UNIVERSITY OF CALIFORNIA,
IRVINE

Scalable Association Rule Learning Algorithm for Very Large Dataset

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Engineering

by

Haosong Li

2020

DEDICATION

This study is wholeheartedly dedicated to my parents, who helped me in all things great and small.

To my academic adviser Professor Phillip Sheu who guided me in this process, and Professor Henry Lee who inspired my pursuit of this area

And lastly, I dedicated this thesis to my girlfriend Tingan Jin, who supported me in the preparation of this thesis during the hardship of COVID-19.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express the most profound appreciation to my committee chair, Professor Phillip Sheu, who gave me this opportunity to work on this topic and consistently provided suggestions to this research.

I would like to thank my committee members, Professor Nader Bagherzadeh and Professor Quoc-Viet Dang. This thesis would not be complete without your help.

In addition, a thank you to Professor Henry Lee, who encouraged me to pursue this area.

ABSTRACT OF THE THESIS

Scalable Association Rule Learning Algorithm for Very Large Dataset

by

Haosong Li

Master of Science in Computer Engineering

University of California, Irvine, 2020

Professor Phillip Sheu, Chair

Many algorithms have been proposed to solve the association rule learning problem. However, most of them suffer from the problem of scalability either because of unacceptable time complexity or tremendous memory usage, especially when the dataset is enormous and the minimum support (*minsup*) is low. This paper introduces a new approach that follows the divide-and-conquer paradigm, which can exponentially reduce both the time complexity and memory usage, even on a single machine.

INTRODUCTION

The association rule learning problem has played a significant role in data mining for the past few decades. Association rules are widely used in many fields, including market basket analysis [28], bioinformatics [22], and more. However, the problem has an NP-hard nature, meaning it is challenging to find the results within a reasonable period of time.

The invention of the Apriori Algorithm [1] made this problem computationally feasible for most computers on regular-sized datasets. Since then, researchers have continued to develop more scalable algorithms. Among others, FP-Growth [13] and Eclat [27] are two algorithms developed after the Apriori algorithm that improve the scalability of the Apriori algorithm.

The increasing popularity of the Internet in recent decades made big data available to many research institutions and companies. Their sizes are so large that traditional algorithms may not be able to handle efficiently. This imposes a challenge to the association rule learning problem as well. Most of the previously designed algorithms, including the Apriori algorithm, the FP-Growth algorithm, and the Eclat algorithm, suffer from the problem of scalability for big data. All these algorithms take an unacceptable time to terminate (will be discussed in the experiments section). In addition, the FP-Tree of the FP-Growth algorithm and the TID list of the Eclat algorithm may not fit in the memory.

This paper introduces an approach that makes it possible to mine associations rules and frequent itemsets for very large datasets. The approach, called the scalable association rule learning (SARL) algorithm, follows the divide-and-conquer paradigm and vertically divides the dataset into almost equivalent partitions using a graph representation and the k-way graph partitioning algorithm [2]. The total complexity of the SARL heuristic algorithm, including the overhead of partitioning the dataset, is lower than that of the Apriori algorithm. The memory usage is also lower than those of the current algorithms.

The rest of the paper is organized as follows. In Section 2, we provide a general view of other association rule learning algorithms and graph partitioning algorithms. In Section 3, we present the SARL heuristic and SARL precise algorithms with examples, formal descriptions, theorems, and proofs. The experiments and results are presented in Section 4, followed by the conclusion and future work.

The contributions of this paper include the provision of association graphs that represent an efficient estimation of potential frequent itemsets, the use of MLkP algorithm to divide the items into partitions while minimizing the loss the information, the generation of a bridge partition to achieve precise computation, and recursive reduction of the size of the bridge partition.

RELATED WORK

Association rule learning/frequent itemset mining has been an active research area. Among others, three approaches are considered the most popular, and possibly the most efficient: the Apriori algorithm, the FP-Growth algorithm, and the Eclat algorithm.

1. The Apriori Algorithm

The Apriori algorithm [1], introduced by Agrawal and Srikant, was the first efficient association rule learning algorithm. It incorporates various techniques to speed up the process as well as to reduce the use of memory. For example, the *Lk-1 X Lk-1* method used in the candidate generation process can reduce the number of candidates generated, and the pruning process can significantly reduce the number of possible candidates at each level.

One of the most important mechanisms in the Apriori algorithm is the use of the hash tree data structure. It uses this data structure in the candidate support counting phase to reduce the time complexity from O(*k*m*n*) to O(*k*m*T+n*), where *k* is the average size of the candidate itemset, *m* represents the number of candidates, *n* represents the number of items in the whole dataset, and *T* is the number of transactions.

The major advantage of the Apriori algorithm comes from its memory usage because only the *k-1* frequent itemsets, *Lk-1*, and the candidates in level *k*, *Ck*, need to be stored in the memory. It generates the minimum number of candidates based on the $L_{k-1} \times L_{k-1}$ (described in [1]) and the pruning method, and it stores them in the compact hash tree structure. In case the candidates fill up the memory from the large dataset and a low *minsup* setting, the Apriori algorithm will not generate all the candidates to overload the memory. Instead, it generates as many candidates as the memory can hold at a time.

2. The FP-Growth Algorithm

The Frequent Pattern Growth algorithm was proposed by Han et al. in 2000 [13]. It uses a tree-like structure (called Frequent Pattern Tree) instead of the candidate generation method used in the Apriori algorithm to find frequent itemsets. The candidate generation method finds the candidates of frequent itemsets before reducing them to the actual frequent itemsets through support counting.

The algorithm first scans the dataset and finds the frequent one itemsets. Then, the frequent pattern tree is constructed by scanning the dataset again. The items are added to the tree in the order of their support. Once the tree is completed, the tree is traversed from the bottom, and the conditional FP-Tree is generated. Finally, the algorithm generates frequent itemsets from the conditional FP-Tree.

The FP-Growth algorithm is more scalable than the Apriori algorithm in most cases since it makes fewer passes to the dataset and does not require candidate generation. However, it suffers from memory limitations since the FP-Tree is fairly complex and may not fit in the memory. Traversing the complexed FP-Tree may also be time-expensive if the tree is not compact enough.

3. The Eclat Algorithm

Different from the Apriori algorithm and the FP-Growth algorithm that work on horizontal datasets (e.g. T001: {1, 3} T002:{1, 4}), the Eclat (Equivalence Class Clustering and bottom-up Lattice Traversal) algorithm [27] uses a vertical dataset (e.g. Item1: {T001, T002}, Item3: {T001}, Item4:{T002}). The Eclat algorithm only scans the dataset once. It finds the frequent itemsets by intersecting the transaction sets.

The Eclat algorithm takes advantage of scanning the dataset only once. However, when the dataset is large, and the *minsup* is set to a low value, the TID associated with each itemset becomes very long. In fact, the results can be larger than the original dataset; therefore, they may not fit into the memory.

4. Other Association Rule Learning Algorithms

There are three categories of association rule mining/frequent itemset mining algorithms[8]: Apriori based algorithms, tree-based algorithms, and pattern growth algorithms. The Apriori algorithm, the Eclat algorithm, and the FP-Growth algorithm are the most popular algorithms for the three categories, respectively.

In the Apriori based algorithm category, proposed by Agrawal and Srikant in [1] the AprioriTID algorithm is similar to Apriori, except that it generates Ck-bar and it mines frequent itemsets from there instead of the dataset. The Apriori Hybrid algorithm is a combination of the Apriori algorithm and the AprioriTID algorithm. The DHP (direct hashing and pruning) algorithm [23] uses a hash function to distribute itemsets into buckets. If a bucket has the support lower than the *minsup*, then the bucket is discarded. The MR-Apriori [19] and HP-Apriori [21] algorithms are distributed versions of the Apriori algorithm. The MR-Apriori uses the MapReduce model on the Hadoop platform. They enable parallel execution of the Apriori algorithm.

The tree-based algorithms, represented by the Eclat algorithm, find the frequent itemset by constructing a lexicographic tree. The AIS algorithm [2] and the SETM algorithm [15] are the two earliest association rule mining algorithms. Reference [1] shows that the Apriori algorithm beats them in running time. The TreeProjection algorithm [3] counts the supports of the frequent itemsets and use the nodes on a lexicographic tree as the representation of these support numbers. The TM algorithm [26] maps the TID of each transaction to transaction intervals before performing intersections between these intervals.

Lastly, the algorithms in the pattern growth category focus on frequent patterns. The P-Mine algorithm [5] is a parallel computing algorithm that utilizes the VLDBMine data structure to store the dataset and speed up the distribution of data, while the LP-Growth algorithm[24] makes use of an array-based linear prefix tree to improve the memory efficiency. The Can-Mining algorithm [14] finds the frequent itemsets from a canonical-order tree, which speeds up the tree traversal process when the number of frequent itemsets is low. Finally, the EXTRACT algorithm[10] uses the theory of Galois lattice to derive association rules.

The algorithms discussed above have scalability problems. The Apriori based algorithms, represented by the Apriori algorithm, have to go through the expensive candidate generation and support counting process. This causes a disadvantage in running time. The tree-based and the pattern-growth type algorithms often suffer from excessive usage of memory. For example, the FP-Growth algorithm could build a complex FP-Tree which does not fit into the memory.

5. Graph Partitioning Algorithms

One of the key steps in the SARL algorithm is to partition the IAG (see below) into k balanced partitions. An efficient graph partitioning algorithm is crucial since the balanced graph partitioning problem is NP-complete [6]. We have implemented three algorithms and compared them for the partitioning cost and running time. They are the recursive version of the Kernighan-Lin Algorithm [18], the Multilevel k-way Partitioning Algorithm (MLkP) [17], and the recursive version of the Spectral Partitioning Algorithm [20]. Other graph partitioning algorithms include the Tabu search based MAGP algorithm [11] and the flow-based KaFFPa algorithm [25].

The Kernighan-Lin algorithm swaps the nodes assigned to both partitions and finds the largest decrease in the total cut size. The Multilevel k-way Partitioning algorithm (MLkP) uses coarsening-partitioning-uncoarsening/refining steps to shrink a graph into a much smaller graph. After partitioning, the graph is rebuilt to restore the original graph. A single global priority queue is used for all types of moves. The Spectral Partitioning Algorithm finds splitting the values such that the vertices in a graph can be partitioned with respect to the evaluation of the Fiedler vector.

Experiments have been conducted to compare the three algorithms. The datasets provided by Christopher Walshaw at the University of Greenwich [12] were used. We also ran experiments on complete graphs with 30 and 300 nodes. Each dataset was tested four rounds with the number of partitions (k) being 2, 4, 8, and 16.

dataset	# of nodes	# of edges	avg degree	k	METIS Time	Spectral Time	METIS Cost	Spectral Cost	KL Time	KL Cost
3elt.graph	4720	13722	2.90720339	2	0.1083529	54.73113894	97	94	Timeout	N/A
3elt.graph	4720	13722	2.90720339	4	0.104274511	45.01839089	220	236	Timeout	N/A
3elt.graph	4720	13722	2.90720339	8	0.082687616	34.23122334	392	341	Timeout	N/A
3elt.graph	4720	13722	2.90720339	16	0.084682226	27.92868638	618	602	Timeout	N/A
add20.graph	2395	7462	3.11565762	2	0.041537523	3.044170141	719	80	Timeout	N/A
add20.graph	2395	7462	3.11565762	4	0.069754601	5.512359381	1296	350	Timeout	N/A
add20.graph	2395	7462	3.11565762	8	0.048701763	12.52986908	1874	1199	Timeout	N/A
add20.graph	2395	7462	3.11565762	16	0.054409027	29.25708413	2370	1647	Timeout	N/A
add32.graph	4960	9462	1.90766129	2	0.06651473	63.91381288	10	8	Timeout	N/A
add32.graph	4960	9462	1.90766129	4	0.064602375	54.39832783	43	33	Timeout	N/A
add32.graph	4960	9462	1.90766129	8	0.068125963	45.34366322	85	89	Timeout	N/A
add32.graph	4960	9462	1.90766129	16	0.069462299	87.3277657	182	136	Timeout	N/A
data.graph	2851	15093	5.293931954	2	0.075086355	19.99383068	219	115	Timeout	N/A
data.graph	2851	15093	5.293931954	4	0.069795132	14.5627892	495	262	Timeout	N/A
data.graph	2851	15093	5.293931954	8	0.094658613	7.923767567	713	392	Timeout	N/A
data.graph	2851	15093	5.293931954	16	0.089031458	6.522737265	1349	992	Timeout	N/A
uk.graph	4824	6837	1.417288557	2	0.05449748	347.5232875	26	11	Timeout	N/A
uk.graph	4824	6837	1.417288557	4	0.073782206	150.0673718	57	50	Timeout	N/A
uk.graph	4824	6837	1.417288557	8	0.056687832	102.6085541	107	82	Timeout	N/A
uk.graph	4824	6837	1.417288557	16	0.060199022	53.14980578	181	145	Timeout	N/A
Complete Graph	30	870	29	2	0.0555	0.3539	225	114	0.0068	225
Complete Graph	30	870	29	4	0.003133	0.0351	337	316	0.01329	337
Complete Graph	30	870	29	8	0.00318	0.0488	393	380	0.02685	393
Complete Graph	300	8700	29	2	0.214009	0.19758	22484	8339	3.4756	22500
Complete Graph	300	8700	29	4	0.207079	0.2026431	33741	28022	4.891045	33750
Complete Graph	300	8700	29	8	0.18528	0.19459	39372	38846	4.888	39374

Figure 1 Experiment Results for Comparing MLkP, Kernighan-Lin, and Spectral Paritioning Algorithms

As shown in Figure 1, the MLkP algorithm has the highest speed in general. It is 560 times faster than the spectral partitioning algorithm and even faster than the recursive Kernighan-Lin algorithm. The spectral partitioning algorithm has, in general, the best partition quality. It is 1.3 times better than MLkP and much better than the recursive Kernighan-Lin algorithm. The recursive Kernighan-Lin algorithm takes too long to complete all five datasets. It also shows serious scalability issues for complete graphs.

