5,(D2,D3,D4)>

D1,(P1,P2,P4)> <D2,(P1,P2,P4)> <D1 D2,(P1,P2,P4)>

Using the outputs from both the tables (Table 20 (Drug/side-effects) and Table 21 (Patient/drugs)) can be used to answer queries such as:

Query: What are the possible frequent side effects that the patient P1, P2 and P4 suffer from?.

Solution: The answer to the query (Patients P1,P2,P4 buy drugs D1 and D2 and drugs D1,D2 have common side-effect ‘3’) can be found by the intersection of the result sets of Table 20 and Table 21:

<3,(D1,D2,D3) > ∩ <D1 D2,(P1,P2,P4)> = <3, (P1,P2,P4)>

The major drawback is that the TidFp algorithm handles queries for multiple related transaction tables, not multiple related sequence tables.

2.3.3 Kernel estimation to identify valuable customers

The kernel estimation algorithm by Zhang and *et al* (2009), introduced the concept of generating the local and global important cutomers (ie. customers who regularly purchase high value products from the store) to a retail store having local branches. This algorithm makes the assumption that a customer 'A' visting local branch X will also visit local branch Y. The local databases contain cutomer information, such as the expenditure of the customer in that local branch and the number of visits and transactions made by the customer in that local branch. Using this information the kernel estimation algorithm uses a mathematic approach to first find the local importance of each customer to the local branch. And then the global importance of the customer to the retail store.

The Figure 18 consists local db 1 and local db 2, each db consists of one local table representing customer information of a local branch. The tables consists of the attributes :NT-number of transactions, CLT-average customer lifetime (or visting) of that shop, CID-customer id, CE-customer expenditure as attributes and D-discount rate given for customer every year.

Input : Local db1 and db2 (Figure 18).

Output: global estimated CLV and global importance of each customer

Local Database 1:

Windsor branch

CUSTOMER Id	CLT	No. of transactions	cutomer expenditure per year	D (%)
A	2	500	1100	10
B	2	600	1100	10
C	2	550	1500	10

Local Database 2:

Hamilton branch

CUSTOMER Id	CLT	No. of transactions	cutomer expenditure per year	D (%)
C	2	100	500	10
D	2	600	1000	10
A	2	150	500	10

Figure 18: Local databases

Step1: Compute local Customer lifetime values of each customer.

In the first step the the CLV-customer lifetime value for each customer in that local branch is calculated using:

$$CLV_i = \sum_{t=1}^{n} p_i(t)\left[\frac{1}{1+d}\right]^{t-1}$$

Where, pi(t) denotes the customer expenditure of customer i during time span t (ie. cutomer expenditure per year column in Figure 18), and n denotes the CLT of customer i and d denotes the discount rate during the lifetime span 1 to n (ie. Column D in Figure 18). Using the above formula CLV is calculated, for example, CLV of customers in Windsor Branch (whose n value is 2 years, d value per year is 10%):

CLV value of A is: $\sum^{2}_{t=1}$ (1100)*[1/(1+0.1)]$^{t-1}$=2099.9

CLV value of B is: $\sum^{2}_{t=1}$ (1100)*[1/(1+0.1)]$^{t-1}$=2099.9

CLV value of C is: $\sum^{2}_{t=1}$ (1500)*[1/(1+0.1)]$^{t-1}$=2863.63

In a similar manner, the local CLV of all the customers in local branches (Windsor and Hamilton) is calculated and values are:

Windsor branch: customer A= 2099.9, B=2099.9, C=2863.63 and Hamilton branch: C=995, D=1990, A=995

Step 2: Compute the estimated CLV of each local db and importance of customer to that local db

The estimated local CLV is calculated using the formula:

$$E_L(X) = \frac{\sum_{i=1}^{n} CLV_{Li}}{n}$$

$E_L(X)$- estimated CLV of each local db is average of the local CLV of that local db. The $E_L(X)$ of Windsor are: CLV values of (A + B + C)/3= 1800.3 and Hamilton branch are: CLV values of (C+ D + A)/3=1326.6.

The importance of each customer i to the local db ie. Local branch is calculated using:

$K_{Li=} |(X_i - E_L(X))/ h_n|$ where, X_i is the local CLV of each customer and $E_L(X)$- estimated CLV, value of h_n is given as n (ie. no. of customers in local db)^-1/5= 3^-1/5=0.802. The exponent value of -1/5 is user-defined. The value is chosen such that, when it is used as a exponent value for the number of customers in local database, the answer yielded is a positive value less than 1. For example, importance of customer A in Windsor branch=(1208.9-1800.3)/0.802=736.72. The importance of each customer in Windsor branch are: A=736.72, B=736.72, C=1475.8 and importance of each customer in Hamilton branch are: C=1003, D=124, A=1003.

Step 3: Compute global estimated CLV and global importance of each customer

The global CLV is is the average of the estimated local CLV of each local db obtained in previous step. It is calculated using:

$$E_G(X) = \frac{\sum_{i=1}^{n} CLV_{Gi}}{n}$$

The global CLV = CLV of Windsor + CLV of Hamilton branches/2=1800.3+1326.6/2=1563.4. The global importance of each customer is calculated using: $K_G=\sum_1^{num} K_{Li}$ ie. Summation of local importance of customer i and num is the number of local branches that the customer i has shopped in, for example customer A has visited branches Windsor and Hamilton, so num is 2. The global importance of each customer is: A=1739, B=736.72, C=2478.7, D=124. Hence the final outputs are the global estimated CLV =1563.4 and global importance of each customer: A=1739, B=736.72, C=2478.7, D=124.

This paper is aimed at deriving the importance of the customers to local and global databases of the store. Though it does not mine frequent patterns from multiple databases, it is a major paper in the field of mining multiple databases.

2.3.4 Mining Stable patterns from multiple correlated databases

The hierarchial Gray clustering algorithm by Yaojin and et al (2013), introduced a new concept of stable patterns. The hierarchial Gray clustering algorithm defines an item 'a' as stable, if the item satisfies the minimum support count 's' in each of the local transaction table (T1,T2..TN, where Ti, 1<=i<=N, is a local transaction table) that it occurs and also the variation of the support count of that item 'a' is less than or equal to a user defined variation value 'v'. The algorithm clusters stable items found from multiple transaction tables into stable patterns according to the similarity of their timestamps (ie. the time at which the stable item 'a' was purchased by a cutomer). The use of stable patterns helps us to identify the standard set of constantly purchased items on a global scale in a retail store having local branches. The HGCA algorithm thus introduces a clustering algorithm that clusters the stable items (obtained from a similarity matrix produced using Gray relational analysis) from multiple time-stamped transaction tables (ie. correlated tables) into stable patterns. The algorithm is explained in detail with the following example.

Input: Multiple time stamped transaction tables (Table 22 and Table 23), user defined support count 's'=2, user-defined varivalue=1, confidence level=0.5.

Output: Stable patterns.

tid	items	Timestamp
1	X1,X2,X4	0.0,0.18,0.12
2	X2,X3,X1	0.2,0.1,0.6
3	X3,X1,X4	0.15,0.5,0.18

Table 22: Transaction table 1

Tid	items	Timestamp
1	X4,X3,X1,X2	0.1,0.17,0.8,0.19
2	X3,X1,X2,X4	0.12,1,0.19,0.2
3	X5,X4,X3,X2	0.8,0.1,0.1,0.2

Table 23: Transaction table 2

Step 1: Generating stable items : Table 22 and Table 23 are transaction tables where each item in each of the transactions in both the tables have a timestamp associated with it. For example, in tid 1 of Table 22, timestamps of X1, X2, X4 are 0.0, 0.18, 0.12. A item that has support count >= to 's' (ie. 2) and the variation of support counts of the item in both tables, Table 22 and Table 23 (ie. Item X1 has support counts 3 and 2 in Table 22 and Table 23, so variation is 3-2=1) is <= 'varivalue' (ie. 1) is considered to be stable item. In our example, X1, X2, X3, X4 are stable items.

Step 2: Calculating gray relational grade matrix (M): Next, using the timestamps of each item we construct a gray relational grade matrix. In order to find the gray relational grade, GRA-Gray Relational Analysis is used, which is used to find similarity between items (Deng J.L (1989)). The following steps are used to find gray relational grades, that will be used to fill the matrix M:

(1) $x_0(k)$, where k=1,2..n, is gray reference factor and $x_i(k)$, where i=1,2…m is gray comparative factor. Gray relation between $x_0(k)$ and $x_i(k)$ is written as $r(x_0(k),x_i(k))$, which is found by following equation (1):

$$r(x_0(k),x_i(k)) = \min_i \min_k |x_0(k)-x_i(k)| + \delta \max_i \max_k |x_0(k)-x_i(k)| \quad \ldots\ldots(1)$$

where, δ is a discrimination coefficient with a value from zero and one. In general, we set $\delta = 1$.

(2) From the gray relation, the gray relational grade is calculated using the following equation (2):

$$r(X_0, X_i) = \frac{1}{n}\sum_{k=1}^{n} r(x_0(k), x_i(k)).$$

…….(2)

Using the equations 1 and 2, each cell of the gray relational grade matrix (ie. M11,M12..M44) can be found. For example, in our example, in order to calculate the first row of grade matrix M (M11, M12, M13, M14) the reference factor: $x_0(k)$, is x1 and its timestamps (x_1 = {0.0,0.6,0.5,0.8,1}) and the comparative factors: xi(k), where i=2,3,4 (ie.x2 = {0.18,0.2,0.19,0.19,0.2}, x3 = {0.1,0.15,0.17,0.12,0.1},x4= {0.12,0.18,0.1,0.2,0.1}). The gray relation grade r(x1, x1) (ie. M11) is calculated using equations 1 and 2 as follows. Using equation 1, the gray relations inside x_1 = {0.0,0.6,0.5,0.8,1} are found:

$r(x_1,x_k)$ = min (min(|0.0-0.0|, |0.0-0.6|, |0.0-0.5|, |0.0-0.8|, |0.0-1|)) + 1 * max (max(|0.0-0.0|, |0.0-0.6|, |0.0-0.5|, |0.0-0.8|, |0.0-1|)) = 1, k=1, 2, 3, 4, 5

$r(x_2,x_k)$ = min (min(|0.6-0.0|, |0.6-0.6|, |0.6-0.5|, |0.6-0.8|, |0.6-1|)) + 1 * max (max(|0.6-0.0|, |0.6-0.6|, |0.6-0.5|, |0.6-0.8|, |0.6-1|)) =1, k=1, 2, 3, 4, 5

$r(x_3,x_k)$ = min (min(|0.5-0.0|, |0.5-0.6|, |0.5-0.5|, |0.5-0.8|, |0.5-1|)) + 1 * max (max(|0.5-0.0|, |0.5-0.6|, |0.5-0.5|, |0.5-0.8|, |0.5-1|)) =1, k= 1, 2, 3, 4, 5

$r(x_4,x_k)$ = min (min(|0.8-0.0|, |0.8-0.6|, |0.8-0.5|, |0.8-0.8|, |0.8-1|)) + 1 * max (max(|0.8-0.0|, |0.8-0.6|, |0.8-0.5|, |0.8-0.8|, |0.8-1|)) =1, k=1, 2, 3, 4, 5

$r(x_5,x_k)$ = min (min(|1-0.0|, |1-0.6|, |1-0.5|, |1-0.8|, |1-1|)) + 1 * max (max(|1-0.0|, |1-0.6|, |1-0.5|, |1-0.8|, |1-1|)) =1, k= 1, 2, 3, 4, 5

Using the gray relations found above, the gray relational grade (ie. value for r(x1, x1)) is found using equation 2: r(x1, x1) = (1+1+1+1+1)/5=1. Similarly the gray relation grade r(x1, x2) (ie. M12), r(x1, x3) (ie. M13), r(x1, x4) (ie. M14) are calculated. The grades or values that fill first row of matrix are: 1, 0.8093, 0.6393, and 0.5545. Now in order to calculate the second row of grade Matrix M, the reference factor is kept as x2 and x1, x3, x4 as comparative factors. After this the second row of matrix M (M21, M22, M23, M24) is calculated as explained previously. This process continues till the grade matrix M is filled completely.

