Tuijin Jishu/Journal of Propulsion Technology
ISSN: 1001-4055
Vol. 45 No. 3 (2024)

Split and Win Apportioning Algorithm – Swaa to Discover Frequent Patterns in Large Database

[1]R.Manivasagan, [2]Dr.B.Senthilkumaran

[1]Research Scholar, Department of Computer science, Christhu Raj College, Affiliated to Bharathidasan University, Tiruchirappalli - 12.

[2]Assistant Professor & Research Advisor, Department of Computer science, Christhu Raj College, Affiliated to Bharathidasan University, Tiruchirappalli - 12.

Abstract:- Mining large datasets and discovering meaningful hidden patterns is not a new area but a lot of improvement is essential to overcome the cost and operational overheads, this paper finds a solution by splitting the large dataset finding the individual partition support count (IPSC) and then the partitioned dataset are merged to find the merged partitioned support count (MPSC) to reduce the burden of time and memory related issues. To find the IPSC and MPSC simple bit vector approach is utilized. The proposed algorithm is compared with the other existing algorithms to gauge its performance with respect to speed and the memory consumption.

Keywords: Frequent Patterns, Large Datasets, Split and Win Apportioning Algorithm, Partition Support Count, Bit Vector Approach

1. Introduction

Retrieving a collection of items that frequently occur in the dataset and whose count is at least as high as the user-specified minimum support threshold value is known as mining frequent itemset. The support count, which should be higher than the user-provided threshold number, indicates how frequently objects are present in the transactional database. To extract the frequent itemset from the transactional database, a variety of methods and algorithms have been developed. In this field, the Apriori algorithm is a pioneering work that was developed by Srikanth Agarwal and his colleagues [1]. Based on the database being used, the new approach proved to be cost-effective. Other researchers have improved, modified, customized, and altered the Apriori algorithm, which has been called the pioneering work in the field. The next popular algorithm is FP-Growth which scans the entire database only twice unlike apriori but the entire database should fit in the memory of the system and consumes a lot of memory. The aforementioned methods, while well-known, nonetheless have several drawbacks. For example, the FP-Growth algorithm necessitates a large amount of memory space in order to maintain the full database, while Apriori constantly searches the database and generates numerous candidates for it. In order to get around these obstacles, the suggested method divides the whole database into individual processors and processes each partition independently, reducing the number of candidates and memory usage. The results are then combined to identify recurring patterns. In order to produce frequent itemsets from very big datasets, Zaki'sparEclat [3] technique leverages the idea of parallel computing in conjunction with vertical data representation. An application of the Apriori method in parallel using map reduces is the Single Pass Counting SPC algorithm [4]. The algorithm functions effectively by addressing the shortcomings of the traditional Apriori algorithm. Here, the support count of the candidates is parallelized, and the entire process is divided into two stages.

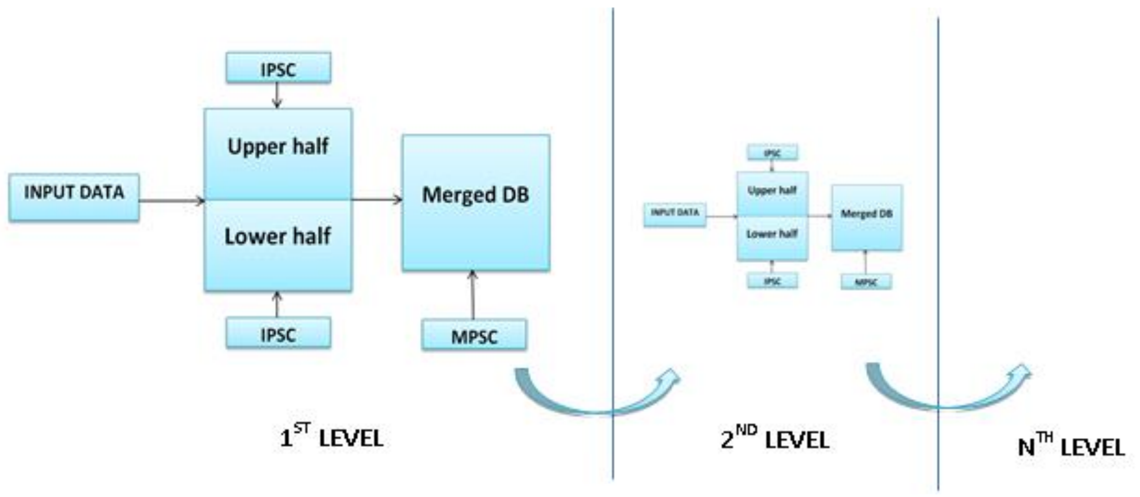


Figure 1: Overall architecture of the proposed method

The figure 1 showcased above is self-explanatory as the input raw data is fed initially, the data is bifurcated into upper half and lower half, then the individual partition support count for both upper and lower are found. Next the partitioned data is merged to find the merged partition support count. Here the unpromising items are pruned and the process continues until the entire meaningful frequent patterns are discovered. The sample dataset used to discover the meaningful frequent patterns are shown in the table 1 and this dataset is partitioned into two portions one from top to the middle and the other from the bottom to the middle.