Considering the MLkP algorithm has the best overall performance, we choose to use this algorithm for graph partitioning in our algorithm.

OUR SOLUTIONS

1. Definitions

- K-itemset: an itemset with k items
- Support: the occurrence of an item in the dataset.
- *Minsup*: the minimum requirement of support. The user usually provides this. Itemsets with support < *minsup* are eliminated.
- Confidence: the indication of robustness of a rule in terms of percentage. Confidence(X→Y) = support($X \cup Y$)/support(X)
- Minconf: the minimum requirement of confidence. The user usually provides this. Rules with confidence < minconf are eliminated.
- Item-Association Graph: a graph structure that stores the frequent associations between pairs of items.
- Balanced K-way Graph Partitioning Problem: Divide the nodes of a graph into k parts such that each part has almost the same number of nodes while minimizing the number of edges/sum of edge weights being cut off.

2. A Scalable Heuristic Algorithm, SARL-Heuristic

The following is an outline of the scalable heuristic algorithm.

Step 1: Find frequent one and two itemsets using the Apriori algorithm (when *minsup* is high) or the direct generation method(when *minsup* is low).

Step 2: Construct the item association graph (IAG) from the result of step 1.

Step 3: Partition the IAG using the multilevel k-way partitioning algorithm(MLkP).

Step 4: Partition the database according to the result of step 3.

Step 5: Call the modified Apriori algorithm or FP-Growth algorithm to mine frequent itemset on each transaction partition.

Step 6: Find the union of the results found from each partition.

Step 7: Generate association rules by running the Apriori-ap-genrules on the frequent itemsets found from step 6.

3. Example

Suppose the following dataset is given and *minsup* is set to 0.1 (or 10%, or $7 * 0.1 \approx 1$ occurrence), and minconf is set to 0.7 (or 70%):

TID	Items
T000	1, 2
T001	1, 2, 3
T002	4, 5
T003	1, 4, 5
T004	2, 3
T005	1, 2, 3
T006	1, 4, 5

Figure 2 Example Dataset 1

First, we use the Apriori algorithm to find frequent two itemsets. As an intermediate step, the Apriori algorithm finds the frequent one-itemset first (shown in Figure 2):

Frequent Itemsets	Support
{1}	5
{2}	4
{3}	3
{4}	3
{5}	3

Figure 3 Frequent One Itemsets

The frequent two-itemsets are found afterwards (shown in Figure 3):

Frequent Itemsets	Support
{1, 2}	3
{1, 3}	2
{1, 4}	2
{1, 5}	2
{2, 3}	3
{4, 5}	2

Figure 4 Frequent Two-Itemsets

Next, we transform the above frequent two-itemsets into an item association graph (IAG), shown in Figure 4:

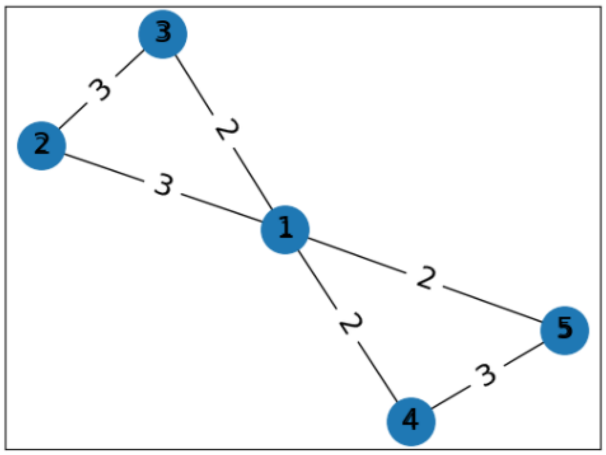


Figure 5 Item Association Graph Example

To construct the graph, we first take itemset {1, 2} with support 3. For this, we create node 1 and node 2 corresponding to the two items in the itemset. The edge between node 1 and node 2 has weight 3, representing the support of the itemset. The process is repeated for every frequent two-itemset found in the previous step. The IAG of this example is shown in Figure 5.

Next, we use the multilevel k-way partitioning algorithm (MLkP) to partition the IAG. In this case, the number of nodes is small, so we only bisect the graph by setting k = 2. The result is shown in Figures 6 and 7.

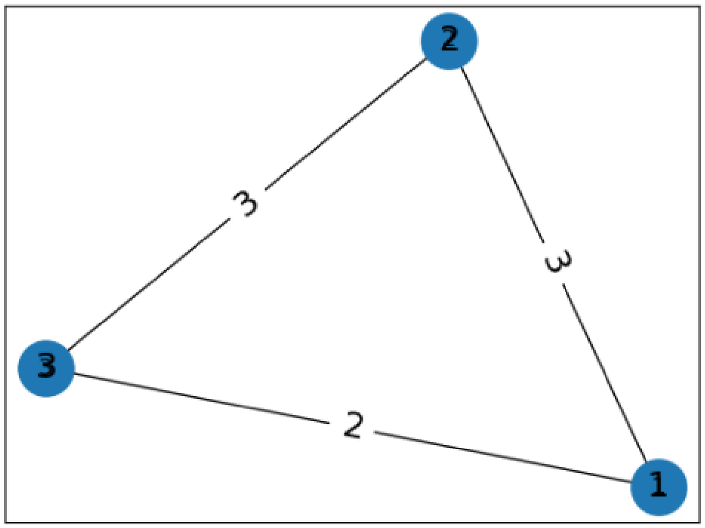


Figure 6 Item Association Graph Partition 1

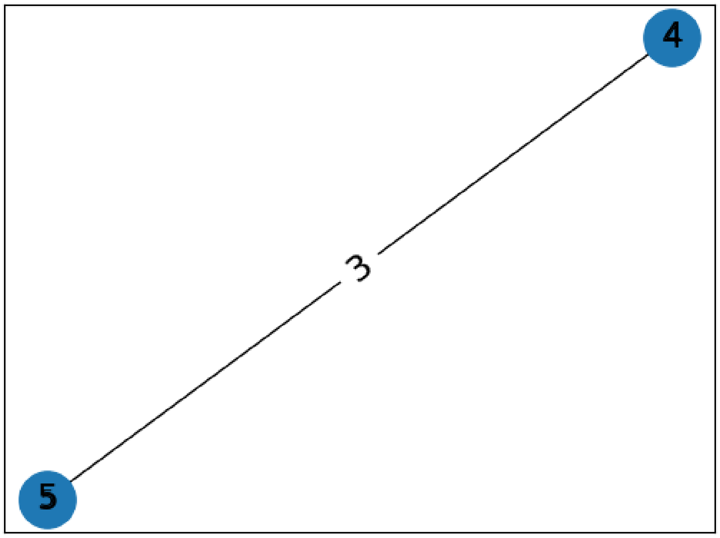


Figure 7 Item Association Graph Partition 2

The MLkP algorithm divides the IAG into two equal or almost equal sets in linear time while the sum of the weights of edges being cut off is the minimum.

Next, we partition the dataset according to the partitions of the IAG, as shown in Figures 8 and 9. Each transaction partition has all the items from the corresponding IAG partition. However, since the algorithm has already found all frequent one and two itemsets, a transaction will not be added to a transaction partition if the transaction has less than three items. For example, T000: {1, 2} is not added to the transaction partition 1, since it only has two items. Some items in the original dataset may not appear in any of the transaction partitions, because the infrequent one/two-itemsets are dropped in the IAG. This simplifies the subsequent computations. In this example, however, all the items are kept in the IAG because the IAG is a relatively dense graph. The following are the transaction partitions:

TID	Items
T001	1, 2, 3
T005	1, 2, 3

Figure 8 Transaction Partition 1

TID	Items
None	None

Figure 9 Transaction partition 2

The next step is to pick the best algorithm and use it to find the frequent k-itemsets with k > 2. For this example, we choose the modified Apriori algorithm because it is faster for mining small datasets as it avoids the process of finding the one and two-itemsets again. The results from partition 1 are shown in Figure 10:

Frequent Itemsets	Support
{1, 2, 3}	2

Figure 10 Frequent Itemsets from Transaction partition 1

Since the modified Apriori algorithm starts at finding three-itemsets, there are no additional frequent itemsets in the first partition. The following(Figure 11) are the results found in transaction partition 2:

Frequent Itemsets	Support
None	N/A

Figure 11 Frequent Itemsets from Transaction partition 2

The final results(shown in Figure 12) of frequent itemsets are simply the union of Figures 3, 4, 10, and 11:

Frequent Itemsets	Support
{1}	5
{2}	4
{3}	3
{4}	3
{5}	3
{1, 2}	3
{1, 3}	2
{1, 4}	2
{1, 5}	2
{2, 3}	3
{4, 5}	3
{1, 2, 3}	2

Figure 12 Frequent Itemset Final Results

After running the Apriori-ap-genrules algorithm, the association rules can be found in Figure 13.

Rules	Confidence
{2} → {1}	0.75
{3} → {2}	1
{5} → {1}	1
{2} → {3}	0.75
{5} → {4}	1

{4} → {5}	1
{1, 3} → {2}	1

Figure 13 Association Rule Generated

All frequent itemsets generated by the SARL algorithm are sound. Meaning each frequent itemset generated indeed is correct, and the support number is accurate. However, there may be some frequent itemsets that cannot be found by the SARL algorithm.

4. Formal Description of the SARL Algorithm

Following is the pseudo-code of SARL:

```
SARL:
results, two_itemset = mod1-Apriori(dataset)
graph = build_IAG(two_itemset)
partitions = METIS.partition(k, graph)
files = partition-dataset(partitions)
for file in files:
    results += mod2-Apriori(file)  #when files are small
    results += FP-Growth(file) #when files are large
rules = Apriori-gen(results)

mod1-Apriori(dataset):
C1 = {}
for transaction in dataset:
    for item in transaction:
        if item not in C1:
            add item to C1
      item.counter = 1
        else:
            item.counter += 1
L1 = {}
for candidate in C1:
    if candidate.counter >= minsup:
        add candidate to L1
C2 = {}
for itemset1 in L1:
    for itemset2 in L1:
        if itemset1 != itemset2:
            add itemset1 U itemset2 to C2
for transaction in dateset:
    for candidate in C2:
```

```
                    if candidate.issubset(transaction):
                              candidate.counter += 1
L2 = {}
for candidate in C2:
          if candidate.counter >= minsup:
                    add candidate to L2
return L1, L2
```

build_IAG(itemsets):

```
for itemset in itemsets:
          graph.add_node(itemset[0])
    graph.add_node(itemset[0])
    graph.add_edge(itemset[0], itemset[1], weight += 1)
return graph
```

partition-dataset(partitions):

```
for transaction in dataset:
          for partition in partitions:
                    intersect = parition intersect transaction
                    if len(inersect) > 2:
            add intersect to dataset_partition_i
return transaction partition names
```

5. Finding Frequent 2 Itemsets using the Apriori Algorithm

The first step of the SARL algorithm is to find the frequent 2 itemsets efficiently.