Step 3: Calculate similarity matrix S from grade matrix M

Each cell of the 4x4 similarity matrix S is filled by using the following calculation that uses the value of the cells in grade matrix (M) found in previous step. $S_{ij=}$ 1-((M_{ij}-$M_{ii/}M_{ii}$) + (M_{ji}-$M_{jj/}M_{jj}$)), where i=1, 2..4 and j=1,2..4. For our example, the similarity matrix is:

$$S = \begin{bmatrix} 1 & 0.81814 & 0.67681 & 0.59596 \\ 0.81814 & 1 & 0.66955 & 0.61395 \\ 0.67681 & 0.88955 & 1 & 0.70741 \\ 0.59596 & 0.61395 & 0.70741 & 1 \end{bmatrix}.$$

Step 4: Clustering of stable items into stable patterns using similarity matrix

Each row in S, represents the factors of a stable item. For example, first row in matrix S, represents stable item X1 and so on. If all the values in the row representing the stable item have values greater than the input confidence level (ie. 0.5), then they belong to same cluster or same stable pattern. In our example, the final stable pattern will be {X1, X2, X3, X4}, since the rows corresponding to each item in S have values greater than 0.5. Hence, we have arrived at our stable patterns. Though this algorithm does not generate exact frequent patterns from multiple databases and is not designed to mine multiple related tables that the proposed algorithm is able to do, it gives a new concept of stable patterns and uses a mathematical approach to cluster the most stable patterns in multiple databases. This method is useful in situatuons where the store needs to find its constant pattern of visitors/buyers from multiple stores, to give special offers to them.

2.3.5 Identifying relevant databases

In some application areas where there is a very large set of databases and the user needs to find specific databases from the entire set of databases, in order to query specific information, there is a need to first find the specific databases that can be queried later, to find information that the user is looking for. For example, if the query is to find the eating habits of people belonging to Sri Lanka and we are given a huge dataset of 1000 databases, then we must first find the relavant databases that can be queried to find the answer. One prominent existing work to solve this type of problem was given by Liu, Lu and Yao (2001). They introduced a mathematical factor known as relevance factor, which identifies how close the attribute in a database is to the given query. For example, if the query is find the eating habits of people belonging to Sri Lanka and the database being checked has attributes ‘habits’ and ‘Country’, the relavance factor will be high for that database (close

to 1), meaning this database can be chosen to be queried in order to find the answer for the input query. The following example shows the process of identifying relevant databases.

Input: Set of databases (Figure 19 and Figure 20) and query Q (Chinese=Rice), treshold=0.1.

Output: Relevant databases

ID	ethnic-group	alcohol	on-diet	snack-between-meals	favorite-non-veg
950351	Chinese	never	no	sometimes	fish
950301	Chinese	never	no	seldom	pork
950282	Chinese	sometimes	no	seldom	fish
940112	Chinese	often	no	often	chicken
940023	Chinese	sometimes	no	sometimes	beef
938976	Chinese	sometimes	no	sometimes	egg
950612	American	never	no	seldom	beef
950122	American	often	no	sometimes	beef
940227	American	sometimes	no	sometimes	chicken
938567	American	sometimes	no	often	pork
950348	Indian	often	no	sometimes	fish
950312	Indian	sometimes	no	seldom	pork
950123	Indian	never	no	sometimes	chicken
940247	Indian	sometimes	no	sometimes	fish
940100	Indian	never	no	sometimes	beef

Figure 19: Database 1

ID	ethnic-Group	main-food	regular eating times	drink	vegetarian
950578	Chinese	rice	3	tea	no
950351	Chinese	rice	3	tea	no
950301	Chinese	rice	3	tea	no
950282	Chinese	rice	3	tea	no
940226	Chinese	rice	3	cola	no
940112	Chinese	rice	3	tea	yes
950612	American	bread	3	coffee	no
950122	American	bread	3	cola	no
940227	American	rice	3	cola	yes
940121	American	bread	3	tea	no
950348	Indian	bread	3	coffee	no
950312	Indian	bread	3	coffee	yes
950123	Indian	rice	3	cola	no
940247	Indian	rice	3	coffee	no
940109	Indian	bread	3	tea	no

Figure 20: Database 2

Step 1: Calculation of Relavance Factor (RF)

The given query Q is ‘Chinese=Rice’, meaning we need to find the relavant database that has information related the input query. A selector ‘s’ is of the form ‘A relop C’ where, A-Attribute, Relop-Relational operators (=,<,> etc.) and C is a constant or values of the attribute. Note that the query Q is also of selector format. The value of RF:

RF(s, Q) =Val(s|Q) * Val(Q)*log Val(s|Q)/Val(s), where, Val(Q) and Val(s) are the number of times that Q and s appear in a database and Val(s|Q) is the number of times ‘s’ appears in database given Q is true. For example, if selector s is ‘alcohol=never’, then we can see that in Figure 19, ‘s’ appears 5 times (ie. Val(s)=5), Q appears 0 times (ie. Val(Q)=0) and Val(s|Q)=2/6=number of times ‘s’ is true / number of times attribute A of input query Q appears (ie. group=chinese) , using these values RF(s,Q) for the given selector is 0, meaning the selector is irrelavant to the database. If a selector has a value greater than or equal to input treshold (ie. 0.1), then it is relavant to database and if the database consists of atleast one selector then it is relavant to the input query.

Step 2: Finding relavant databases: For our example, the calculated RF values for each selector (ie. each Attribute and its values of the database) for database 1 (Figure 19) and database 2 (Figure 20) is shown in Figure 21 and Figure 22. From the Figure 22 we can see that, only database 2 is relavant to the input query since it contains 2 selectors. Hence the final output will be database 2 (Figure 20) as relavant database to find information related to input query ‘Chinese=Rice’.

No	Selector s	$RF(s,Q)$
1	alcohol=never	0.000000
2	alcohol=sometimes	0.019907
3	alcohol=often	-0.017536
4	on diet=no	0.000000
5	snacks between meal=sometimes	-0.052607
6	snacks between meal=seldom	0.042924
7	snacks between meal=often	0.021462
8	favorite non-veg=fish	0.042924
9	favorite non-veg=pork	-0.017536
10	favorite non-veg=chicken	-0.017536
11	favorite non-veg=beef	-0.045205
12	favorite non-veg=egg	**0.088129**

Figure 21: RF values of Db1

No	Selector s	$RF(s,Q)$
1	main food=rice	**0.294786**
2	regular eating times=3	0.000000
3	drink=tea	**0.278834**
4	drink=cola	-0.045205
5	vegetarian=no	0.019631
6	vegetarian=yes	-0.017536

Figure 22: RF values of Db2

This paper was among the very first ones to mine the relavant databases from a large set of databases, that can answer a given query. This paper is important in field of mining

multiple databases, since before being able to mine the frequent patterns from database to answer our queries, the user must be able to chose the correct database to be mined.

2.3.6 Mining multiple databases using pipeline feedback technique

The pipeline feedback technique by Adhikari.A et al (2010) aims to give an efficient methodology to mining large sets of local databases, rather than give an algorithm to mine the frequent patterns. The main concept is to sort the databases based on decreasing order of their sizes and then mine each database using a standard frequent itemset mining algorithm such as FP tree algorithm (Han et al (2001)) and finally synthesize the large collection of patterns obtained.

The following example describes the process in detail.

Input: Multiple transaction databases (Figure 23), support=2

Output: synthesized frequent pattern

Db1:

tid	items
1	A,B
2	B,C,D
3	A,C

Db2:

tid	items
1	D
2	C,D
3	D,E

Figure 23: Multiple databases (Db1,Db2)

Step 1: Order databases in descending order of their size. Since both db1 and db2 have same size '3' (ie. number of transaction records), they are already in order.

Step 2: For the first database, db1, use a standard single table mining algorithm such as Fp tree algorithm (Han et al (2001)). Pass the result pattern set obtained (ie. all itemset having support>= input support (ie. 2), A, B, C) to the next database, db2. Now mine db2 using same standard single table mining algorithm and append the result set obtained with previous result set and send it to next database. This process continues until there are no

more databases to be mined. In our example, the final result set consists of items A, B, C, D.

Step 3: Synthesize the global resultset obtained in step 2. It is clear that all the itemsets that are repeating in resultset occur in more than one databases. In order to find whether such itemset X are highly frequent on a global scale, the following equation is used to find global support count for that itemset X:

(1 / (sum of size of all dbs)) * ($\sum_1^m$ support count of X in db 'm' * size of db), where 1<=m<=n is the number of databases in which X was frequent. In our example, only item C appears more than once and global support count (C) = 1/6 * (2*3 + 2*3) = 2. Since C appears 2 times in global result set, it is a synthesized global frequent itemset.

2.3.7 Cluestering local frequent items in multiple databases

Clustering local frequency items in multiple databases by Animesh.A (2013), gives a similarity measure that is derived from the common items in multiple databases, to cluster the frequent itemsets obtained in each local database. We shall see a detailed example of the algorithm.

Input: local transaction db1 and local transaction db2 (Figure 24), support count = 0.5.

Output: local clusters of frequent itemsets in each local database

tid	Items
1	(abc)
2	(d)
3	(bc)
4	(abc)

tid	Items
1	(ab)
2	(abc)
3	(ed)
4	(abc)

Figure 24: local db1 and local db2

Step 1: Calculate similarity value:

The similarity value is calculated using:

S (Local db1, Local db2) = |I (Db1) ∩ I (Db2)| / |I (Db1) U I (Db2)|, where I (Db1) ∩ I (Db2) is the unique number of items that are both in local db1 and local db2 in Figure 24. I (Db1) U I (Db2) is the total number of unique items in both local databases in Figure 24. In our example, items a, b, c are present in both local databases and the total number of unique items in both databases is a, b, c, d, e. Hence, S (Local db1, Local db2) = 3/6=0.5.

Step 2: Find local frequent itemsets in each local database

Using a standard frequent itemset mining approach such as FP tree algorithm (Han et al (2001)), the frequent itemsets are mined from local database 1 and local database 2 in Figure 24. The frequent itemsets in each local database with their support counts that are >= the input support count 0.5 are:

Local db1: a: 0.5, b:0.7, c:0.7, bc:0.7, ab: 0.5, ac:0.5, abc:0.5

Local db2: a: 0.7, b:0.7, c:0.5, bc:0.5, ab: 0.7, ac:0.5, abc:0.5

Step 3: Find the local clusters in each local database

All the frequent itemsets in each local database whose support count value matches the similarity value (ie. 0.5), computed in step 1 belong to same local cluster. For example, itemsets a, ab, ac, abc belong to same cluster in local db1 (since, their support count=similarity value). Similarly, itemsets c, bc, ac, abc belong to same local cluster in local db2. Hence, the final local cluster result sets of each local database in Figure 24 are:

Local cluster in local db1: itemsets a, ab, ac, abc.

Local cluster in local db2: itemsets c, bc, ac, abc.

This method can be used in finding the similar frequent itemsets in each local table belonging to each local branch of a retail store.

CHAPTER 3 : PROPOSED TIDFSEQ ALGORITHM

There are many important reasons for mining frequent patterns from multiple databases. In applications such as E-Commerce sites, the historical information of products purchased online for each online store are constantly stored in multiple databases. Historical queries can be answered by mining such multiple product databases, such as: Find the frequent laptops purchased during the year 2010 in Bestbuy and Amazon, find all the products from Bestbuy, Amazon, Walmart that have discounts associated with them during the September month, find the frequent brand of laptop products that have been purchased in Bestbuty, Amazon and Walmart for five consecutive years. There is a need to find the frequent local and global patterns of product purchased from customer transaction databases of retail stores having same structure (ie. local customer transaction databases of a retail store such as Walmart having customer id and the products purchased by the customer in each of its local database). Queries useful for the retail store (such as Walmart having many local branches) can be answered using the local and global frequent patterns of prodcut purchased, such as: Find the global frequent patterns that consist of milk products in them, find the frequent local product patterns that are also frequent on global scale, find the frequent pattern of products purchased in branches located in the western region of US. Not all multiple tables have the same structure, there is also a need to mine frequent itemsets/sequences from multiple related databases having different structure (ie. Patient/Drugs and Drugs/side-effects that are related by the foreign/primary key attribute: Drugs) to answer queries related to the multiple tables. For example, in an application having related sequence databases such as Patient/Drugs and Drugs/side-effects sequence databases, the mined patterns from the related sequence tables can help answer queries such as: Find the frequent sequence of side effects that patients p1 and p3 suffer from, find all the patients that are affected by frequent sequence of side effects that have side effect ‘s1’ in their pattern. In applications such as E-Commerce sites, the information of products (such as product name, price) sold by each online store (such as Bestbuy) are constantly stored in multiple databases. For example, the laptop products sold in Bestbuy and Walmart are stored in two product databases corresponding to Bestbuy and Walmart respectively. By mining such multiple databases, apart from answering historical queries, comparative queries can also be answered, such as: Find the frequent set of product patterns that contain

‘Apple’ products in the pattern, from the product information tables of E-commerce websites that sell ‘Apple’ products with discount, find the set of frequently bought laptop product names from the product information tables corresponding to the E-commerce websites that provide highest discounts, find the E-commerce website which sells the cheapest ‘samsung’ TV products.

Existing systems in mining frequent patterns from multiple databases can be categorized into: Mining frequent patterns from multiple tables having same structure (for eg, local customer transaction databases of a retail store such as Walmart having customer id and the products purchased by the customer in each of its local database) and mining frequent patterns from multiple tables having different structure (for eg, Patient/Drugs and Drugs/side-effects that are related by the foreign/primary key attribute: Drugs). Some existing systems that mine frequent patterns from multiple databases having same structure are: ApproxMAP algorithm by Hyue and *et al*. (2006), the ApproxMap algorithm breaks its input sequences (e.g., the 2-column sequence < (123) (45) >) into columns so it can find the collection of approximate frequent sequences of all the columns as the approximate sequence of the database. The same method is applied on the local frequent sequences obtained from each local cutomer transaction database that must have the same table structure, to get the frequent global approximate sequence patterns. The main limitation of this algorithm is that it does not generate exact frequent sequence patterns and does not work for multiple tables with different structure. The hierarchial Gray clustering algorithm by Yaojin and et al (2013), introduced a new concept of stable patterns. It defines an item ‘a’ as stable, if the item satisfies the minimum support count ‘s’ in each of the local transaction table (T1,T2..TN, where Ti, 1<=i<=N, is a local transaction table) that it occurs and also the variation of the support count of that item ‘a’ is less than or equal to a user defined variation value ‘v’. The algorithm clusters stable items found from multiple transaction tables into stable patterns according to the similarity of their timestamps (ie. the time at which the stable item ‘a’ was purchased by a cutomer). The use of stable patterns helps us to identify the standard set of constantly purchased items on a global scale in a retail store having local branches. Since this algorithm is specific to mining stable patterns, it is not useful to mine frequent sequences from multiple tables with different structures. The kernel estimation algorithm by Zhang and *et al* (2009), introduced the concept of

generating the local and global important cutomers (ie. customers who regularly purchase high value products from the store) from a retail store having local branches. The local databases contain cutomer information, such as the expenditure of the customer in that local branch and the number of visits and transactions made by the customer in that local branch. Using this information the kernel estimation algorithm uses a mathematic approach to first find the local importance of each customer to the local branch. And then the global importance of the customer to the retail store. Since this algorithm is specific to mining local and global important customers, it is not useful to mine frequent sequences from multiple tables with different structures. Some existing systems that mine frequent patterns from multiple databases having different structure are: TidFP algorithm by Ezeife and Zhang (2009).The TidFp algorithm mines frequent item_sets from multiple sources using transaction ids for integrating patterns through set operations (e.g., intersect, union) in order to answer global queries involving multiple sources. But the main limitation is that it only mines frequent itemsets from multiple transaction tables and not frequent sequences. It is clear that none of the existing systems are able to mine multiple related sequence tables having different structure for frequent sequences and answer queries using the result sets. The main purpose of this thesis is to develop a multiple related sequence database mining algorithm to mine the frequent sequence patterns and answer valuable user queries. The proposed TidFSeq algorithm belongs to the category of mining multiple tables having different structures, the proposed algorithm mines multiple related sequence tables having different structure and answers queries related to the multiple tables (ie. The proposed algorithm answers queries such as: What are the frequent side effects affecting patients p1 and p3 in Drug/sequence of side-effects and Patient/Drugs sequence tables).

Problem Definition :

Given multiple related sequence tables where each table consists of sequence id and corresponding sequence of items and a minimum support count ‘s’, the problem of mining frequent sequences from multiple related sequence databases is to mine the exact frequent sequences with support counts greater or equal to the given minimum support count ‘s’ from each sequence table to be able to answer complex queries on the multiple related sequence tables. For example, what are the frequent side effects that affect the patients p1

and p3? , is a query that can be answered by the proposed algorithm from Patient/sequence of drugs table and Drug/sequence of side effects table.

3.1 Problems Addressed

The proposed algorithm extends the TidFp algorithm by Ezeife, C.I. and Zhang, D (2009). The TidFp algorithm mines multiple related transaction tables to generate exact frequent itemsets in order to answer queries for multiple related transaction tables. The proposed TidFSeq algorithm extends the problem of mining itemsets to the problem of mining sequences. The challenges with mining frequent sequences from MDBs that is solved by this thesis are that the TidFSeq algorithm first computes the element (ie. Sequence item position id) in which each item in each sequence (ie. sequence id) occurs by replacing the <frequent items, transaction id list> tuple used in the TidFp with a <sequence id, position id list> tuple. For every item 'i' in the kth sequence of n-sequence of length 'n', the TidFSeq algorithm first transforms it into a tuple that specifies (a) it's transaction id (Tid) and (b) the list of the kth sequence in this transaction that item 'i' occurs (called it's position id list). Next the GSP-like candidate generation approach is used on our transformed sequences to generate frequent sequences with transacion ids which can be used to answer complex queries from MDB's through set operations.

3.2 Definitions used in proposed algorithm

The following concepts and definitions are used in the proposed algorithm:

Definition 1: *Elements in a sequence*

Sequence elements are defined as the set of events or items that appear together in a sequence. Consider the sequence < (d1 d2 d3) (d1) (d4) >. This sequence consists of 3 elements, the first is (d1 d2 d3) which has 3 items, the second element is (d1) which has just one item, the third element is (d4) which also has 1 item in it. The first element is named as 'e1, second element is 'e2, likewise third element is 'e3'.

Definition 2 *Position id (pid)*

The position id is the sequence element in which the item occurs. Consider the sequence < (d1 d2 d3) (d1) (d4) >. Item d1 in this sequence is found in the first element (d1 d2 d3) and

also in the second element (d1). Item ‘d1’ in this sequence is said to be in positions e1 and e2. Similarly, the position of item ‘d3’ is ‘e1’, since it is found only in the first element (d1 d2 d3).

Definition 3 Description of the <Sequence id, Position_id list> tuple

Each item/ sequence is associated with the sequence ids in which they occur and also the position ids in which they occur in each sequence id. This information is represented in form of a tuple associated with each item/sequence. If an item ‘1’ appears in sequence ids sid1 (say the position ids of item 1 in sid 1 are e1, e3) and sid3 (say the position ids of item 1 in sid 3 are e1, e2). Then <sid, pid_list> tuple for item 1: < (sid1, (e1, e3)), (sid3, (e1, e2)) >.

Definition 4 Description of I-step and S-step sequences

The following terms were first defined by Ayres and et al (2002):

I-Step sequence: A sequence of form (ab) is known as a I-step sequence. Example: (1 2 3), meaning items 1, 2 and 3 occur together in a sequence (ie. items 1,2 and 3 are purchased during same transaction). They are itemset sequences.

S-Step sequence: A sequence of form (a) (b) is known as a S-Step sequence. Example: (1) (2) (3), meaning items 1, 2 and 3 occur separately in a sequence (ie. items 1,2 and 3 are purchased in different transactions). They are multi set sequences.

3.3 The TidFseq Algorithm

In this section we give the formal representation of the TidFSeq algorithm in Figure 25 (TidFseq algorithm), Figure 26 (I-step pruning) and Figure 27 (S-step pruning) and detailed explanations for the algorithms.

Algorithm 1 Algorithm TidFSeq ()

Input: Multiple related sequence databases $MDB_{i,}$ i=1,2..n and their corresponding candidate item sets $C_1^{i,}$ i=1,2…n , User-defined min-support count 's'.

Output: Frequent sequences and their associated sequence ids < (seq1,sid1….sidn), (seq2,sid1…sidn) (seqn,sid1…sidn)>

Other variables: C_{k-} Candidate –k sequences, F_k- Frequent -k sequences, k=1 initially, Fp-final list of frequent sequences.

Begin

For each MDB_i, i=1,2…$n_:$

(1) Scan sequence db to transform candidate itemset C_1^i:
C_k={$item_{1:}$<$sid_{1,}$ pid_list_1>, $item_{2:}$ <$sid_{2,}$ pid_list_2>…….$item_{n:}$ <$sid_{n,}$ pid_list_n>}, where k=1 and $item_{1,}$ $item_{2...}item_n$ ϵ C_1^i, i=1,2..n

(2) For each candidate-1 item in C_1^i:
2.1 If (sid count of candidate-1 item>= s) then

2.2 candidate-1 item ϵ $F_{k,}$ where k=1

End for-loop

(3) C_{k+1} sequence: F_k (GSP_gen join) $F_{k,}$ k=2,3..n

(4) while ($C_k \neq Ø$) do
4.1 If (candidate sequence is I-step sequence of the form (a b)) then

4.2 Call I-step pruning (); (Algorithm 2 in Figure 26)

4.3 elseif (candidate sequence is a S-step sequence of the form (a) (b)) then

4.4 Call S-step pruning (); (Algorithm 3 in Figure 27)

End while-loop

(5) if ($F_k \neq Ø$) then goto step 3
(6) Fp=F1 U F2 U …..Fk.

End for-loop
End

Figure 25: TidFSeq algorithm

Step by step Explanation of the TidFseq algorithm (Figure 25):

Inputs to the algorithm: Multiple related sequence databases, their corresponding candidate item sets and the input user defined support count 's' (For example, input Drug/sequence of side-effects (MDB1), 1-candidate item set for MDB1: C_1^1={1, 2, 3, 4}, Patient/Drugs sequence tables (MDB2), 1-candidate item set for MDB2: C_1^2={d1, d2, d3, d4} and user-defined support count=2).

Step 1: Transform the 1-candidate item set for MDB_i into < Sequence id, Position id list > tuple format so we can compute frequency of each candidate -1 item: Scan the input sequence MDB_i once. For each item in 1-candidate item set, the sequence ids and the positions in which the item is found in each sequence, is recorded as a set of tuples and each tuple is of the form <Sequence id , Position_id list>.

Step 2: Find frequent-k sequence for k=1: In the transformed 1-candidate item set obtained from previous step, if the count of sequence id sets of the candidate 1 items are greater than or equal to minimum support count, then they are frequent 1 items (ie. F_k, where k=1).

Step 3: Candidate-k+1 sequence generation using GSP join: Next, k+1 candidate sequences are generated (using GSP candidate generation (Srikant and Agrawal (1996))) from the Frequent k sequences. The candidate sequences are found using the GSP gen join. The F_k (GSP gen join) F_k, will generate all possible combinations of itemset sequences (ie. sequences of the form (231)) and also all possible combinations of multi set sequences (ie. sequences of the form (2) (3)) between the sequences that are being joined. For example, when items (1, 2) do (GSP gen join) with (1, 2). The possible itemset sequences are (11), (12), (21), (22). The possible multi set sequences are (1)(1), (1)(2), (2)(1), (2)(2).