Although the Apriori algorithm has scalability issues for very large datasets, it provides a fast and convenient feature to extract intermediate results and a tolerable speed for the first two passes.

The Apriori algorithm finds frequent itemset Lk for each k, and each Lk is stored separately. We run the Apriori algorithm until it finds L2, the frequent two-itemset.

The following explains the detailed processing of finding L2, the frequent two itemsets, using the Apriori algorithm.

The algorithm first tries to find the frequent one itemsets by traversing the dataset and count the occurrence of each unique item, if the number of occurrences of an item is less than the *minsup* provided by the user, that item is eliminated from the list of frequent one-itemset. The frequent two itemsets are discovered based on the frequent one itemsets. The algorithm generates C2, the candidate sets for the frequent two itemsets, using Lk-1 × Lk-1:

$insert\ into\ C_k$

$select\ p.item_1, p.item_2, \dots, p.item_{k-1}, q.item_{k-1}$

$from\ L_{k-1}\ p, L_{k-1}\ q$

$where\ p.item_1 = q.item_1, \dots, p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1};$

This method generates a minimum number of candidates from the frequent one itemsets so that we can have fewer candidates to consider in the support counting phase. The Apriori algorithm also predicts and eliminates some infrequent itemsets before support counting by implementing the Apriori principle in the pruning step. If an item in C2 is not in L1, that item is infrequent, so all the two itemsets that include this item are dropped.

The Apriori algorithm is modified to terminate when pruning is done, and the two itemsets are found.

6. Construction of the Item Association Graph

The item association graph G is constructed based on the two itemsets generated by the Apriori algorithm. G is an undirected, weighted graph. A node Vi is created for each unique item i in the two itemsets T with the maximum item number being n.

$$\{V\} = \{\bigcup_{i=0}^{n} V_i \,|\, i \in |T|\}$$

The edges E in graph G are formed for each itemset in T:

$$\{E\} = \{ \bigcup_{i=0,j=0}^{n} E_{ij} \,|\{i,j\} \in T\}$$

The weight of each edge E_{ij} is equal to the support of itemset {i, j} in T:

$$W(E_{ij}) = Support(\{i,j\}) \mid \{i,j\} \in T$$

7. Partition the IAG using the Multilevel k-way Partitioning algorithm (MLkP)

The Multilevel k-way partitioning (MLkP) algorithm [17] is an efficient graph partitioning algorithm. The time complexity is O(E), where E is the number of edges in a graph, and the maximum load imbalance is limited to 3%.

The general idea of MLkP is to shrink (coarsen) the original graph into a smaller graph, then partition the smaller graph using an improved version of the KL/FM algorithm. Lastly, it restores(uncoarsen) the partitioned graph to a larger, partitioned graph.

METIS is a software developed by Karypis at the University of Minnesota. It includes an implementation of the MLkP algorithm that takes a graph as the input and outputs groups of nodes separated after the partition.

7. Transaction Partitioning

Based on the results of the MLkP algorithm that divide the items into groups P1, P2,…,Pm, we can partition the transactions into the same number of groups, where each group D_i contains only the items in partition P_i. For a transaction to be included in D_i, it must have all the items from partition P_i. If a transaction includes more items than the items from partition P_i, the intersection between the items in P_i and the transaction will be added to D_i. That is, part of the transaction will be added to D_i. As a result, each transaction in a transaction partition must be a subset of the corresponding transaction in the original dataset. If a transaction has less than three items, the transaction is not added. This is because we have already mined the one and two itemsets, and are only interested in three or more itemsets. This optimization helps to reduce the size of transaction partitions.

$$D_i = \{\bigcup_{j=1}^{n} T_j \,|(T_j \rightarrow \mathrm{S_j} \cap P_i|S_j \in D)\}$$

In the above, D_i is transaction partition i, T_j is the transactions to be added to partition i, S_j is the jth transaction in the original dataset, P_i is the item partition i, and D is the original dataset.

8. Algorithm Selection and Association Rule Learning on Transaction Partitions

One of the benefits that come with our solution is that the association rule learning on each transaction partition can be optimized by using the algorithm that best fits the partition.

During the association rule learning on the partitioned datasets, we have three algorithms that are considered efficient for this job: the Apriori algorithm, the FP-Growth algorithm, and the Eclat algorithm.

Since the modified Apriori algorithm has already computed the one itemsets and two itemsets during the preparation phase, the candidate generation feature of the Apriori algorithm using the horizontal dataset is handy in this case. We have modified the Apriori algorithm to skip the frequent one/two itemsets finding stages and start with finding the frequent three itemsets from the transaction partitions. This modification is particularly helpful when the *minsup* is set to high so that the expected number of itemsets are limited after the two itemsets.

We can estimate the expected number of itemsets from the average transaction length of each transaction partition. A higher average transaction length indicates a higher possibility of the presence of a long "tail" in the result. Results with long tails have itemsets with considerable maximum lengths, while results with short tails only contain itemsets with small maximum lengths. A dataset with an expected long tail means the association rule learning algorithm does not terminate soon after finding the two itemsets.

The average transaction length provides a fast and straightforward reference for selecting the best algorithm for each transaction partition. If the average transaction length is low, the Apriori algorithm can be the right choice, as the modified Apriori algorithm continues from the two itemsets that the preparation phase has already calculated. If the average transaction length is high, we can take advantage of the scalability of the FP-Growth algorithm. We omit the Eclat algorithm because the FP-Growth and the Eclat algorithms do not have the same advantage provided by the modified Apriori algorithm, of which the algorithm can start with the two itemsets. In addition, studies [16] show that the Eclat algorithm is slightly less scalable than the FP-Growth algorithm.

Next, the selected algorithm is used to find the frequent local itemsets from the given transaction partition. After the algorithm has terminated, a simple union is performed on the frequent itemsets found from each partition. Finally, *Apriori-ap-genrule* is used to derive the rules from the frequent itemsets. This step is relatively simple.

9. Time Complexity and Space Complexity

The theoretical time and space complexity of the Apriori algorithm is $O(2^d)$ where d is the number of unique items in the dataset.

The theoretical time and space complexities of the SARL algorithm consists of the complexity of several parts:

(1) Time complexity of frequent 2-itemsets generation using the Apriori algorithm

Finding frequent 2-itemsets requires finding 1-itemsets first. This step is simply $O(n)$ as the algorithm traverses the dataset once. Next, the candidate generation for 2-itemsets takes $O(d^2)$ where d is the number of unique items in the dataset. Finally, the support checking requires $O(n + d^2T)$ where T is the number of transactions in the dataset. Therefore, the time complexity of this step is $O(d^2T + n)$.

(2) Time complexity of IAG construction

Since each edge in the IAG is a representation of a frequent two-itemset, and the maximum number of two-itemsets is $\frac{d^2+d}{2}$, the maximum number of edges in IAG is also $\frac{d^2+d}{2}$. Therefore, constructing the IAG takes $O(d + \frac{d^2+d}{2})$ or $O(d^2)$.

(3) Time complexity of IAG partition

The time complexity of the IAG partition is equal to the time complexity of the MLkP algorithm, which is $O(E)$ or $O(d^2)$.

(4) Time complexity of transaction partition

The dataset is traversed once to assign items into different partitions. Hence the time complexity is $O(n)$.

(5) Time complexity of algorithm selection and running the selected algorithm

The algorithm selection requires the calculation of the average transaction width of each transaction partition. The time complexity of this is $O(kn)$, where k is the number of partitions.

If the modified Apriori algorithm is selected, the theoretical time complexity for each partition is $O(2^{1.03d/k})$ the coefficient 1.03 comes from the 3% maximum imbalance of the partitions caused by the MLkP algorithm. The total running time for all partitions is $O\left(k * 2^{\frac{1.03d}{k}}\right) = O(2^{\frac{1.03d}{k}})$, and the total time complexity of the SARL algorithm, when the modified Apriori algorithm is selected, is $O\left(d^2T + n + d^2 + d^2 + n + 2^{\frac{1.03d}{k}}\right) = O(d^2T + n + 2^{\frac{1.03d}{k}})$. Assume $n \gg d$, and $2^{\frac{1.03d}{k}} \gg n$, the time complexity can be simplified to $O(2^{\frac{1.03d}{k}})$. Compared with the time complexity of the Apriori algorithm, the SARL is $O\left(\frac{2^d}{2^{\frac{1.03d}{k}}}\right) = O(2^{\frac{k-1.03}{k}d})$ times faster than the Apriori algorithm. The exponential speed up comes from the small number of unique items in each transaction partition. The algorithm chosen to mine frequent itemsets from the transaction partitions only needs to consider a portion of all items for each partition.

(6) Space complexity of frequent 2-itemsets generation using the Apriori algorithm

Finding the frequent two itemsets requires finding the one itemsets first. This step is $O(d)$, where d is the number of unique items in the dataset, as we need to keep at most d items in the memory. Next, the candidate generation step for the 2-itemsets takes $O(d^2)$ space for at most $\frac{d(d-1)}{2}$ frequent 2-itemsets candidates. Finally, the support checking requires another $O(d^2)$ space to store the support numbers. Hence, this step requires $O(d^2)$ space.

(7) Space complexity of IAG construction

Since each edge in the IAG is a representation of a frequent two-itemset, and the maximum size of the two-itemsets is $\frac{d^2+d}{2}$, the maximum number of edges in IAG is also $\frac{d^2+d}{2}$. Therefore, storing the IAG takes $O(d^2)$ space.

(8) Space complexity of IAG partition

The space complexity of the IAG partition is equal to the space complexity of the MLkP algorithm, which is $O(E)$ or $O(d^2)$.

(9) Space complexity of transaction partition

The dataset is traversed once to assign items into different partitions. We can assume each partition can fit into the memory. Therefore, the space complexity is $O(\frac{n}{k})$.

(10) Space complexity of algorithm selection and running the selected algorithm

The algorithm selection requires the calculation of the average transaction width of each transaction partition. The space complexity of this is $O(k) = O(1)$, where k is the number of partitions.

If the modified Apriori algorithm is selected, the theoretical space complexity for each partition is $O\left(2^{\frac{1.03d}{k}}\right)$, where the coefficient 1.03 comes from the 3% maximum imbalance of partitions caused by the MLkP algorithm. The total space complexity for all partitions is therefore $O\left(k * 2^{\frac{1.03d}{k}}\right) = O(2^{\frac{1.03d}{k}})$, and the total space complexity of the SARL algorithm, when the modified Apriori algorithm is selected, is $O\left((3-1) * d^2 + \frac{n}{k} + 2^{\frac{1.03d}{k}}\right) = O(d^2 + \frac{n}{k} + 2^{\frac{1.03d}{k}})$. Assume $\frac{n}{k} \gg d$, and $2^{\frac{1.03d}{k}} \gg \frac{n}{k}$, the space complexity can be simplified to $O(2^{\frac{1.03d}{k}})$. Compared with the space complexity of the Apriori algorithm, SARL uses only $O\left(\frac{2^{\frac{1.03d}{k}}}{2^d}\right) = O\left(2^{\frac{1.03-k}{k}d}\right) = o(\frac{1}{2^{\frac{k-1.03}{k}d}})$ space comparing to the Apriori algorithm. The exponential reduction of space usage comes from the small number of unique items in each transaction partition. If the modified Apriori is chosen to mine frequent itemsets from the transaction partitions, it only generates a small number of candidates for each transaction partition, since it does not consider items in other partitions.

(11) Summary of the time and space complexities

Both the time and space complexity of the SARL algorithm is $O(2^{\frac{1.03d}{k}})$.

10. Error Bound

The SARL heuristic sacrifices some precision to obtain the speed up. However, every frequent itemset found by the algorithm is correct, and the support associated with each frequent itemset is also correct. The heuristic may miss some trivial frequent itemsets, i.e., the itemsets with low support. During the IAG partition phase, the MLkP algorithm makes cuts on the IAG to minimize the sum of the weights of the edges that are cut off. This feature helps to prevent large weights from cut off, while some trivial, small-weight (support) edges may be lost.

In the most (extreme) case, when every transaction has all items and *minsup* is set to 0, we can calculate the error bound. In this case, the IAG is a complete graph, and the fraction edge cut off by the MLkP algorithm is $\frac{n*\left(n-\frac{n}{k}\right)}{E} = \frac{(k-1)n}{k(n-1)}$. When n is very large, the fraction is approximately $\frac{k-1}{k}$. In this case, we can set k as low as 2 to still maintain 50% coverage for the frequent three or more itemsets. The calculation of frequent one and two itemsets is always accurate because they are calculated using the Apriori algorithm or the direct-generate algorithm.

The error rate should be significantly lower in more practical cases. However, it is difficult to estimate such error rate considering it is affected by many factors such as the closeness of groups of items (i.e., does an item appear with only a small number of other items?), the choice of *minsup*, and the max length of the frequent itemsets. We can make a rough estimation by introducing a parameter P_{out}, the ratio of edges cut off in the IAG. $P_{out} = \frac{E_{cut}}{E_{total}}$. This parameter is determined by the characteristics of a dataset, the *minsup* choice, and the number of partitions we choose. P_{out} is also a rough estimation of the error rate for the frequent two or more itemsets. Assume the ratio of the frequent two or more itemsets found is P_m , $P_m = \frac{\# \, frequent \; 2+ \; itemsets}{\# \, total \; frequent \; itemsets}$, then the total error bound can be computed as $Error_{total} = P_m * P_{out}$.

11. Benefits of Having Datasets Fit into the Memory

Since the transaction partitions are small enough to fit into the memory, any operations performed on the dataset should be much faster. For example, the Apriori algorithm makes the number of passes on the dataset equal to the maximum length of frequent itemsets. Each of these passes requires reading the dataset from the disk. With our solution, the SARL algorithm makes at most two passes to the dataset. The first pass is to generate the frequent one and two itemsets, and in the second pass, the algorithm brings a fraction of the dataset into the memory. All further passes are made directly in the memory, resulting in speedup.

12. Theorems and Proofs

Theorem 1: Soundness - All frequent itemsets and association rules generated by the SARL algorithm are correct.

Proof:

Assume the SARL algorithm generates an incorrect frequent itemset. We can assume the correctness of the Apriori algorithm and the FP-growth algorithm. Therefore, there must be an error in transaction partitioning. There could be two possible types of error in transaction partitioning:

(Possibility 1) The number of itemset appearances is higher than it should be.

(Possibility 2) Some transactions have additional items that are introduced by error.

Assume the first possibility is true. We divide the dataset vertically (item-wise) during the transaction partitioning phase. Since every item in the original dataset D that belongs to P_i must be added to D_i. All unique items in a transaction partition must appear in the same number of transactions as the original dataset. Hence, the number of itemset appearances, or the support of each itemset, should be the same as the original dataset. This conflicts with the first possibility: the number of itemset appearances is higher than it should be.

Assume the second possibility is true. During the transaction partitioning phase, each transaction in the original dataset may be assigned to a transaction partition, or it may be split into different disjoint parts. Therefore, each transaction in a transaction partition must be a subset of the corresponding transaction in the original dataset, and this process cannot add any new items into any transactions. Hence, we find a contradiction between our algorithm and the second possibility.

In summary, since both possibilities are proved to be false, the SARL algorithm is sound.■

Theorem 2: Computing the frequent two itemsets is considered relatively trivial compared to computing the frequent three or more itemsets.

Proof:

If the computation of the frequent two itemsets takes more than half of the total computation time, we may say computing frequent two itemsets is not trivial.

To characterize the distribution of frequent itemsets is relatively difficult due to the challenges in modeling the data. To get a general view on this topic, we developed a mathematical model to simulate the characteristics of any dataset. The relationships of all the frequent itemsets can be depicted using an itemset lattice diagram shown below:

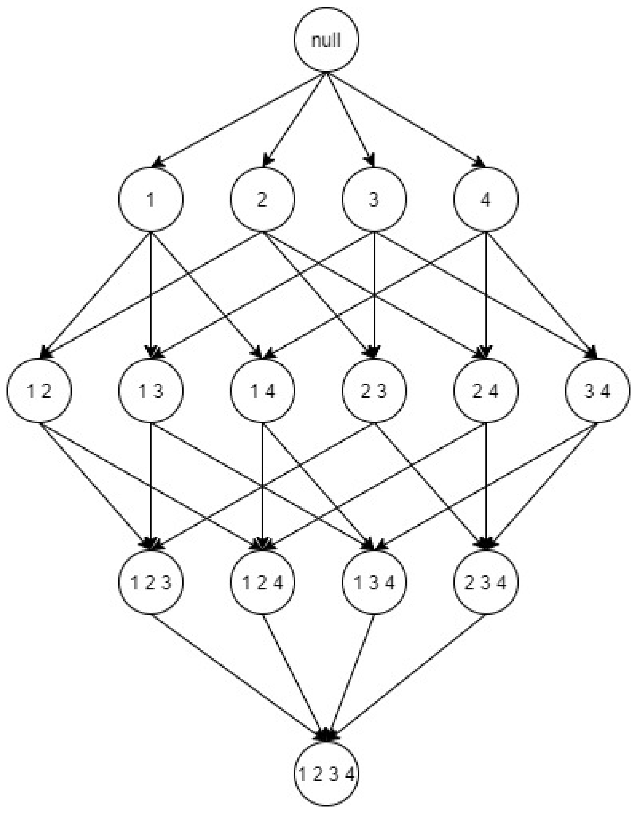


Figure 14 An Itemset Lattice

Figure 14 shows the case when every itemset has the support greater than *minsup*. However, in most cases, each layer will have some itemsets being removed due to either one of the two reasons: the anti-monotone property of the Apriori principle or the lack of support (i.e., support < *minsup*). To model the former, we apply the anti-monotone property to the itemset lattice. The anti-monotone property is as follows:

$$\forall X, Y \in J: (X \subset Y) \rightarrow f(Y) \leq f(X),$$

where if $J = 2^I$, I being a set of items, X is a subset of Y, then the measure f must be anti-monotone. Applying this property to the lattice, we can have the following explanation: if an itemset is infrequent, then all of its supersets must also be infrequent.

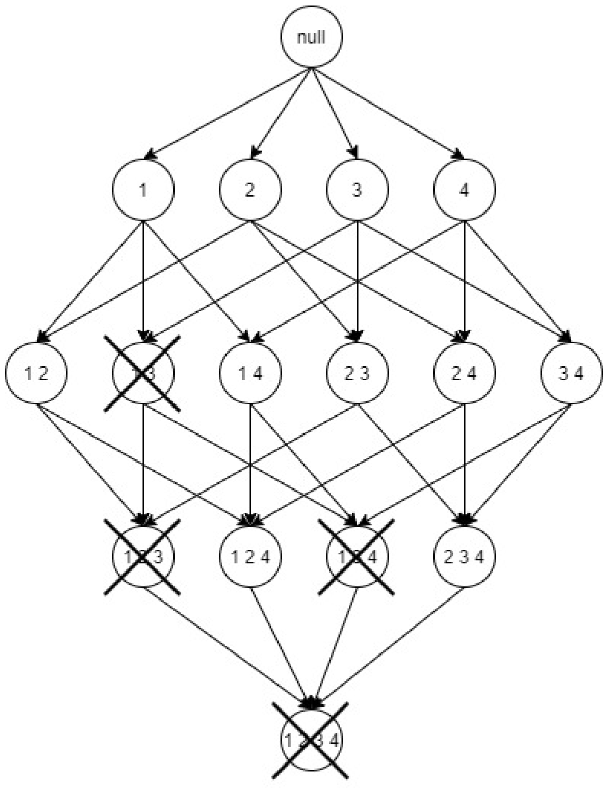


Figure 15 An Example of Pruning

For example, in Figure 15, if {*1, 3*} is infrequent, then *{1, 2, 3}, {1, 3, 4},* and *{1, 2, 3, 4}* are all infrequent.

To model this property, we can imagine that each infrequent itemset in the same layer causes some supersets in the next layer to be infrequent. The first infrequent itemset results in *n-k+1* infrequent itemsets in the next layer, where n is the number of unique items in the dataset, and k is the current layer number or the number of items in each itemset in the current layer. We know that each layer has C_k^n itemsets if none of them is infrequent. Then the next layer will have C_{k+1}^{n}total itemsets. Since *n-k+1* is the number of current infrequent itemsets in the next layer, $\frac{n-k+1}{C_{k+1}^{n}}$ is the current fraction of frequent itemsets over all the itemsets in the next layer. Therefore, $1-\frac{(n-k+1)}{C_{k+1}^{n}}$ is the probability of having a frequent itemset in the next layer if we randomly choose an itemset, and the second infrequent itemset should cause $\left(1-\frac{(n-k+1)}{C_{k+1}^{n}}\right)*(n-k+1)$ infrequent itemsets in the next layer. For the same reason, the third infrequent itemset in the current layer should cause

$\left(1-\frac{(n-k+1)+\left(1-\frac{(n-k+1)}{C_{k}^{n}}\right)*(n-k+1)}{C_{k+1}^{n}}\right)*(n-k+1)$ infrequent itemsets in the next layer. We can now estimate the number of infrequent itemsets I in the next layer using the number of infrequent itemsets in the current layer:

$$I_k = (n-k+1)+\left(1-\frac{(n-k+1)}{C_{k+1}^{n}}\right)*(n-k+1)+\left(1-\frac{(n-k+1)+\left(1-\frac{(n-k+1)}{C_{k}^{n}}\right)*(n-k+1)}{C_{k+1}^{n}}\right)*(n-k+1)+\cdots$$

The remaining frequent itemsets in layer k considering the above estimation of the influence of the Apriori principle is $C_{k}^{n}-I_{k-1}$. Let us assume the probability p that an itemset to be frequent, assuming its parent is frequent. We can have the final estimated number of frequent itemsets for layer k:

$$f_k = (C_{k}^{n}-I_{k-1})*p^{k}$$

For n = 200, p = 0.8, 0.6, 0.4, 0.2, 0.1, we can estimate the number of two, three, and more itemsets as shown in Figure 16:

p	# two itemsets	# three itemsets	# four itemsets	2/(3+4)
0.8	12736	228346	972761	0.010603552
0.6	7164	41585	714273	0.009477971
0.4	3184	6761	30960	0.084409215
0.2	796	589	1898	0.320064335
0.1	199	67	118	1.075675676

Figure 16 Estimation of the Number of Itemsets

For n = 2000, p = 0.8, 0.6, 0.4, 0.2, 0.1, we can estimate the number of two, three, and more itemsets as shown in Figure 17:

p	# two itemsets	# three itemsets	2/3
0.8	1279360	231482728	0.005527
0.6	719640	42159431	0.017069
0.4	319840	6855578	0.046654
0.2	79960	597871	0.133741
0.1	19990	68301	0.292675

Figure 17 Estimation of the Number of Itemsets for Larger Dataset

The above model with examples shows that the number of two itemsets is, on average, less than only 10% of the number of three or more itemsets. This means that only less than 10% of all computation power is consumed by the two itemsets. Thus, our algorithm speeds up the costly part, the part that mines three or more itemsets. ■

Theorem 3: Consider a value of *minsup* such that the fraction of frequent one itemset over the total number of unique items, *d*, denoted by *f*, is less than (*1 - the maximum imbalance rate)*, where *the maximum imbalance rate* is usually set to 3% based on the MLkP algorithm. If the partition by MLkP is *k*-way, then each partition contains less than *d/k* unique items, where *d* is the total unique items in the original

dataset. As a consequence, the complexity of each partition can be reduced.

Proof:

Assume that given f < 100% - 3% or f < 97%, and a transaction partition has $d_i \geq d/k$ unique items. According to our algorithm, since $d_i \geq d/k$, a partition in the IAG must have more than or equal to *d/k* nodes. As we assumed earlier, the maximum imbalance rate for the MLkP algorithm is set to 3%, then the number of nodes *n* in the IAG can be calculated as $\frac{d}{k} * 0.97 * k \leq\ n \leq \frac{d}{k} * 1.03 * k\ or\ 0.97d \leq n \leq 1.03d$. Since *n* cannot be more than the total number of unique items, $0.97d \leq n \leq d$. However, we know *f < 97%* or *f *d< 0.97d*, and $n \leq f * d$ since some frequent one itemsets may not appear in any frequent two itemsets, so $n \leq f * d < 0.97d$ and $n < 0.97d$. This contradicts $0.97d \leq n \leq d$. Therefore, the assumption $d_i \geq d/k$ is false, and the reverse, $d_i < \frac{d}{k}$, must be true. ■

13. Scalable Precise Algorithm, SARL-Precise

The following example aims to show the motivation for the development of the SARL-Precise algorithm.