Step 4: Finding k+1 frequent itemset and multi set sequences from Candidate-k sequences: After the candidate generation step, if the candidate sequence is of the form (a b) ((ie.) items a and b are purchased together in same transaction in a retail store by a customer), then it is an I-step sequence and I-step pruning algorithm (Figure 26) is called.

Explanation of **Algorithm 2**: I-step pruning ()

Input: The I-step candidate sequences of the form (ab) generated by candidate generation method of the main algorithm (ie. Algorithm 1) and min-support count.

Algorithm 2: I-step pruning ()

Input: I-step candidate sequences C_k, k=2, 3…n, user-defined min-support count 's'.

Output: Result list containing frequent I-step sequences and their corresponding

< Sequence id, Position id list > tuple.

Other variables: support- indicating the support count of the I-step candidate sequence, Sida and Sidb - sequence ids of items 'a' and 'b', Pid_a and Pid_b - Position_id list of items 'a' and 'b'.

Begin

1) Compute support count for I-step sequence (a b):
   1.1 Sid (a b) = (Sida ∩ Sidb), where Sid_a=Sid_b
   1.2 For each matching sid in Sid (a b):
   1.3 if (Pid_a = Pid_b) then
   1.4 support=support+1;
   End for-loop
2) If (support >= s) then
   2.1 I-step sequence is frequent
   2.2 Add Frequent I-step sequence to Result list.
   Else Return null.

Return Result list.

End

Figure 26: I-step pruning ()

Step 1: Compute support count for the I-step candidate sequence of form (ab): This is done using the following rule: If items 'a' and 'b' have same sequence ids and occupy the same positions (same position ids), then support count is incremented. For example, say there is a candidate I-step sequence (1 2) and the < Sequence id, Position_id list > tuples for item 1: <(sid1,pos1) (sid2,pos1)> (ie. item 1 occurs in sid1 at position id: pos1 and also occurs in sid2 at position id: pos1) and tuples for item 2: <(sid1,pos1) (sid2, (pos1,pos4))> (ie. item 2 occurs in sid1 at position id: pos1 and also occurs in sid2 at position ids: pos1, pos4).

From the tuples, we can see that item 1 and item 2 have a total of two matching sequence ids (ie. sid 1 and sid2) and also the corresponding position ids are matching (ie. items 1 and 2 occur at pos 1 in sid1 and again occur at pos1 in sid2). The < Sequence id, Position id list> tuple for I-step sequence (1 2): <(sid1,pos1) (sid2,pos1)> and the support count of the I-step sequence is 2.

Step 2: Checking whether the I-step sequence is frequent: Next, the support count computed in previous step is checked whether it satisfies the minimum support count (ie. if the support count of I-step sequence is greater than or equal to minimum support count). If the minimum support count is 2, then the I-step sequence (1 2), whose support count calculated in previous step, which is 2, satisfies the minimum support count and is frequent.

Output of I-step pruning (): The frequent sequence and it's < Sequence id, Position_id list > tuples will be added to the result set that will be returned to the main program (Algorithm 1 shown in Figure 25), else if not frequent null value is returned, indicating that the I-step sequence is not frequent.

If the candidate sequence is of the form (a) (b) (ie. items a and b are purchased during different transactions in a retail store by a customer), then it is an S-step sequence and S-step pruning algorithm (Algorithm 3 in Figure 27) is called.

Explanation of **Algorithm 3**: S-step pruning ()

Input: The S-step candidate sequences of the form (a) (b) generated by candidate generation method of the main algorithm (ie. Algorithm 1) and min-support count.

Step 1: Compute support count for the S-step candidate sequence of form (a)(b): This is done using the following rule: If items 'a' and 'b' must have same sequence ids and'a' occupies earlier position than that of 'b' (ie. position id of a is lesser than position id of b), then support count is incremented. For example, say there is an S-step candidate sequence (1) (2) and the < Sequence id, Position_id list > tuples for item 1: <(sid1,pos1) (sid2,pos2)> (ie. item 1 occurs in sid1 at position id: pos1 and also occurs in sid2 at position id: pos2) and tuples for item 2: <(sid1,pos3) (sid2, (pos1,pos4))> (ie. item 2 occurs in sid1 at position id: pos3 and also occurs in sid2 at position ids: pos1, pos4). From the tuples, we can see that item 1 and item 2 have a total of two matching sequence ids (ie. sid 1 and sid2) and

also the corresponding position ids of item 1 are lesser than the corresponding position ids of item 2 (ie. the position id (pos1) of item 1 in sid1 is lesser than position id (pos3) of item 2 in sid1 and the position id (pos2) of item 1 in sid2 is lesser than position id (pos4) of item 2 in sid2). The tuples for S-step sequence (1) (2): <(sid1, (pos1, pos3)) ((sid2, (pos2, pos4))> and the support count of the S-step sequence is 2.

Algorithm 3: S-step pruning ()

Input: S-step candidate sequences Ck, k=2, 3…n, user-defined min-support count 's'.

Output: Result list containing frequent S-step sequences and their corresponding < Sequence id, Position id list > tuple

Other variables: support- indicating the support count of the S-step candidate sequence, Sida and Sidb - sequence ids of items 'a' and 'b', Pid_a and Pid_b - Position_id list of items 'a' and 'b'.

Begin

1) Compute support count for S-step sequence (a) (b):

1.1 Sid (a b) = (Sida ∩ Sidb), where Sid_a=Sid_b

1.2 For each matching sid in Sid (a b):

1.3 if (Pid_a < Pid_b) then

1.4 support=support+1;

End for-loop

2) If (support >= s) then

2.1 S-step sequence is frequent

2.2 Add Frequent S-step sequence to Result list.

Else Return null.

Return Result list.

End

Figure 27: S-step pruning ()

Step 2: Checking whether the S-step sequence is frequent: Next, the support count computed in previous step is checked whether it satisfies the minimum support count (ie. if the support count of S-step sequence is greater than or equal to minimum support count).

If the minimum support count is 2, then the S-step sequence (1) (2), whose support count calculated in previous step, which is 2, satisfies the minimum support count and is frequent.

Output of S-step pruning(): The frequent sequence and it's < Sequence id, Position_id list > tuples will be added to the result set that will be returned to the main program (Algorithm 1 shown in Figure 25), else if not frequent null value is returned, indicating that the S-step sequence is not frequent.

Step 5: Finding higher order frequent –k sequences: After, I-step and S-step pruning algorithms return to main program, steps 3 and 4 are called recursively until no more candidates can be generated. Ie. using the rules for I-step and S-step pruning as discussed above, every time candidate sequences are generated the infrequent sequences are pruned retaining the frequent sequences and their corresponding < Sequence id, Position_id list > tuples. This process keeps repeating until no more candidates can be generated.

Step 6: Finding final set of frequent sequences: The final output will be the union of all frequent sequences and their corresponding sequence ids obtained till now. For example, If the inputs are: Drugs/sequence of side effects table and Patient/sequence of drugs purchased table, then final output from Drugs/sequence of side effects table will be, frequent sequence of side effects with their corresponding drugs and final output from Patient/sequence of drugs purchased table will be frequent sequence of drugs with their corresponding patients. This final result set is used to answer user queries for the input related sequence tables such as what are the frequent sequence of side effects that affects patients p1 and p3? We shall see the detailed explanation of the algorithm and how the final output is used to answer queries in the next subsection.

3.4 Example Application of TidFSeq Algorithm

Input: Multiple related sequence tables, MDB_1 (Table 24 (Drug/sequence of side-effects)), 1-candidate item set for MDB_1: C_1^1= {1, 2, 3, 4}, MDB_2 (Table 25 (Patient/sequence of drugs purchased)), 1-candidate item set for MDB_2: C_1^2= {d1, d2, d3, d4} and user-defined support count=2

Output: Frequent sequences and their associated sequence ids.

Drug	Sequence of side effects
d1	<(1 2 3) (1)>
d2	<(1 2 3)>
d3	<(3)(4)>
d4	<(1)(3)>

Table 24: Drug/side-effects

Patient	Sequence of Drugs Purchased by patient
p1	<(d1 d2 d3)>
p2	<(d3)(d4)>
p3	<(d1 d2)(d1)(d3)>
p4	<(d1)(d3)>

Table 25: Patient/Drugs

For each MDB_i , with i=1, do the following steps:

Begin

Step 1: Transform the 1-candidate item set for MDB_i into < Sequence id, Position id list > tuple format so we can compute frequency of each candidate -1 item:

Scan the input sequence MDB_i once. For each item in 1-candidate item set, the sequence ids and the positions in which the item is found in each sequence, is recorded as a set of tuples and each tuple is of the form <Sequence id , Position_id list>. For example, in MDB_1 (Table 24), the 1-candidate list C_1^1:{1, 2, 3, 4}, will be transformed to $C_1^{1:}$ {1: <(d1,(e1,e2)), (d2,(e1)), (d4,(e1))>, 2:< (d1,(e1)), (d2,(e1))>, 3:< (d1,(e1)), (d2,(e1)), (d3,(e1)), (d4,(e2))>, 4:<(d3,(e2))>}. This means that the first candidate item '1', in the 1-candidate set, occurs in sequence ids d1, d2, d4 and in sequence attribute columns 1 and 2

referred to as elements 1 and 2 (ie. e1 and e2). Similarly, the tuples for other candidate 1 items have the sequence ids in which the item occurs and the corresponding sequence attribute columns (ie. position ids). The tuples are shown in Table 26 and Table 27, where table 26 consists of tuples corresponding to Table 24 (MDB_1) and table 27 consists of tuples corresponding to Table 25 (MDB_2). For example, in Table 26, we can see that item '1' has a tuple of sequence ids (ie. d1, d2 and d4) and the corresponding position ids for each sequence id (ie. pid e1, e2 in sid d1, pid e1 in sid d2, pid e1 in sid d4). Note that each itemset in a sequence such as <(123)(1)> represents a column. Thus, the sequence <(123)(1)> has two columns, where itemset (123) belongs to column 1 (or element 1) and itemset (1) belongs to column 2 (or element 2). ***Thus, the transformed 1-candidate itemset list for MDB_1 and MDB_2:***

C_1^1:{1, 2, 3, 4}, will be transformed to $C_1^{1:}$ {1: <(d1,(e1,e2)), (d2,(e1)), (d4,(e1))>, 2:< (d1,(e1)), (d2,(e1))>, 3:< (d1,(e1)), (d2,(e1)), (d3,(e1)), (d4,(e2))>, 4:<(d3,(e2))>}.

C_1^2:{d1, d2, d3, d4}, will be transformed to $C_1^{2:}$ {d1: <(p1,(e1)), (p3,(e1,e2)), (p4,(e1))>, d2:< (p1,(e1)), (p3,(e1))>, d3:< (p1,(e1)), (p2,(e1)), (p3,(e3)), (p4,(e2))>, d4:<(p2,(e2))>}.

1		2		3		4	
Sid	Pid_list	Sid	Pid_list	Sid	Pid_list	Sid	Pid_list
d1	e1 e2	d1	e1	d1	e1	d3	e1
d2	e1	d2	e1	d2	e1		
d4	e1			d3	e1		
				d4	e1		

Table 26: Transformed 1-candidate item sequences for MDB_1

d1		d2		d3		d4	
Sid	Pid_list	Sid	Pid_list	Sid	Pid_list	Sid	Pid_list
p1	e1	p1	e1	p1	e1	p2	e2
P3	e1 e2	P3	e1	P2	e1		
p4	e1			p3	e3		
				p4	e2		

Table 27: Transformed 1-candidate item sequences for MDB_2

Step 2: Find frequent-k sequence for k=1

In the transformed 1-candidate item sequences obtained from previous step, if the count of sequence id sets of the candidate 1 items are greater than or equal to minimum support count (ie. 2), then they are frequent 1 items. For example, in Table 26 (ie. transformed 1-candidate item sequences for MDB_1), item '4' is infrequent since it appears only once in sid 'd3', but the min-support count is 2, thus item '4' is pruned since it does not satisfy support count. Similarly, in Table 27 (ie. transformed 1-candidate item sequences for MDB_2), item 'd4' is infrequent since it appears only once in sid 'p2', but the min-support count is 2, thus item 'd4' is pruned since it does not satisfy support count. ***Thus the F_k sequences (k=1), for inputs MDB_1 and MDB_2 are F_1^1={1, 2, 3} and F_1^2= {d1, d2, d3} respectively.***

Step 3: Finding candidate-2 sequences from frequent-1 sequences (finding Ck+1 sequences from Fk sequences)

Next, k+1 candidate sequences (ie. k+1=2) are generated (using GSP candidate generation (Srikant and Agrawal (1996))) from the Frequent k sequences (ie. k=1). The candidate sequences are found using the GSP gen join. In order to obtain the k-sequence candidates (Ck+1), the frequent sequences found in previous step (Fk), joins with itself in an Apriori-gen way. ***The following rule is used to generate candidate-k sequences of length 3 or more (ie. k>=3):***

The rule is that every sequence - s in Fk joins with other sequences - s'in Fk, if the last elements of s (excluding the first element of s) is same as first elements of s' (excluding the last element of s'). For example, sequence s-<(1) (2) (3)> can join with sequence s'-<(2) (35)>, since the last elements of s (ie. (2) (3)) is same as first elements of s' (ie. (2)(3)). But in this step since, only candidate -2 sequences are being generated, which are candidate sequences of length 2, ***the following rule is used to generate the candidate -2 sequences (ie. k<3):***

The F_k (GSP gen join) F_k, will generate all possible combinations of itemset sequences (ie. sequences of the form (231)) and also all possible combinations of multi set sequences (ie. sequences of the form (2) (3)) between the sequences that are being joined. For example,

for MDB_1, the frequent k sequences (where k=1), obtained in previous step are: {1, 2, 3}. The GSP-gen join of items (1, 2, 3) with (1, 2, 3) will generate all possible itemset sequences of item 1 with items {1,2,3} (ie. (11), (12), (13)), all possible itemset sequences of item 2 with items {1,2,3} (ie. (21), (22), (23)) and all possible itemset sequences of item 3 with items {1,2,3} (ie. (31), (32), (33)).

The GSP-gen join of items (1, 2, 3) with (1, 2, 3) will also generate all possible multi set sequences of item 1 with items {1,2,3} (ie. (1)(1), (1)(2), (1)(3)), all possible multi set sequences of item 2 with items {1,2,3} (ie. (2)(1), (2)(2), (2)(3)) and all possible multi set sequences of item 3 with items {1,2,3} (ie. (3)(1), (3)(2), (3)(3)). Using the rules for GSP join, ***the candidate-2 sequences for the input sequence table, Table 24 (MDB_1):***

C_2^1= (11), (12), (13), (21), (22), (23), (31), (32), (33), (1)(1), (1)(2), (1)(3), (2)(1), (2)(2), (2)(3), (3)(1), (3)(2), (3)(3) and ***the candidate-2 sequences for the input sequence table, Table 25 (MDB_2):***

C_2^2= (d1)(d1), (d1)(d2), (d1)(d3), (d1d1), (d1d2), (d1d3), (d2)(d1) (d2)(d2), (d2)(d3), (d2d1), (d2d2), (d2d3), (d3)(d1), (d3)(d2), (d3)(d3), (d3d1), (d3d2), (d3d3).

Step 4: Finding frequent-2 itemset and multi set sequences from Candidate-2 sequences (Fk sequences from Ck sequences)

For each candidate 2 sequence obtained from step 3:

If the candidate sequence items occur together then it is an I-step sequence (itemset sequence) of the form (a b). The following condition is followed to check if the candidate sequence is frequent or not: if 'a' and 'b' have same sequence ids and occupy the same positions, in greater than or equal to min-support number of sequences, then I-step sequence of form (a b) is frequent. In the candidate 2 sequences generated in previous step for MDB_1, the candidate I-step sequence (1 2) has the following <Sequence id, Position_id list> tuples for item 1 and item 2: Item 1: <(d1, (e1, e2)), (d2, e1), (d4, e1)> (ie. item 1 occurs in sid: d1 at position id: e1, e2 and also occurs in sid: d2 and sid: d4 at position id: e1) and Item 2: <(d1, e1), (d2, e1)> (ie. item 2 occurs in sid: d1 at position id: e1 and also occurs in sid: d2 at position id: e1). From the tuples, it can be seen that, item 1 and item 2 have a total of two matching sequence ids and position ids (ie. item1 and item 2 occur at

position id: e1 in sid: d1 and also occur at position id: e1 in sid: d2). Since the condition for I-step sequence is met twice, the support count of sequence (1 2) is 2, which satisfies the input support count (2). Hence, I step sequence (1 2) is frequent sequence. ***Using the I-step pruning rule, the itemset candidate -2 sequences that are frequent in MDB_1 and MDB_2 are (ie. F_2^1 and F_2^2 itemset sequences of MDB_1 and MDB_2 respectively):***

F_2^1: (1 2), (1 3), (2 3) and F_2^2: (d1 d2)

If items of candidate sequence occur separately then it is a S-step sequence (multi set sequence) of the form (a) (b). The following condition is followed to check if the candidate sequence is frequent or not: if 'a' and 'b' have same sequence ids and 'a' occupies earlier position than that of 'b', in greater than or equal to min-support number of sequences, then S-step sequence of form (a)(b) is frequent. In the candidate 2 sequences generated in previous step for MDB_2, candidate S-step sequence (d1) (d3) has the following <Sequence id, Position_id list> tuples for item d1 and item d3: Item d1: <(p1, e1), (p3, (e1, e2)), (p4, e1)> (ie. item d1 occurs in sid: p1 at position id: e1 and occurs in sid: p3 at position ids: e1, e2 and also occurs in sid: p4 at position id: e1) and item d3: <(p1, e1), (p2, e1), (p3,e3), (p4, e2)> (ie. item d3 occurs in sid: p1, p2, p3, p4 at position ids: e1, e1, e3, e2 respectively). From the tuples, it can be seen that, items d1 and d3 have matching sequence ids p3 and p4. And for sid p3: corresponding pids of item d1 (ie. e1, e2) is less than the corresponding pid of item d3 (ie. e3) and similarly for sid p4: corresponding pid of item d1 (ie. e1) is less than corresponding pid of d3 (ie. e2). Since the condition for S-step sequence is met twice, the support count of sequence (1) (2) is 2, which satisfies the input support count (2). Hence, S-step sequence (1) (2) is a frequent sequence. ***Using the S-step pruning rule, the multi set candidate -2 sequences that are frequent in MDB_1 and MDB_2 are (ie. F_2^1 and F_2^2 multi set sequences of MDB_1 and MDB_2 respectively): F_2^1: null and F_2^2: (d1)(d3).*** Note that the frequent – 2 list for MDB_1 (F_2^1), does not contain any multi set frequent sequences, hence it is null. ***Hence, the complete frequent -2 lists for MDB_1 and MDB_2 are: F_2^1: (1 2), (1 3), (2 3) and F_2^2: (d1 d2) (d1)(d3)***

Step 5: Generating Candidate – 3 sequences from frequent – 2 sequences

The following rule is used to generate candidate-k sequences of length 3 or more (ie. k>=3): Every sequence - s in Fk joins with other sequences - s'in Fk, if the last elements

of s (excluding the first element of s) is same as first elements of s' (excluding the last element of s'). For example, the frequent -2 sequences (F_2^1***)*** in MDB_1 are: (1 2), (1 3), (2 3). The GSP-gen join of the (F_2^1***)*** list with itself will generate: (1 2 3), since, when frequent sequence (1 2) joins with frequent sequences (1 2), (1 3) and (2 3):

A. Sequence s- (1 2) cannot join with sequence s'-(1 2), since according to GSP-gen join rule, the last elements of s, excluding the first element of s (ie. item 2) must be same as first elements of s', excluding the last element of s' (ie. item 1). Since, they are not same, the GSP join does not occur.
B. Similarly, Sequence s- (1 2) cannot join with sequence s'-(1 3), since according to GSP-gen join rule, the last elements of s, excluding the first element of s (ie. item 2) must be same as first elements of s', excluding the last element of s' (ie. item 1). Since, they are not same, the GSP join does not occur.
C. Sequence s- (1 2) can join with sequence s'-(2 3), since according to GSP-gen join rule, the last elements of s, excluding the first element of s (ie. item 2) must be same as first elements of s', excluding the last element of s' (ie. item 2). Since, they are same, the GSP join does occur and the resulting candidate -3 sequence is (123).

The same technique is carried out for the other sequences in the join (ie. (1 3) and (2 3)), all the joins will yield the candidate -3 sequence (123), in this case. Using the GSP gen join rule discussed now, ***the candidate-3 sequences for the input sequence table, Table 24 (MDB_1):*** C_3^1= (123) and ***the candidate-3 sequences for the input sequence table, Table 25 (MDB_2):*** C_3^2= null. The candidate -3 sequences for MDB_2 is null, since, no more candidate sequences that follow the GSP-gen join rule could be generated.

Step 6: Finding frequent-3 itemset and multi set sequences from Candidate-3 sequences

The same rules for I-step and S-step pruning are applied on the candidate-3 sequences obtained for MDB_1 and MDB_2. The candidate -3 sequences for MDB_2 is null, since, no more candidate sequences that follow the GSP-gen join rule could be generated. Hence the only candidate -3 sequences to be checked for frequency is in C_3^1 (C-3 sequences for MDB_1). The candidate -3 sequence list for MDB_1 consists of one itemset sequence (123). The candidate itemset sequence (123) has the following <sequence id, position id_list> tuple: (123): <(d1, (e1)) (d2,(e1))>, meaning the itemset sequence is found in sequence ids

d1 and d2. And in the corresponding position ids e1 (in sid d1) and e1 (in sid d2) respectively. Since the number of sequence ids in which the sequence (123) occurs is 2 (ie. occurs in d1 and d2), it satisfies the minimum support count (ie.2). ***Hence the frequent – 3 sequences for MDB1 and MDB2 are: F_3^1: (123) and F_3^2: null***. The frequent -3 sequence list for MDB2 is null since, the candidate -3 sequence for MDB2 was also null (ie. no more candidate sequences could be generated).

Step 7: ***Finding final set of frequent sequences:*** The final list of frequent sequences will be the union of all the frequent-k sequences found so far. Fp=Fp1 U Fp2 U……Fpk. The outputs for inputs MDB_1 (Table 24) and MDB_2 (Table 25) are shown in Figure 28. The final output of each MDB_i (i=1,2,..n), will be all the frequent sequences and the sequence ids in which they occur. For example, in Figure 28, the final output for MDB_1 (Drug/side effects) consists of the frequent sequences (ie. frequent sequence of side effects) and the sequence ids (ie. Drugs) in which they occur. Similarly, the final output for MDB_2 (Patient/drugs) consists of the frequent sequences (ie. frequent sequence of drugs) and the sequence ids (ie. patients) in which they occur.