Suppose the same dataset discussed earlier, as shown in Figure 18, is given and *minsup* is set to 0.1 (or 10%, or $7 * 0.1 \approx\ 1$ occurrences), and *minconf* is set to 0.7 (or 70%):

TID	Items
T000	1, 2
T001	1, 2, 3
T002	4, 5
T003	1, 4, 5
T004	2, 3
T005	1, 2, 3
T006	1, 4, 5

Figure 18 Example Dataset

First, we use the Apriori algorithm to find frequent two-itemsets. As an intermediate step, the Apriori algorithm finds frequent one itemset, as shown in Figure 19:

Frequent Itemsets	Support
{1}	5
{2}	4
{3}	3
{4}	3
{5}	3

Figure 19 Frequent One Itemsets

The frequent two-itemsets are found afterward, as shown in Figure 20:

Frequent Itemsets	Support
{1, 2}	3
{1, 3}	2
{1, 4}	2
{1, 5}	2
{2, 3}	3
{4, 5}	2

Figure 20 Frequent Two Itemsets

Next, we transform the above frequent two-itemsets into an item association graph, as shown in Figure 21.

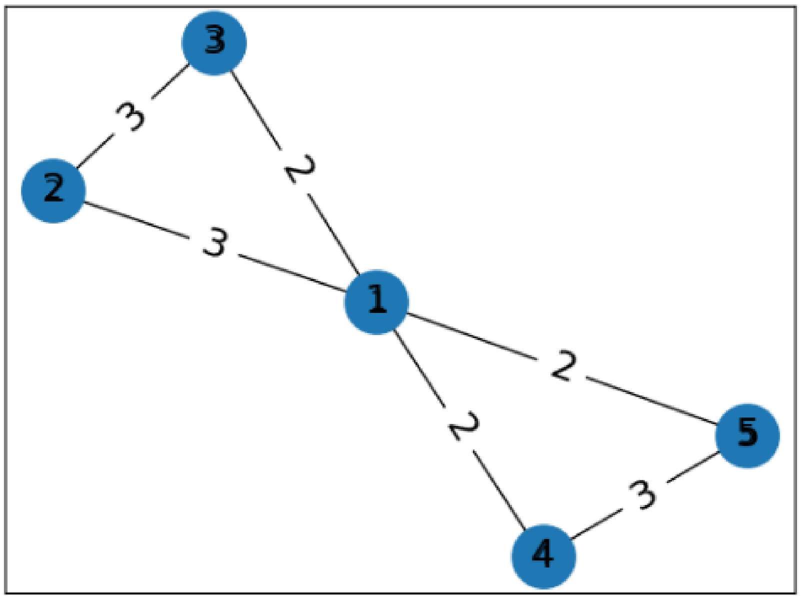


Figure 21 Item Association Graph Example

To construct the graph, we take itemset {1, 2} with support 3, create node 1 and node 2 that correspond to the two items in the itemset. The edge between node 1 and node 2 has weight 3, representing the support of the itemset. The process is repeated for every frequent two-itemset found in the previous step.

Next, we use the multilevel k-way partitioning algorithm (MLkP) to partition the IAG. In this case, the number of nodes is small, so we only bisect the graph by setting $k = 2$.

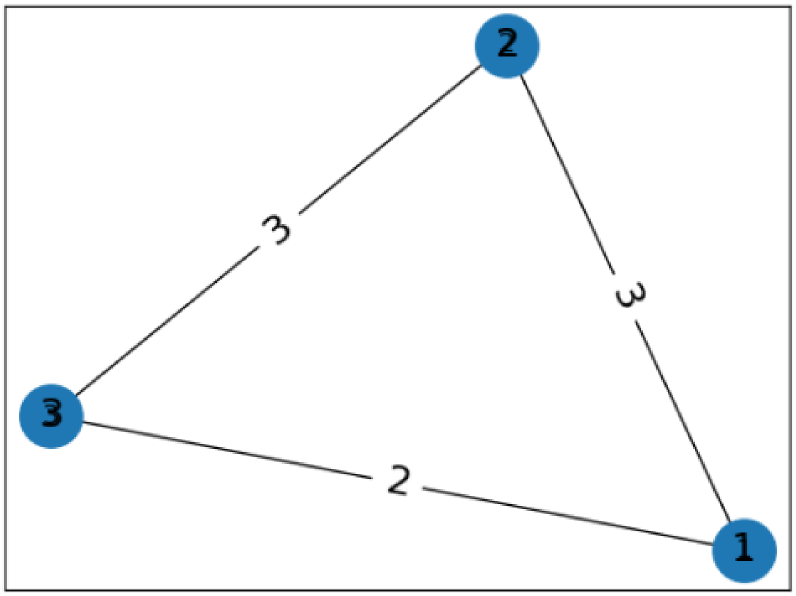


Figure 22 Item Association Graph Partition 1

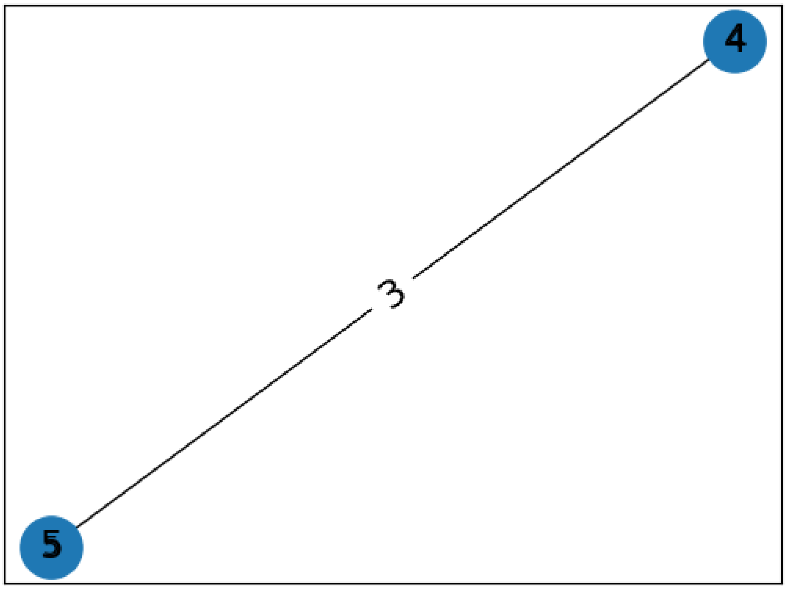


Figure 23 Item Association Graph Partition 2

The MLkP algorithm divides the IAG into two equal or almost equal sets in linear time while the sum of the weights of edges cut off is the minimum. The results are shown in Figure 22 and Figure 23.

Next, we partition the dataset according to the partitions of the IAG, as shown in Figures 19 and 20. Each transaction partition has all the items from the corresponding IAG partition. However, since the algorithm has already found all frequent one and two itemsets, a transaction will not be added to a transaction partition if the transaction has fewer than three items. For example, T000: {1, 2} is not added to the transaction partition 1, since it only has two items. Some items in the original dataset may not be included in any of the transaction partitions, because the infrequent one/two-itemsets are dropped in the IAG. This simplifies the subsequent computations. In this example, however, all the items are kept in the IAG because the IAG is a relatively dense graph. Figures 19 and 20 show the transaction partitions.

TID	Items
T001	1, 2, 3
T005	1, 2, 3

Figure 24 Transaction Partition 1

TID	Items
None	None

Figure 25 Transaction Partition 2

Note that there is no transaction in partition 2. This is because partition 2 has only two unique items, it is not possible to mine any frequent three itemsets from this partition. Therefore, partition 2 is discarded.

In this step, TID T003 and T006 are marked as *divided* transactions, because they have elements that are divided between two different IAG partitions. The first partition is not enough to discover the potential frequent itemset {1, 4, 5}.

To avoid the loss of the divided transactions, in the scalable precise algorithm, a *bridge* transaction partition is constructed based on the divided transactions from the last step, as shown in Figure 26.

TID	Items
T003	1, 4, 5
T006	1, 4, 5

Figure 26 Bridge Transaction Partition

According to the algorithm, the above bridge transaction partition is converted into an IAG and then bisected by the MLkP algorithm. The results are two partitions, [1, 4] and [5]. At the same time, similar to the previous step, both transactions are marked as divided and will be added to the next bridge partition. The following (Figure 27 and Figure 28) are two small bridge partitions generated from the first bridge. They are both empty because none of them contains any transactions of three items or more.

TID	Items
None	None

Figure 27 Partition1 of The Bridge

TID	Items
None	None

Figure 28 Partition 2 of The Bridge

From the above two partitions, a second bridge(shown in Figure 29) is derived using the generate-bridge function.

TID	Items
T003	1, 4, 5
T006	1, 4, 5

Figure 29 The Second Bridge

The second bridge has the same size as the first bridge, so we can stop here and discard the second bridge.

The next step is to pick the best algorithm and use it to find the frequent k-itemsets with k > 2. For this example, we choose the modified Apriori algorithm because it is faster for mining small datasets, and it avoids the process of finding the frequent one and two itemsets again. The results from partition 1 are shown in Figure 30:

Frequent Itemsets	Support
{1, 2, 3}	2

Figure 30 Frequent Itemsets from Transaction partition 1

Since the modified Apriori algorithm starts at finding three-itemsets, there are no additional frequent itemsets in the second partition, as shown in Figure 31.

Frequent Itemsets	Support
None	N/A

Figure 31 Frequent Itemsets from Transaction partition 2

Figure 32 shows the frequent itemset found in the bridge transaction partition.

Frequent Itemsets	Support
{1, 4, 5}	2

Figure 32 Frequent Itemsets from Bridge transaction partition

We can compute the final result by taking the union of the frequent itemsets from Figures 19, 20, 30, 31, and 32. The support of each frequent itemset is the maximum support among the results of all partitions.

Frequent Itemsets	Support
{1}	5
{2}	4
{3}	3
{4}	3
{5}	3
{1, 2}	3
{1, 3}	2
{1, 4}	2
{1, 5}	2
{2, 3}	3
{4, 5}	3
{1, 2, 3}	2
{1, 4, 5}	2

Figure 33 Frequent Itemset Final Results

After running the Apriori-ap-genrules algorithm, the rules shown in Figure 34 can be discovered.

Rules	Confidence
{2} → {1}	0.75
{3} → {2}	1
{5} → {1}	1
{2} → {3}	0.75
{5} → {4}	1
{4} → {5}	1
{1, 3} → {2}	1
{1, 4} → {5}	1
{1, 5} → {4}	1

Figure 34 Association Rule Generated

Comparing the result with the result of the Apriori algorithm, the algorithm is sound and complete. Every frequent itemset and rule is correct, and it finds all frequent itemsets as well as all the rules.

14. Concise Description of the SARL-Precise Algorithm

Step 1: Find itemsets with size one and two using the Apriori algorithm or direct generation algorithm.

Step 2: Construct the item association graph(IAG) from the result of step 1.

Step 3: Partition the IAG using multilevel k-way partitioning algorithm(MLkP)

Step 4: Partition the database according to the result of step 3, mark those transactions required to be assigned to the bridge partition

Step 5: Construct the bridge partition using the result from step 4. Recursively reduce the bridge partition into smaller partitions.

Step 5: Choose an algorithm to mine frequent itemset on each database partition based on the characteristic of different algorithms.

Step 6: Summarize the result. For each frequent itemset found in any partition, choose the highest support among all partitions.

15. Another Example

This example shows how the SARL algorithm works on a slightly more complex dataset, shown in Figure 35, which requires more recursions for bridge generation.

TID	Items
T000	1, 2, 4
T001	2, 4, 5
T002	2, 3, 4
T003	1
T004	1, 2, 3
T005	2, 3, 5
T006	1, 3, 4
T007	2, 3, 5
T008	2, 3

Figure 35 Another Example Dataset

Let us run the SARL algorithm. Firstly, the modified Apriori algorithm is used to find frequent one and two itemsets. The results are shown in Figures 36 and 37.

Frequent Itemsets	Support
{1}	4
{2}	7
{3}	6
{4}	4
{5}	3

Figure 36 Frequent One Itemsets

Frequent Itemsets	Support
{1, 2}	2
{1, 3}	2
{1, 4}	2
{2, 3}	5
{2, 4}	3
{2, 5}	3
{3, 4}	2
{3, 5}	2

Figure 37 Frequent Two Itemsets

Similar to the previous examples, the IAG is constructed according to the frequent two itemsets found above, as shown in Figure 38.

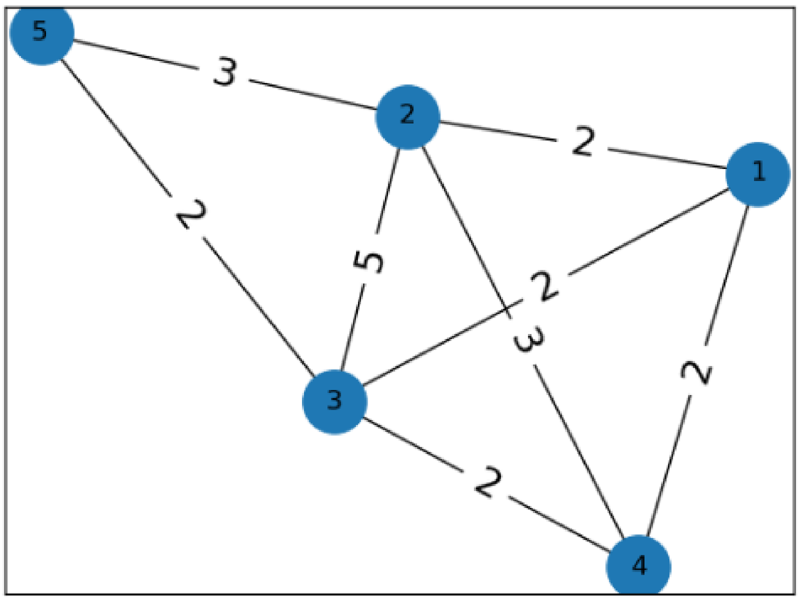


Figure 38 IAG for Example 2

The MLkP algorithm divides the IAG into two partitions, as shown in Figure 39 and Figure 40:

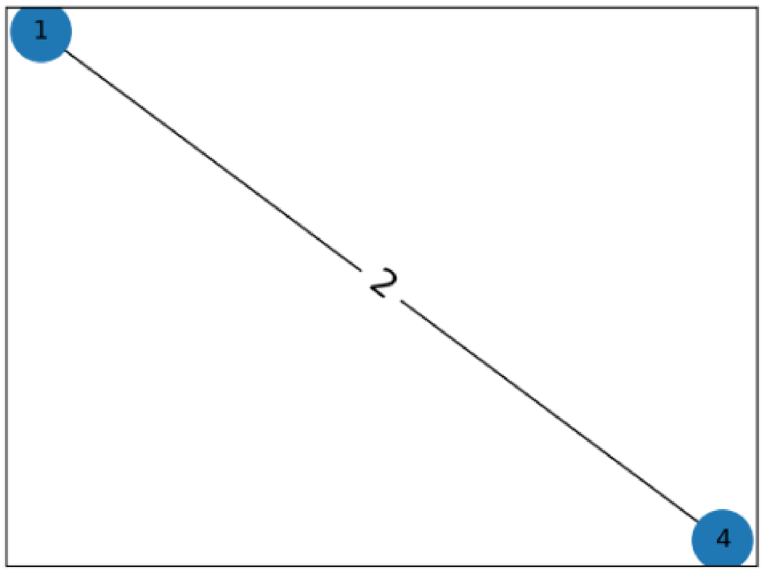


Figure 39 IAG Partition 1

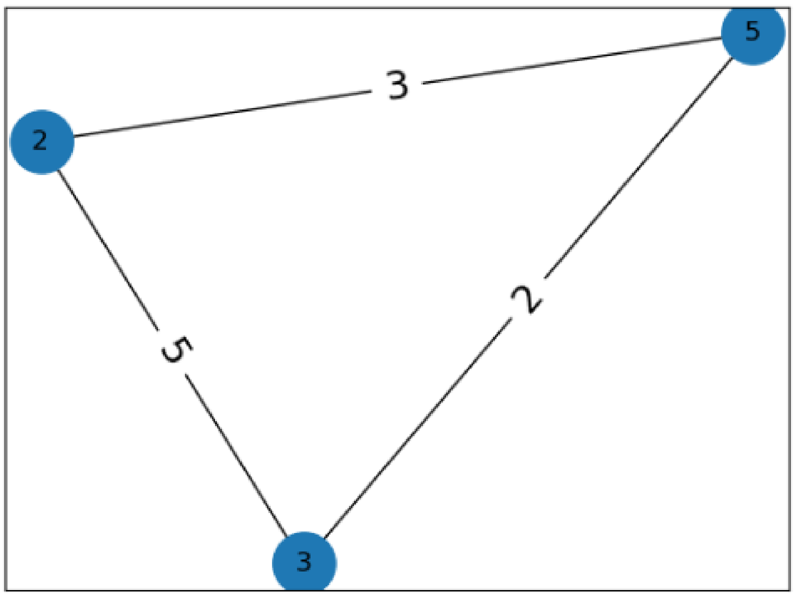


Figure 40 IAG Partition 2

Next, the SARL algorithm divides the original dataset into two transaction partitions, as shown in Figure 41 and Figure 42, according to the two IAG partitions above with the same method discussed in the previous example:

TID	Items
None	None

Figure 41 Transaction partition 1

TID	Items
T005	2, 3, 5
T007	2, 3, 5

Figure 42 Transaction partition 2

In this process, we mark each divided transaction and put its TID into a list if that transaction has three or more items: [T000, T001, T002, T004, T006]

Next, the first bridge partition is constructed with the transactions in the list above, as shown in Figure 43.

TID	Items
T000	1, 2, 4
T001	2, 4, 5
T002	2, 3, 4
T004	1, 2, 3
T006	1, 3, 4

Figure 43 The First Bridge Partition

Now, the first bridge partition is treated as the original dataset to generate an IAG, partitioned by the MLkP algorithm ([1,3] and [2, 4, 5]), and generates two small bridge partitions, as shown in Figure 44 and Figure 45:

TID	Items
None	None

Figure 44 Small Bridge Partition 1

TID	Items
T001	2, 4, 5

Figure 45 Small Bridge Partition 2

A second bridge partition can be built during this process, as shown in Figure 46:

TID	Items
T000	1, 2, 4
T002	2, 3, 4
T004	1, 2, 3
T006	1, 3, 4

Figure 46 The Second Bridge Partition

Comparing to the previous bridge, the second bridge has one fewer transaction and one fewer unique item. Therefore, we have a reduced bridge partition. Since the decomposition is lossless, the first bridge can be safely discarded.

If we repeat this process one more time, the IAG partitions are [1,2] and [3, 4]. Then we can obtain two empty bridge partitions and the bridge partition shown in Figure 47.

TID	Items
T000	1, 2, 4
T002	2, 3, 4
T004	1, 2, 3
T006	1, 3, 4

Figure 47 The Third Bridge Partition

This third bridge partition is the same partition as the second one. Therefore, we can discard this one. In summary, we have losslessly transformed the original dataset into the partitions shown in Figures 48, 49, and 50.

TID	Items
T005	2, 3, 5
T007	2, 3, 5

Figure 48 The Transaction Partition 1

TID	Items
T001	2, 4, 5

Figure 49 The Transaction Partition 2

TID	Items
T000	1, 2, 4
T002	2, 3, 4
T004	1, 2, 3
T006	1, 3, 4

Figure 50 The Second Bridge Partition

After running the modified Apriori algorithm on each of the partitions. We can find the frequent itemset shown in Figure 51:

Frequent Itemsets	Support
{2, 3, 5}	2

Figure 51 Frequent Itemsets Found

The example shown above has only one frequent three itemset. However, according to Theorem 2, given a low *minsup* and a large dataset, there should be more frequent three or more itemsets than frequent two itemsets.

We can find the union of all frequent itemsets to compute the final result, as shown in Figure 52. The support of each frequent itemset is the maximum of those across the results from all partitions for the same itemset.

Frequent Itemsets	Support
{1}	4
{2}	7
{3}	6
{4}	4
{5}	3
{1, 2}	2

{1, 3}	2
{1, 4}	2
{2, 3}	5
{2, 4}	3
{2, 5}	3
{3, 4}	2
{3, 5}	2
{2, 3, 5}	2

Figure 52 Final Frequent Itemsets

By running the Apriori-ap-genrules algorithm, the rules shown in Figure 53 can be found.

Rules	Confidence
{2} → {3}	0.71
{3} → {2}	0.83
{5} → {2}	1
{4} → {2}	0.75
{3, 5} → {2}	1

Figure 53 Association Rules

Again, both frequent itemsets and association rules found by the SARL algorithm are sound and complete.

16. Formalized Algorithm

Following is the pseudo-code for the SARL-Precise Algorithm.

(12) Pseudo Code

```
SARL_Precise:
bridges = []
results, two_itemset = mod1-Apriori(dataset)
graph = build_IAG(two_itemset)
partitions = METIS.partition(k, graph)
files, div_index = partition-dataset(dataset, partitions)
bridges = generate-bridge(files, div_index, k, dataset, 0, infinity)
files += bridges
for file in files:
        results += mod2-Apriori(file)  #when files are small
        results += FP-Growth(file) #when files are large
```

```
rules = Apriori-gen(results)
```

mod1-Apriori(dataset):

```
C1 = {}
for transaction in dataset:
        for item in transaction:
                if item not in C1:
                        add item to C1
          item.counter = 1
                else:
                        item.counter += 1
L1 = {}
for candidate in C1:
        if candidate.counter >= *minsup*:
                add candidate to L1
C2 = {}
for itemset1 in L1:
        for itemset2 in L1:
                if itemset1 != itemset2:
                        add itemset1 U itemset2 to C2
for transaction in dateset:
        for candidate in C2:
                if candidate.issubset(transaction):
                        candidate.counter += 1
L2 = {}
for candidate in C2:
        if candidate.counter >= *minsup*:
                add candidate to L2
return L1, L2
```

build_IAG(itemsets):

```
for itemset in itemsets:
        graph.add_node(itemset[0])
   graph.add_node(itemset[0])
   graph.add_edge(itemset[0], itemset[1], weight += 1)
return graph
```

partition-dataset(dataset, partitions):

```
div_index = []
for transaction in dataset:
        for partition in partitions:
                intersect = parition intersect transaction
                if len(intersect) < len(transaction):
                   div_index.append(TID_of_transaction)
             elif len(intersect) > 2:
             add intersect to dataset_partition_i
```

```
return dataset_partition_names, div_index
```

generate-bridge(div_index, k, dataset, i, last_len):

```
    count = 0
    for TID in div_index:
        bridge_i. add (dataset[TID])
        count+=1
    if count == last_len:
        return bridge_i
    else:
        dataset = bridge_i
        results, two_itemsets = mod1-Apriori(dataset)
        graph = build_IAG(two_itemset)
        partitions = METIS.partition(k, graph)
        temp_files, div_index = partition-dataset(dataset, partitions)
        return generate-bridge( div_index, k, dataset, i+1, count), temp_files
```

17. Recursively Generating Bridge Transaction Partitions

The bridge dataset and their partitions are necessary to achieve a precise calculation of frequent three or more itemsets. The main purpose of deriving them is to reconsider all potential errors, so the SARL algorithm does not miss any frequent itemsets.

The algorithm is greedy and defined recursively. In each recursion, it first finds all divided transactions that were labeled during the transaction partition step, and they are added to the bridge partition. The divided transactions were labeled so that there is no need to make a whole pass of the dataset again to check if a transaction is divided. Then, it calculates the number of transactions in the bridge partition and compares it to the previous, undivided bridge partition. If the current bridge partition has the same number of transactions as the previous one, that means the current reduction does not reduce the size of the bridge partition any further. The algorithm discards the current bridge partitions. On the other hand, if the total number of transactions in the new bridge partition is less than that of the previous bridge partition, then the algorithm divides the bridge partition again.

Some additional transaction partitions are generated through this process. However, calculating frequent itemsets in these smaller partitions are relatively simple since they are small enough to fit into the memory and contain fewer unique items than their parent bridge partition. Reducing the size of the bridge partition could result in an exponential reduction in the complexity of finding frequent itemsets, as we analyzed earlier.

18. Analysis

The time and the space complexities of the SARL-Precise algorithm, when the modified Apriori algorithm is chosen, are the same as those of the Apriori algorithm. That is, $\boldsymbol{O}(\mathbf{2}^{\boldsymbol{d}})$ for both time and space complexities. From our previous analysis, the SARL-Heuristic algorithm has a complexity of $\boldsymbol{O}(\mathbf{2}^{\frac{\mathbf{1.03}\boldsymbol{d}}{\boldsymbol{k}}})$ for both time and space. In the most extreme case, every transaction of the original dataset is divided, then the bridge partition will be the same as the original dataset, and the total time complexity will be $\boldsymbol{O}\left(\mathbf{2}^{\frac{\mathbf{1.03}\boldsymbol{d}}{\boldsymbol{k}}}\right)+\boldsymbol{O}(\mathbf{2}^{\boldsymbol{d}})$ = $\boldsymbol{O}(\mathbf{2}^{\boldsymbol{d}})$. Similarly, the total space complexity is also $\boldsymbol{O}(\mathbf{2}^{\boldsymbol{d}})$. However, the SARL-Precise algorithm runs faster than the Apriori algorithm in most cases. This is because the MLkP algorithm finds the sub-optimized solution to cut the minimum number of transactions. Thus, the size of the bridge is usually much smaller than the original dataset.