I/P:

Drug	Sequence of side effects
d1	<(1 2 3) (1)>
d2	<(1 2 3)>
d3	<(3)(4)>
d4	<(1)(3)>

O/P:

Frequent Sequences(side effects)	Sequence ids (drugs)
1	d1 d2 d4
2	d1 d2
3	d1 d2 d3 d4
(1 2)	d1 d2
(1 3)	d1 d2
(2 3)	d1 d2
(1 2 3)	d1 d2

Patient	Sequence of Drugs Purchased by patient
p1	<(d1 d2 d3)>
p2	<(d3)(d4)>
p3	<(d1 d2)(d1)(d3)>
p4	<(d1)(d3)>

Frequent Sequences(Drugs purchased)	Sequence ids(patients)
d1	p1 p3 p4
d2	p1 p3
d3	p1 p2 p3 p4
(d1 d2)	p1 p3
(d1)(d3)	p3 p4

Figure 28 (a) and (b): Final output for TidFSeq algorithm:

Fig. 28 (a): Frequent Sequences of Side Effects with Drugs having these side effect
Fig. 28 (b): Frequent Sequences of Drugs Taken with Patients Taking these drugs

The FS1 (Frequent sequence of side effects and corresponding drugs that they occur in) computed by TidFSeq algorithm for MDB1 is shown in Figure 28 (a):

FS1= F_1^1 U F_2^1 U F_3^1 = {(1 (d1d2d4)), (2 (d1d2)), (3 (d1d2d3d4)), ((12) (d1d2)), ((13) (d1d2)), ((23) (d1d2)), ((123)(d1d2))}

The FS2 (Frequent sequence of drugs and corresponding patients that purchase the drugs) computed by TidFSeq algorithm for MDB2 is shown in Figure 28 (b):

FS2= F_1^2 U F_2^2 = {(d1 (p1p3p4)), (d2 (p1p3)), (d3 (p1p2p3p4)), ((d1d2) (p1p3)), ((d1)(d3) (p3p4))}

Sample Query handled by the proposed algorithm for the given input tables (MDB1 (Drug/ side effects), MDB2 (patient/drugs)):

What are the possible frequent sequence of side effects that the patients p1 and p3 suffer from?

Answer: Patients p1 and p3 have purchased drugs d1 and d2 together. And the drugs d1 and d2 cause side effects 1, 2 and 3 to occur together. This answer can be derived by intersection set operation of the result sets of the input tables shown in Figure 28:

< (d1 d2), p1 p3 > ∩ <(1 2 3), d1 d2 > = < (1 2 3), p1 p3 >

Now we clearly see how the proposed algorithm is able to solve such complex queries for multiple related sequence tables that the existing systems were not able to achieve. The next section shows the proposed algorithm flowchart for the example inputs described in this section.

3.5 Flowchart for TidFSeq algorithm

The Figure 29 shows the flowchart of the proposed TidFSeq algorithm. The inputs to the algorithm in this flowchart are multiple related sequence tables and user defined minimum support counts, we have taken the Drugs/sequence of side effects table and Patient/sequence of drugs purchased table as the input multiple related sequence tables and support count as s=2. The following is a summary of the proposed TidFSeq algorithm, whose flowchart is given in Figure 29. The first step is to scan the tables once, and store

the < Sequence id, Position_id list > tuple for each item. Next, the items that do not satisfy the user defined minimum support count are removed (if the count of the number of sequence ids in which the item occurs in is less than the user defined minimum support count). The items that are retained are frequent 1 items. Next, candidate sequences are generated (using GSP candidate generation (Srikant and Agrawal (1996))) from the Frequent 1 items.

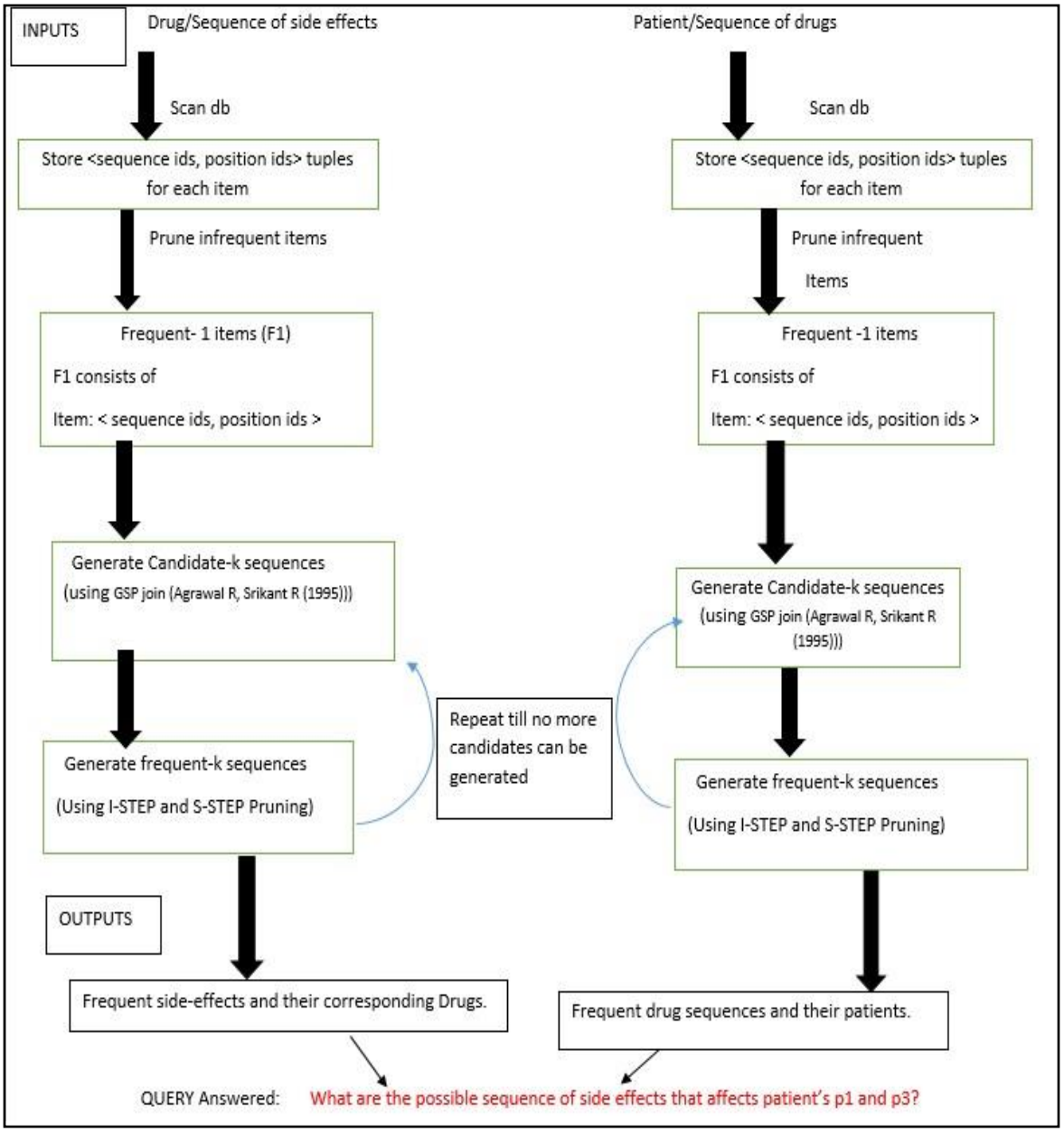

Figure 29: Proposed system flowdiagram

After the candidate generation step, the frequent sequences are found from the candidate sequences using the I-step and S-step pruning proposed by this algorithm. If the sequence is of the form (a b) ((ie.) items a and b are purchased together in same transaction in a retail store by a customer), then it is an I-step sequence and I-step pruning is done to check if it is frequent or not. (Ie.) If ‘a’ and ‘b’ have same sequence ids and occupy the same positions (same position ids), in greater than or equal to min-support number of sequences, then (ab) is frequent. For example, say there is an I-step sequence (1 2) and the < Sequence id, Position_id list > tuples for item 1: <(sid1,pos1) (sid2,pos1)> (ie. item 1 occurs in sid1 at position id: pos1 and also occurs in sid2 at position id: pos1) and tuples for item 2: <(sid1,pos1) (sid2, (pos1,pos4))> (ie. item 2 occurs in sid1 at position id: pos1 and also occurs in sid2 at position ids: pos1, pos4). From the tuples, we can see that item 1 and item 2 have a total of two matching sequence ids (ie. sid 1 and sid2) and also the corresponding position ids are matching (ie. items 1 and 2 occur at pos 1 in sid1 and again occur at pos1 in sid2). Since the condition for I-step sequence is met twice, the support count of sequence (1 2) is 2.

If the sequence is of the form (a) (b) (ie. items a and b are purchased during different transactions in a retail store by a customer), then it is an S-step sequence and S-step pruning is done to check if it is frequent or not. (Ie.) If ‘a’ and ‘b’ have same sequence ids and‘a’ occupies earlier position than that of ‘b’ (ie. position id of a is lesser than position id of b), in greater than or equal to min-support number of sequences, then (a)(b) is frequent. For example, say there is an S-step sequence (1) (2) and the < Sequence id, Position_id list > tuples for item 1: <(sid1,pos1) (sid2,pos2)> (ie. item 1 occurs in sid1 at position id: pos1 and also occurs in sid2 at position id: pos2) and tuples for item 2: <(sid1,pos3) (sid2, (pos1,pos4))> (ie. item 2 occurs in sid1 at position id: pos3 and also occurs in sid2 at position ids: pos1, pos4). From the tuples, we can see that item 1 and item 2 have a total of two matching sequence ids (ie. sid 1 and sid2) and also the corresponding position ids of item 1 are lesser than the corresponding position ids of item 2 (ie. the position id (pos1) of item 1 in sid1 is lesser than position id (pos3) of item 2 in sid1 and the position id (pos2) of item 1 in sid2 is lesser than position id (pos4) of item 2 in sid2). Since the condition for S-step sequence is met twice, the support count of sequence (1) (2) is 2.

Using the rules for I-step and S-step pruning as discussed above, every time candidate sequences are generated the infrequent sequences are pruned retaining the frequent sequences and their corresponding < Sequence id, Position_id list > tuples. This process keeps repeating until no more candidates can be generated. The final output from Drugs/sequence of side effects table will be, frequent sequence of side effects with their corresponding drugs and final output from Patient/sequence of drugs purchased table will be frequent sequence of drugs with their corresponding patients. This final result set is used to answer user queries for the input related sequence tables such as what are the frequent sequence of side effects that affects patients p1 and p3? The next chapter compares the experimental performance of TidFSeq algorithm with the ApproxMap algorithm and PrefixSpan algorithm, on three aspects: comparison based on query handled, comparison based on speed of processing, comparsion based on memory usage.

CHAPTER 4: EVALUATION OF TIDFSEQ ALGORITHM

This section compares the experimental performance of TidFSeq algorithm with the ApproxMap algorithm and PrefixSpan algorithm, which is a standard and prominent sequence mining algorithm. The three algorithms are implemented with Java language running under Eclipse environment. All experiments are performed on Intel(R) CORE i7-4700HQ CPU @ 2.40 Ghz. The operating system is Linux. Synthetic datasets are generated using the publicly available synthetic data generation program of the SPMF library at http://www.philippe-fournier-viger.com/spmf/index.php. Section 4.1 compares the results obtained by the TidFSeq algorithm, ApproxMap algorithm and PrefixSpan algorithm.

4.1 Comparison of algorithms over the query handled

Input 1(Drug/sequence of side effects):

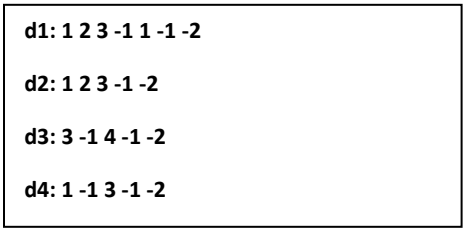

Figure 30: Input 1

Input 2(Patient/sequence of drugs purchased)

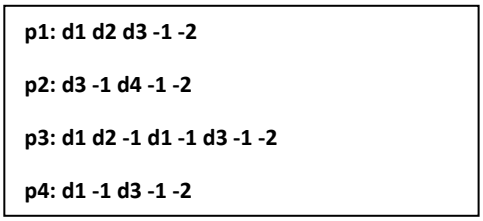

Figure 31: Input 2

Two small and simple synthetic sequence datasets (Figure 30 and Figure 31) that represent two multiple related sequence tables (ie. Drug/sequence of side effects and Patient/sequence of drugs purchased) are used to show how the proposed algorithm is able to mine the frequent sequences from both the datasets in order to solve a sample query that the ApproxMap and PrefixSpan algorithms are not able to solve. Note that the synthetic datasets are in the form of text file, where each row represents a sequence and each itemset in sequence is separated by '-1' and '-2' represents the end of sequence. For example, 1 2

3 -1 2 -1 -2 is a sequence, which contains two itemsets (1 2 3) and (2). And standard sequence datasets do not contain the sequence ids, it is assumed that each row represents a sequence and has an associated sequence id. For reference purpose, the sequence ids are included in both the datasets (Figure 30 and Figure 31) before the start of each sequence separated by a ':'.

The following table (Table 28) shows the final frequent sequences for both the related sequence inputs (Figure 30 and Figure 31) generated by the TidFSeq, ApproxMap and the PrefixSpan algorithms and their ability to answer the query (Query: What are the possible frequent sequence of side effects that the patients p1 and p3 suffer from?) in the table.

TidFSeq Result sets for Inputs 1 and 2	ApproxMap Result sets for Inputs 1 and 2	PrefixSpan Result sets for Inputs 1 and 2
(Drug/side effects) Output_table1: Frequent sequence of side-effects and corresponding drugs. (Patient/drugs) Output_table 2: Frequent sequence of drugs and corresponding patients.	(Drug/side effects) Approximate frequent sequence of side effects. (Patient/drugs) Approximate frequent sequence of drugs.	(Drug/side effects) The exact frequent sequences of side effects. (Patient/drugs) The exact frequent sequence of drugs.
Solution to Query: SQL JOIN query to get the answer: **SELECT Output_table1.side_effects,Output_table2.patients FROM Output_table1 INNER JOIN Output_table2 ON Output_table1.drugs=Output_table2.Drugs_purchased** Answer: (1 2 3) p1 p3 Ie. Patients p1 and p3 suffer from side effects (1 2 3) that occurs together.	Solution to query cannot be derived since the frequent sequences are not exact and SQL query to intersect the result sets cannot be used here.	Though the frequent sequences obtained from both input datasets are exact, SQL query to intersect both resultsets in order to get the answer will not work here.

Table 28: Comparison of query handling done by TidFSeq, ApproxMap and PrefixSpan algorithms

In the next subsections, the following experimentations are carried out for datasets conatining 250K, 500K, 750K, 1M and 2M sequences over different support counts (ie. 10%, 20%, 30%, 40%, 50%).

Next subsections provide the comparison of the three algorithms based on :

1. Execution speed (speed of processing) of the algorithms for datasets conatining 250K, 500K, 750K, 1M and 2M sequences over different support counts (ie. 10%, 20%, 30%, 40%, 50%).

2. Memory usage (in terms of MB) for Different Data Sizes at Minsupport of 40%.
3. Acuracy of the frequent sequences obtained for datasets conatining 250K, 500K, 750K sequences at Minsupport of 40%.
4. Execution speed for frequent sequences obtained for 250K sequence dataset having sequences of length greater than 10 elements.

4.2 Comparison of execution speed (speed of processing) of the algorithms

4.2.1 Execution times (in secs) for different datasets of different sizes at MinSupport of 40%

Algorithms / Size	250K	500K	750K	1M	2M
TidFSeq	34.32	59.2	78.45	113.56	145.62
ApproxMap	66.2	72.1	98.7	156.9	220.15
PrefixSpan	70	88	107.83	331	560.45

Table 29: Execution times (in secs) for different datasets of different sizes at MinSupport of 40%

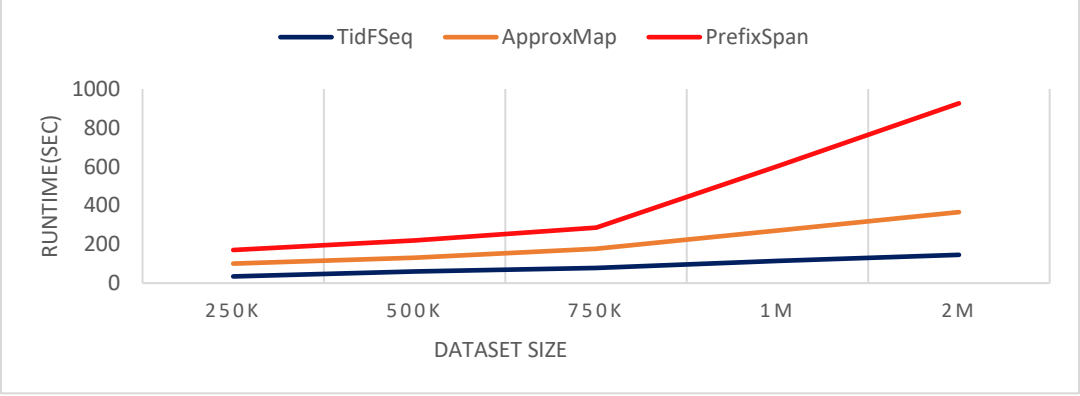

Figure 32: Execution times (in secs) for different datasets of different sizes at MinSupport of 40%

From Table 29 and Figure 32, we can see that the TidFseq algorithm performs considerably better than ApproxMap and PrefixSpan algorithm in terms of runtime for increasing data size (ie.increasing number of sequences). Though the ApproxMap just finds approximate patterns, compared to the TidFseq algorithm that finds exact sequences, the ApproxMap has to scan the input database twice (ie. first time to make all sequences of equal length and second time to scan the preprocessed database for frequent approximate sequences) compared to the TidFseq algorithm that scans database once to get the sequence id and position id list tuple for the items.

4.2.2 Execution times (in secs) for small-sized dataset (250K) for different support counts

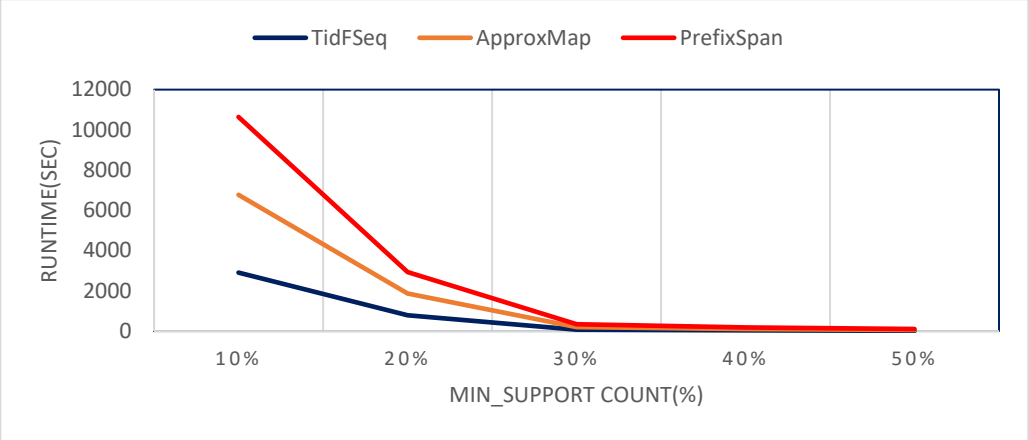

Figure 33: Execution times (in secs) for small-sized dataset (250K) for different support counts

Algorithms / Support	10%	20%	30%	40%	50%
TidFseq	2905.1	792	73.03	34.32	14.2
ApproxMap	3866.6	1072.1	128.7	66.2	41.5
PrefixSpan	3869.09	1073	141	78.4	46.66

Table 30: Execution times (in secs) for small-sized dataset (250K) for different support counts

From Table 30 and Figure 33, we can see that with increasing support count, the runtime for all the algorithms reduce. But the TidFseq algorithm clearly has better runtimes than the ApproxMap and PrefixSpan algorithm, especially when support is lesser than or equal to 30%.

4.2.3 Execution times (in secs) for Medium-sized dataset (500K) for different support counts

Algorithms / Support	10%	20%	30%	40%	50%
TidFseq	4011	876	103.1	59.2	25.7
ApproxMap	4934.76	1372.1	298.7	72.1	61.5
PrefixSpan	5009.14	2777.67	545.89	320	115

Table 31: Execution times (in secs) for Medium-sized dataset (500K) for different support counts

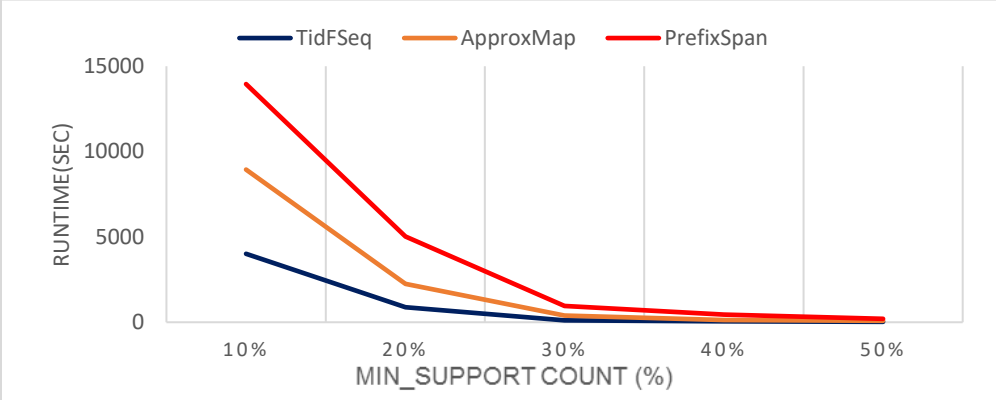

Figure 34: Execution times (in secs) for Medium-sized dataset (500K) for different support counts

From Table 31 and Figure 34, we can see a similar trend in runtimes just as how it occurred for all the three algorithms when data size was 250K.

4.2.4 Execution times for large-sized dataset (2M) for different support count.

When processing very large data and when support counts are less, the running times for all three algorithms are also bigger. It can be seen from Table 32 and Figure 35, that even when data is large, at higher support counts the TidFseq algorithm runs faster than the ApproxMap and PrefixSpan algorithms.

Algorithms / Support	10%	20%	30%	40%	50%
TidFseq	30216.66	4876.23	701.01	145.62	70.89
ApproxMap	39994.96	5172.41	1008.5	220.15	142.3
PrefixSpan	41101	9908.4	5640.11	1590	896.32

Table 32: Execution times for large-sized dataset (2M) for different support counts

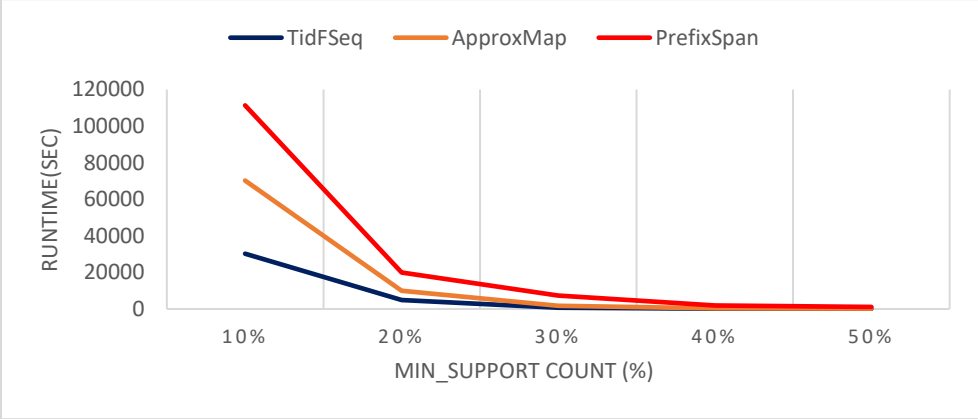

Figure 35: Execution times (in secs) for large-sized dataset (2M) for different support counts

4.2.5 Memory Usages (in terms of MB) for Different Data Sizes at Minsupport of 40%

Algorithms / Size	250K	500K	750K	1M	2M
TidFseq	14 MB	23 MB	32 MB	48 MB	60 MB
ApproxMap	5 MB	9 MB	16 MB	21 MB	28 MB
PrefixSpan	4 MB	7 MB	11 MB	20 MB	25 MB