19. Theorems and Proofs

Theorem 5: Each decomposition of a bridge partition or the original dataset is lossless.

Proof:

After dividing the original dataset into two transaction partitions, there are two possibilities for each transaction of the original dataset. Each transaction T must be either divided and assigned to different partitions or assigned to a single partition as a whole. For the latter case, all existing subsets of T, including duplications of T, must also be assigned into the same partition. We can say that partition contains the complete information for any itemsets that are subsets of T. Therefore, both the Apriori algorithm and the FP-growth algorithm will get the same support for frequent itemsets that are subsets of T. As for the former case, the bridge partition B includes all divided transactions. If a frequent itemsets F is not a subset of any transaction T, T being a transaction completely assigned to P_i , then all transactions K that are supersets of F must be assigned to B. On the other hand, if F is a subset of some transaction T, being a transaction completely assigned to P_i, then this will be the latter case. The former case can be described as,

$$F \subset \begin{cases} T \mid T \in P_i \ \wedge (\forall t \subset T) \in P_i \\ K \mid K \in B \ \wedge (\forall k \subset K) \in B \end{cases}$$

Therefore, for each frequent itemset F, all transactions that contain any supersets of F are assigned to only one partition. So all support numbers are guaranteed to be the same as the ones before decomposition.

For the same reasons, all subsequent decompositions are performed on the bridge partitions and are lossless as well.

EXPERIMENTS AND RESULTS

We designed and conducted experiments on both small and large datasets to demonstrate the scalability of our algorithm. The experiments were performed on a computer with the following settings:

OS: Ubuntu 64-bit virtual machine

CPU: Intel Core i7-4720HQ

Memory: 8192MB allocated to the virtual machine

Disk: 5400RPM, 64MB Cache, 6.0Gb/s, SSHD, 8GB flash memory

Programming Language: Python 3.7

The datasets [3] we use include Mushroom [9], T10I4D100K [3], and T40I10D100K [3]. The details of each dataset will be covered later.

For each of these datasets, we tested the SARL algorithm with various settings for the FP-Growth and the Apriori algorithms on different values of *minsup*. The various settings of the SARL algorithm are as follows:

2apF: k = 2, Apriori-based, heuristic mode.

2apT: k = 2, Apriori-based, precise mode.

2fpF: k = 2, FP-Growth-based, heuristic mode.

2fpT: k = 2, FP-Growth-based, precise mode.

4apF: k = 4, Apriori-based, heuristic mode.

4apT: *k* = 4, Apriori-based, precise mode.

4fpF: *k* = 4, FP-Growth-based, heuristic mode.

4fpT: *k* = 4, FP-Growth-based, precise mode.

The Mushroom Dataset

The mushroom dataset has the following metrics:

Number of unique items: 119

Number of transactions: 8124

Average transaction width: 23

File size: 558 KB

This is a small dataset considering its size. However, the complexity of mining frequent itemsets is non-trivial due to its large average transaction width. The experiments was done repeatedly for *minsup* of 20%, 10%, 7%, 4%, 1%, 0.7%, 0.4%, and 0.1%. The time limit for each experiment was set to 400 seconds for each of the 80 experiments. The results are shown in Figure 54:

	fp	sarl 2apF	sarl 2apT	sarl 2fpF	sarl 2fpT	sarl 4apF	sarl 4apT	sarl 4fpF	sarl 4fpT	ap
20	79.2578	62.78268		54.88132	126.1449	29.18472		13.04025	94.26079	
10	379.6436			56.5663	305.3277	129.0632		19.01914	269.448	
7	354.9317			113.5315		165.1814		17.03973	368.4053	
4				161.2162				44.68621		
1				362.2856				39.26179		
0.7				165.8576				51.73652		
0.4				386.2269				46.14297		
0.1								100.1229		

Figure 54 Experiments from the Mushroom Dataset

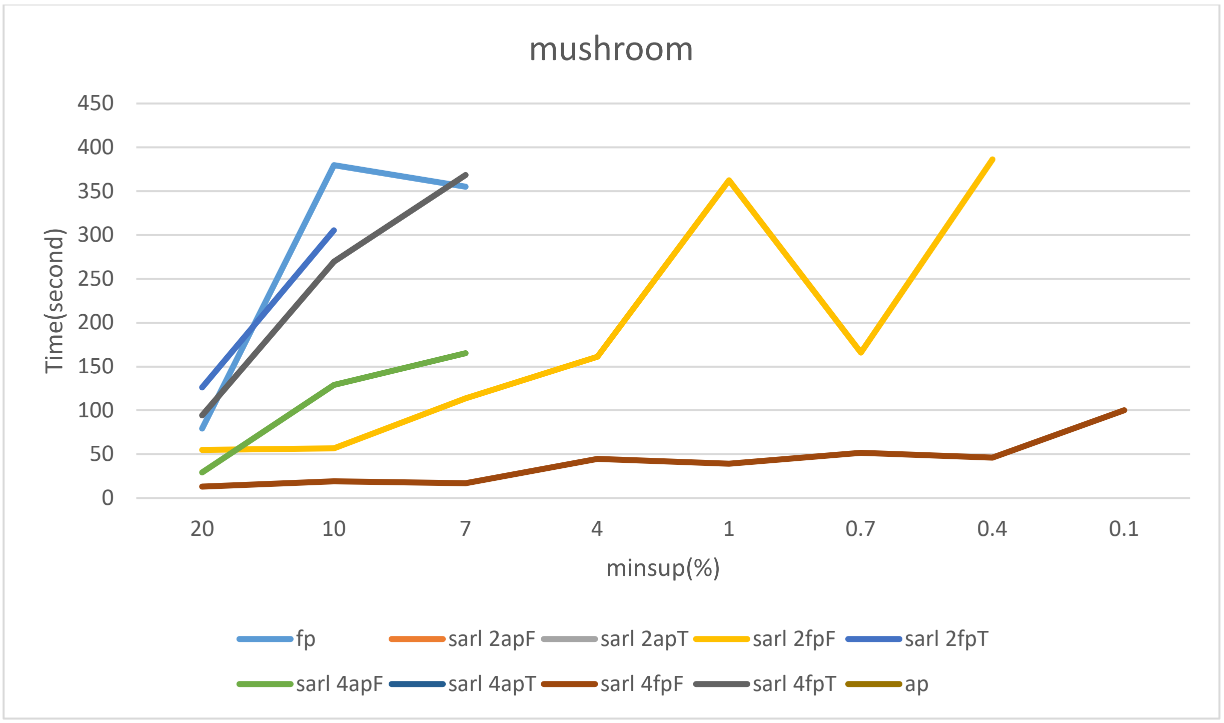


Figure 55 Mushroom Dataset Test Results

According to Figure 55, the results show that the four-partition, FP-Growth-based, heuristic SARL algorithm scales the best for this dataset regardless of the *minsup* value. It is 6 to 20.8 times faster than the FP-Growth algorithm when the FP-Growth algorithm does not exceed the time limit. Although it does not cover all frequent itemsets or association rules, depending on the application, this could be a great tradeoff. The two-partition, FP-Growth-based, heuristic SARL algorithm is the second-best in terms of running time. It has a higher accuracy with acceptable running time except for the test with 0.1% *minsup*. Two precise, FP-Growth based SARL algorithms mostly outperform the FP-Growth and the Apriori algorithms. The Apriori algorithm does not perform well on this dataset, and it does not terminate for any of the experiments. The average transaction width might have a high impact on the performance of the Apriori algorithm.

The second dataset we have tested is T10I4D100K. It has the following statistics:

Number of unique items: 870

The average size of transactions: 10

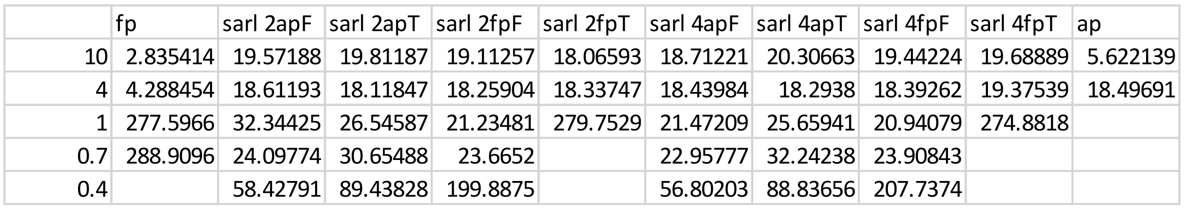
The average size of the maximal potentially large itemsets: 4

Number of transactions: 100000

File size: 4MB

The algorithms were tested on T10I4D100K for *minsup* of 10%, 4%, 1%, 0.7%, and 0.4%. This dataset has a medium size (for this environment), so the time limit is set to 300 seconds for each of the 50 experiments.

Figure 56 shows the results for T10I4D100K:



	fp	sarl 2apF	sarl 2apT	sarl 2fpF	sarl 2fpT	sarl 4apF	sarl 4apT	sarl 4fpF	sarl 4fpT	ap
10	2.835414	19.57188	19.81187	19.11257	18.06593	18.71221	20.30663	19.44224	19.68889	5.622139
4	4.288454	18.61193	18.11847	18.25904	18.33747	18.43984	18.2938	18.39262	19.37539	18.49691
1	277.5966	32.34425	26.54587	21.23481	279.7529	21.47209	25.65941	20.94079	274.8818	
0.7	288.9096	24.09774	30.65488	23.6652		22.95777	32.24238	23.90843		
0.4		58.42791	89.43828	199.8875		56.80203	88.83656	207.7374		

Figure 56 T10I4D100K Test Results

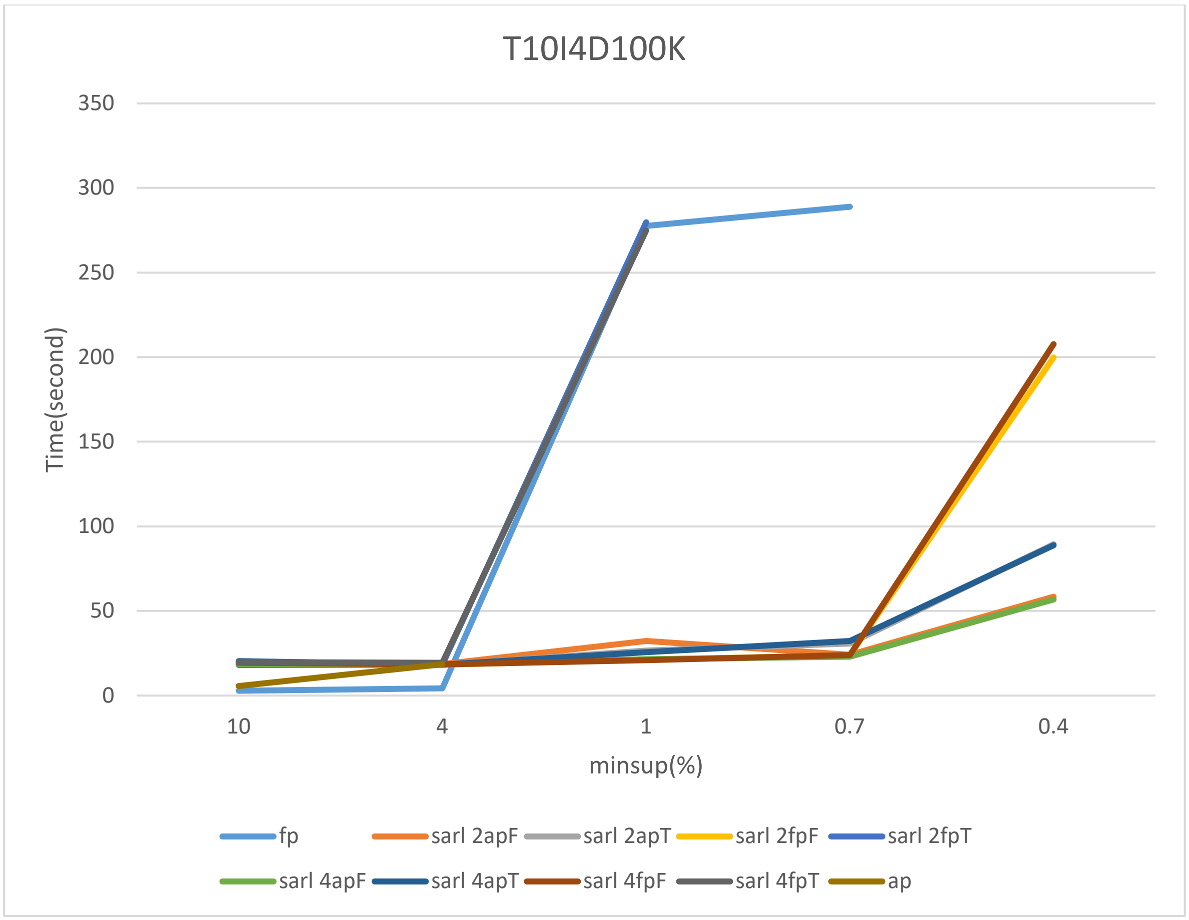


Figure 57 T10I4D100K Test Results

From Figure 57, the Apriori algorithm has an average performance for the initial *minsup* of 10% and 4%. However, it quickly reached the maximum running time after that and unable to finish the task in time for all subsequent *minsup*. The FP-Growth has a better performance. It was the fastest for higher *minsup* of 10% and 4%, but jumped to almost 300 seconds for 1% and 0.7%, before having timeout at 0.4%. All settings of the SARL algorithm outperform the Apriori and the FP-Growth algorithm for middle and low *minsup*. The SARL algorithm is slightly slower at a high *minsup* of 10%, and they were tied with the Apriori but was slightly slower than FP-Growth at *minsup* of 4%.