Table 33: Memory Usages for Different Data Sizes at Minsupport of 40%

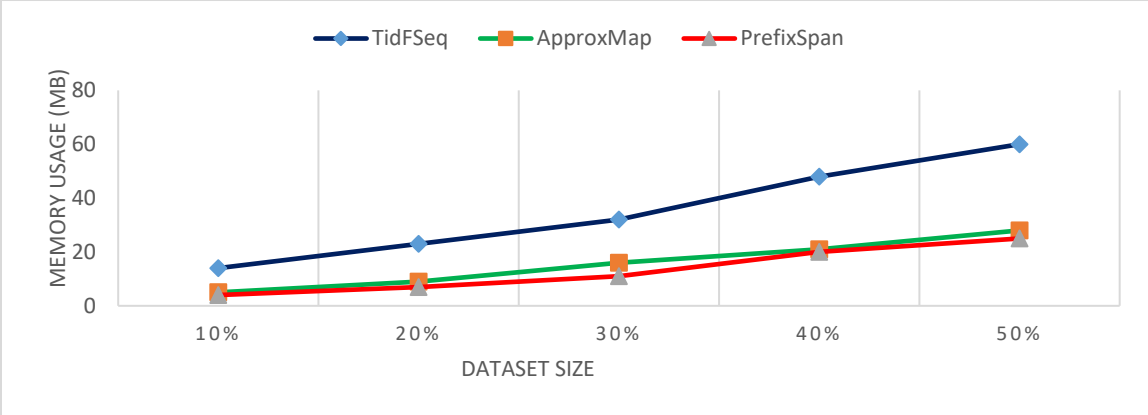

Figure 36: Memory Usages for Different Data Sizes at Minsupport of 40%

From Table 33 and Figure 36, it can be seen that though TidFseq algorithm has better runtime, the ApproxMap and PrefixSpan algorithm has better memory usages than the TidFseq algorithm. The memory usage increases with increase in size for both algorithms, but the memory usage for TidFseq algorithm increases at a much higher rate than the ApproxMap algorithm. This is because the TidFseq algorithm stores the sequence ids and corresponding position id list in form of tuples for each item/sub-sequence, where as the ApproxMap only stores the preprocessed database from which it directly outputs the frequent approximate patterns, while the PrefixSpan also takes some storage space for storing the projected databases in its method. Hence though memory usage requirement is higher for the TidFseq algorithm than the ApproxMap and PrefixSpan algorithms it is a reasonable trade-off since the output frequent sequences and their corresponding sequence ids help answer complex queries for multiple related tables that both the algorithms cannot answer.

4.3 Comparison of algorithms based on Acuracy of the frequent sequences obtained for datasets of different sizes

4.3.1 Acuracy of the frequent sequences obtained for datasets conatining 250K, 500K, 750K sequences at Minsupport of 40%.

Accuracy is calculated by finding:

Accuracy= Total number of exact frequent sequences for a sequence dataset / Number of exact sequences found

The following table, Table 34 shows the accuracy in terms of percentage for the TidFSeq, ApproxMap and PrefixSpan algorithms on datasets conatining 250K, 500K, 750K sequences at Minsupport of 40%. We can see that the TidFSeq and PrefixSpan algorithms mine the exact frequent sequences for datasets of all sizes. But the accuracy of the ApproxMap algorithm reduces as the dataset size increases.

Algorithms / Size	250K	500K	750K
TidFseq	100 %	100%	100%
ApproxMap	79%	71%	63%
PrefixSpan	100%	100%	100%

Table 34: Acuracy of the frequent sequences obtained for datasets conatining 250K, 500K, 750K sequences at Minsupport of 40%

4.4 Comparison of algorithms based on execution speed for (250K) dataset having longer sequences at Minsupport of 20%.

It can be seen from Table 35, that at a minimum support count of 20%, for a 250K dataset, the TidFSeq algorithm performs little slow when compared to it's speed on the datasets having shorter length sequences. It can also be seen that, the PrefixSpan algorithm sometimes runs slightly faster for longer sequences, this is because it uses small sized temporary prefix databases, that eliminate the need to use tuples,

which is being used in the TidFSeq algorithm. But the ApproxMap algorithm is still slow compared to the TidFSeq algorithm, since it does column wise support counting and hence more columns need to be processed for longer sequences.

Algorithms / length	10	15	20
TidFseq	480	890	1431
ApproxMap	620	1011	1666
PrefixSpan	488.96	856	1300

Table 35: Comparison of algorithms based on execution speed for (250K) dataset having longer sequences at Minsupport of 20%.

CHAPTER 5: CONCLUSION AND FUTURE WORK

This paper proposes the Transaction id frequent sequence pattern (TidFSeq) algorithm which extends the techniques of the TidFP algorithm for mining item sets to the problem of mining frequent sequences from MDBs to answer queries for multiple related sequence tables. A new technique of using < sequence ids, position id_list > tuples was derived from the <transaction id, itemsets> tuples used in the TidFp algorithm, in order to mine frequent sequences from multiple tables. Using the frequent sequences and their corresponding sequence ids mined from multiple related sequence tables, valuable user queries for multiple related sequence tables was answered. The experimental performance of TidFSeq algorithm with the ApproxMap algorithm and PrefixSpan algorithm, on three aspects: comparison based on query handled, comparison based on speed of processing, comparsion based on memory usage. It was observed that the TidFSeq algorithm had better processing speed for multiple sequence datasets than the ApproxMap and PresfixSpan algorithm. But the TidFSeq algorithm consumed more memory space for multiple sequence datasets than the ApproxMap and PresfixSpan algorithm. It was also observed that the processing speed of the proposed TidFSeq algorithm slowed down for sequence datasets containing long sequences (sequences of length more than 10).

5.1 Future Work

The following are some future work that can be done to extend our TidFSeq algorithm:

1) Other ways of storing sequences such as hashing methods can be used rather than use of memory consuming data tuples.
2) A parallel processing framework such as Map-Reduce framework can be utilized to process multiple sequence tables with more than two attributes, in parallel and data can stored in a distributed storage system such as Hadoop Distributed file System,in order to handle really big data.
3) Also the proposed algorithm can be extended to just mine maximal frequent sequence patterns and closed sequential patterns to avoid the setback in processing time caused by mining exact frequent sequence patterns that are long (ie. sequences of length more than 10).

REFERENCES

Adhikari.A, P.R. Rao, B. Prasad, J. Adhikari, Mining multiple large data sources, International Arab Journal of Information Technology, 7 (2) (2010), pp. 243–25.

Anderberg.M.R Cluster Analysis for Applications. Academic Press, 1973.

Agrawal.R and R. Srikant (1994) Fast algorithms for mining association rules. In Pro of the 20th Int'l Conf. on Very Large Databases (VLDB '94), Santiago, Chile, June 1994.

Altman, N. S. (1992). "An introduction to kernel and nearest-neighbor nonparametric regression". *The American Statistician*. 46 (3): 175–185. doi:10.1080/00031305.1992.10475879.

Animesh.A, P. R. Rao (2007) Enhancing quality of knowledge synthesized from multi-database mining, Pattern Recognition Letters, Volume 28 Issue 16.

Animesh.A (2013) Clustering local frequency items in multiple databases, July 2013 Information Sciences: an International Journal, Volume 237, Publisher: Elsevier Science Inc.

Agarwal R, Aggarwal CC, Prasad VVV (2001) A tree projection algorithm for generation of frequent itemsets. J Parallel Distribut Comput 61:350–371

Agrawal R, Shafer JC (1996) Parallel mining of association rules: design, implementation, and experience. IEEE Trans Knowl Data Eng 8:962–969

Ayres.J, J. Gehrke, T. Yiu and J. Flannick (2002)"Sequential Pattern Mining using a Bitmap Representation"*SIGKDD'02.*

Deng.J.L, Introduction of grey system theory, Journal of Grey System, 1 (1) (1989), pp. 1–24

El-Sayed M, Carolina R, Elke AR (2004) FS-miner: efficient and incremental mining of frequent sequence patterns in web logs. In: Proceedings of the 6th ACM international workshop on web information and data management, Washington DC, pp 128–135

Ezeife CI, Lu Y (2005) Mining web log sequential patterns with position coded pre-order linked WAP-tree. Int J Data Mining Knowl Discov, Kluwer Acad Publ 10:5–38.

Ezeife CI, Lu Yi, Liu Yi (2005) PLWAP sequential mining: open source code proceedings of the open source data mining workshop on frequent pattern mining implementations, in conjunction with ACM SIGKDD, Chicago, August 21–24, pp 26–29.

Ezeife.C.I, Zhang, D (2009): "TidFP: Mining frequent patterns in different databases with transaction ID". In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol.5691, pp. 125–137. Springer, Heidelberg (2009).

Galindo, C. (1998). On nonparametric estimation of mean functionals. Statistics and Probability Letters, 39, 143–149.

Hye-Chung.K,Joong Hyuk,Wei Wang(2006).Sequential Pattern Mining in Multi-Databases via Multiple Alignment.DMKD,12,151-180.

Han.J, Micheline Kamber, Jian Pei . (2012).*Data Mining: Concepts and Techniques 3rd Edition*. Waltham, MA: Elesevier Inc.

Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: Proceeding of the 2000 ACM-SIGMOD international conference on management of data (SIGMOD'00), Dallas, TX, pp 1–12.

Liu.H, H. Lu and J. Yao (2001) ,"Toward multidatabase mining: Identifying relevant databases" ,*IEEE Trans. Knowl. Data Eng.*,vol. 13 ,no. 4 ,pp.541 -553, 2001.

MacQueen, J. B. (1967). *Some Methods for classification and Analysis of Multivariate Observations*. Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. UC Press. pp. 281–297. MR 0214227. Zbl 0214.46201. Retrieved 2009-04-07.

Park JS, Chen MS, Yu PS (1995). An effective hash-based algorithm for mining association rules. In: Proceeding of the 1995 ACM-SIGMOD international conference on management of data (SIGMOD'95), San Jose, CA, pp 175–186.

Pei J, Han J, Mortazavi-asl B, Zhu H (2000) Mining access patterns efficiently from web logs. In: proceedings 2000 Pacific-Asia conference on knowledge discovery and data mining (PAKDD'00), Kyoto, pp 396–407

Pei J, Han J, Lakshmanan LVS (2001) Mining frequent itemsets with convertible constraints. In: Proceeding of the 2001 international conference on data engineering (ICDE'01), Heidelberg, Germany, pp 433–332

Pei.J, J. Han, H. Pinto, Q. Chen, U. Dayal,(2004) "Prefix Span: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth", IEEE Transactions on Knowledge & Data Engineering, vol. 16, no.1, pp.1424-1440, January, 2004.

Russell, Stuart; *Norvig, Peter* (1995). *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall. *ISBN 978-0137903955*

Rokach, Lior; Maimon, O. (2008). *Data mining with decision trees: theory and applications*. World Scientific Pub Co Inc. ISBN 978-9812771711.

Savasere A, Omiecinski E, Navathe S (1995) An efficient algorithm for mining association rules in large databases. In: Proceeding of the 1995 international conference on very large data bases (VLDB'95), Zurich, Switzerland, pp 432–443.

Shichao Zhang, Xindong Wu, Chengqui Zhang (2003) Mining Multi-Database Mining,IEEE Computational Intelligence Bulletin,Vol.2,No 1,pp 5-13.

Srikant.R and R. Agrawal (1996) Mining sequential patterns: Generalizations and performance improvements, Proc. of the 5th Int. Conf. Extending Database Technology, pp.3-17, 1996.

T. Mehenni, A.Moussaoui (2012) Data mining from multiple heterogeneous relational databases using decision tree classification, Pattern Recognition Letters, Volume 33 Issue 13 .

VIVA AUCTORIS:

NAME: VIGNESH ARAVINDAN

PLACE OF BIRTH: Chennai, India

YEAR OF BIRTH: 1994

EDUCATION : PSBB, Chennai, India- High School 2009-2011

Anna University, Chennai, India- Undergrad Computer science 2011-2015

University of Windsor, Windsor, Ontario M.Sc.CS 2015-2016