The dataset T40I10D100K has the following statistics:

Number of unique items: 942

The average size of transactions: 40

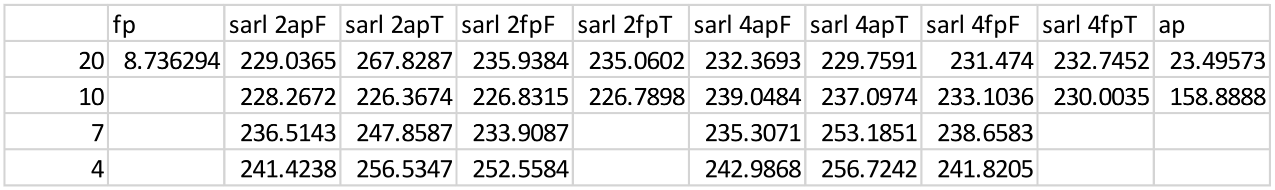
The average size of the maximal potentially large itemsets:10

Number of transactions:100000

File size: about 15 MB

This relatively large-size dataset was tested on *minsup* values of 20%, 10%, 7%, and 4%. The maximum running time was set to 300 seconds each for a total of 40 experiments.

Figure 58 shows the results of the experiments:



	fp	sarl 2apF	sarl 2apT	sarl 2fpF	sarl 2fpT	sarl 4apF	sarl 4apT	sarl 4fpF	sarl 4fpT	ap
20	8.736294	229.0365	267.8287	235.9384	235.0602	232.3693	229.7591	231.474	232.7452	23.49573
10		228.2672	226.3674	226.8315	226.7898	239.0484	237.0974	233.1036	230.0035	158.8888
7		236.5143	247.8587	233.9087		235.3071	253.1851	238.6583		
4		241.4238	256.5347	252.5584		242.9868	256.7242	241.8205		

Figure 58 T40I10D100K Test Results

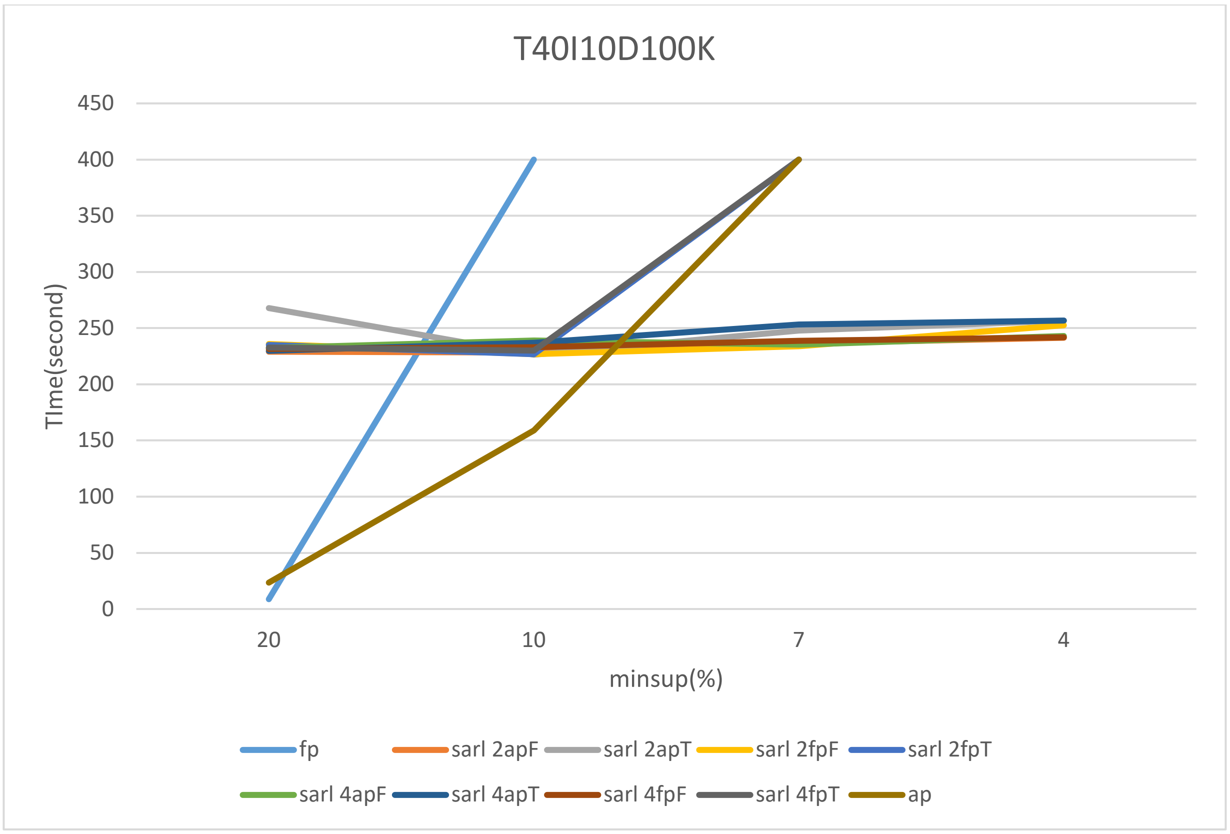


Figure 59 T40I10D100K Test Results

The results of the experiments(shown in Figure 58 and Figure 59) show an obvious distinction between the scalability of different algorithms. All settings of the SARL algorithm demonstrates very high scalability. Almost all settings of the SARL algorithm had stable running time throughout the entire range of *minsup*. Surprisingly, the Apriori algorithm performs better than the FP-Growth algorithm with *minsup* between 20% and 7%. However, it is still unable to terminate within the time limit for *minsup* = 4%. Lastly, the FP-Growth algorithm does not scale very well on this dataset. It failed to terminate within the given time for both 7% and 4% of *minsup*.

CONCLUSIONS AND FUTURE WORK

We have proposed a scalable, highly parallelizable association rule mining algorithm (SARL algorithm) in this paper. The contributions include the use of the divide-and-conquer method to speed up complex computations, the use of an item association graph that provides an efficient estimation of potential frequent itemsets, the use of the MLkP algorithm to divide the items into partitions while minimizing the loss of information, the generation of the bridge partition to achieve precise computation, and recursive reduction of the bridge partition. We have shown the scalability of the SARL algorithm through a series of experiments. The results indicate that the SARL algorithm has better scalability than both the Apriori and the FP-Growth algorithms in most cases.

In the future, we plan to extend our work on the following tasks:

- Develop the parallel version of the SARL algorithm and its implementation.
- Develop a scalable SARL algorithm to mine hierarchical item association rules.
- Study how characteristics of the datasets that influence the performance of the SARL algorithm.

REFERENCES

1. Agrawal, R., & Srikant, R. (1994, September). Fast algorithms for mining association rules. In Proc. 20th int. conf. very large data bases, VLDB (Vol. 1215, pp. 487-499).
2. Agrawal, R., Imieliński, T., & Swami, A. (1993, June). Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (pp. 207-216).
3. Agarwal, R. C., Aggarwal, C. C., & Prasad, V. V. V. (2001). A tree projection algorithm for generation of frequent item sets. *Journal of parallel and Distributed Computing*, *61*(3), 350-371.
4. A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning. J. Global Optimization, 29(2):225-241, 2004.
5. Baralis, E., Cerquitelli, T., Chiusano, S., & Grand, A. (2013, April). P-Mine: Parallel itemset mining on large datasets. In *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)* (pp. 266-271). IEEE.
6. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., & Schulz, C. (2016). Recent advances in graph partitioning. In *Algorithm Engineering* (pp. 117-158). Springer, Cham.
7. C. Borgelt, "Frequent item set mining," Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, vol. 2, no. 6, pp. 437–456, 2012.
8. Chee, C. H., Jaafar, J., Aziz, I. A., Hasan, M. H., & Yeoh, W. (2019). Algorithms for frequent itemset mining: a literature review. *Artificial Intelligence Review*, *52*(4), 2603-2621.

9. Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

10. Feddaoui, I., Felhi, F., & Akaichi, J. (2016, August). EXTRACT: New extraction algorithm of association rules from frequent itemsets. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)* (pp. 752-756). IEEE.

11. Galinier, P., Boujbel, Z., & Fernandes, M. C. (2011). An efficient memetic algorithm for the graph partitioning problem. *Annals of Operations Research*, *191*(1), 1-22.

12. Goethals, B. (n.d.). Frequent Itemset Mining Dataset Repository. Retrieved 2020, from http://fimi.uantwerpen.be/data/

13. Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. *ACM sigmod record*, *29*(2), 1-12.

14. Hoseini, M. S., Shahraki, M. N., & Neysiani, B. S. (2015, November). A new algorithm for mining frequent patterns in can tree. In *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)* (pp. 843-846). IEEE.

15. Houtsma, M., & Swami, A. (1993). *Set-oriented mining for association rules. IBM Almaden research center*. research report RJ 9567, San Jose.

16. J. Heaton, "Comparing dataset characteristics that favor the Apriori, Eclat or FP-Growth frequent itemset mining algorithms," SoutheastCon 2016, Norfolk, VA, 2016, pp. 1-7, doi: 10.1109/SECON.2016.7506659.

17. Karypis, G., & Kumar, V. (1998). Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, *48*(1), 96-129.

18. Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, *49*(2), 291-307.

19. Lin, X. (2014, June). Mr-apriori: Association rules algorithm based on mapreduce. In *2014 IEEE 5th international conference on software engineering and service science* (pp. 141-144). IEEE.

20. McSherry, F. (2001, October). Spectral partitioning of random graphs. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science* (pp. 529-537). IEEE.

21. Nadimi-Shahraki, M. H., & Mansouri, M. (2017, March). Hp-Apriori: Horizontal parallel-apriori algorithm for frequent itemset mining from big data. In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)(* (pp. 286-290). IEEE.

22. Naulaerts, S., Meysman, P., Bittremieux, W., Vu, T. N., Vanden Berghe, W., Goethals, B., & Laukens, K. (2015). A primer to frequent itemset mining for bioinformatics. *Briefings in bioinformatics*, *16*(2), 216-231.

23. Park, J. S., Chen, M. S., & Yu, P. S. (1995). An effective hash-based algorithm for mining association rules. *Acm sigmod record*, *24*(2), 175-186.

24. Pyun, G., Yun, U., & Ryu, K. H. (2014). Efficient frequent pattern mining based on linear prefix tree. *Knowledge-Based Systems*, *55*, 125-139.

25. Sanders, P., & Schulz, C. (2011, September). Engineering multilevel graph partitioning algorithms. In *European Symposium on Algorithms* (pp. 469-480). Springer, Berlin, Heidelberg.

26. Song, M., & Rajasekaran, S. (2006). A transaction mapping algorithm for frequent itemsets mining. *IEEE transactions on knowledge and data engineering*, *18*(4), 472-481.

27. Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE transactions on knowledge and data engineering*, *12*(3), 372-390.