Table.1. Sample dataset

Transaction ID	**Items**
1	M1, M2, M3, M5, M6, M15
2	M1, M3, M7
3	M5, M9
4	M1, M3, M4, M5, M7
5	M1, M3, M5, M7, M12
6	M5, M10
7	M1, M2, M3, M5, M6, M16
8	M1, M3, M4
9	M1, M3, M5, M7, M13
10	M1, M3, M5, M7, M14

The above shown dataset is bifurcated using the procedure Top Down Bottom Up as shown in the following table 2.

Top Down partition – P1			**Bottom Up partition – P2**	
TID	**Items**		**TID**	**Items**
1	M1, M2, M3, M5, M6, M15		**1**	M1, M3, M5, M7, M14
2	M1, M3, M7		**2**	M1, M3, M5, M7, M13
3	M5, M9		**3**	M1, M3, M4

4	M1, M3, M4, M5, M7		**4**	M1, M2, M3, M5, M6, M16
5	M1, M3, M5, M7, M12		**5**	M5, M10

Table 2: Bifurcated sample dataset

The pseudo code to bifurcate the dataset is shown below and the procedure Top Down Bottom Up is self-explanatory.

```
Procedure Top Down Bottom Up ( Input dataset D)

INPUT: raw data D

OUTPUT: Bifurcated data P1 and P2

BEGIN:

1.    Load the input dataset D
2.    Find the total row count Rcount
3.    For index = 0 to Rcount-1 do
4.    Store in P1 = D[index]
5.    Store in P2 = D[Rcount – index]
6.    IF ( index = Rcount-index)
7.    EXIT  For loop
8.    Close IF
9.    Close FOR

END Procedure
```

The bifurcated data is assumed to be sorted and the distinct items present in the partitions P1 and P2 are found using the procedure Discover Distinct which is shown below along with the individual count of the items.

```
Procedure Discover Distinct (Partitioned data P)

INPUT: Partitioned data P

OUTPUT: Distinct Item and its count

BEGIN:

1.    Load the partitioned dataset P
2.    DistArr[ ][ ] = empty
3.    ∀ transactional Row TR ∊ P do
4.    ∀ Item II ∊ TR do
5.    IF ( II not in DistArr[ ] ) do
6.    II →DistArr[ ]
7.    Count II individually →DistArr
8.    Close IF
9.    Close For
10.   Close For
11.   Return DistArr

END Procedure
```

The transactional rows' are iterated in a loop to acquire each row individually from the partitioned data. After determining the total number of items in the individual transactional row, each item is retrieved one at a time and checked against the distinct items DistArr[]. If the item is not found in DistArr[], it is placed there; if not, the item count is increased. During the first iteration in the loop, the transactional row 1 = {M1, M2, M3, M5, M6, M15} is fetched. Now the total number of items present in the row1 is computed and found to contain 6 elements.

The inner loop is executed 6 times to discover the distinct elements.The first item retrieved is M1 and compared with the empty DistArr[], since the item "M1" is not present in the DistArr[], the first distinct element found is "M1" and stored in DistArr[].

ITERATION 2: "M2" not available in DistArr[ ], store "M2"

ITERATION 3: "M3" not available in DistArr[ ], store "M3"

ITERATION 4: "M5" not available in DistArr[ ], store "M5"

ITERATION 5: "M6" not available in DistArr[ ], store "M6"

ITERATION 6: "M15" not available in DistArr[ ], store "M15"

Inner loop closes

Outer loop iteration 2 commences,

Transactional row 2 in P1 = {M1, M3, M7} is initially fetched

The item count of the row2 is found to be 3 and the inner loop iterates three times as shown

ITERATION 1: "M1" available in DistArr[ ], increment count of M1

ITERATION 2: "M3" available in DistArr[ ], increment count of M3

ITERATION 3: "M7" not available in DistArr[ ], store "M7"

Inner loop ends

Similarly all the rows are fetched to discover the distinct elements and the resultant is shown in the table 3.

Table 3: Distinct element found from two partitions P1 and P2

PARTITION 1		**PARTITION 2**	
DISTINCT ELEMENT	**IPSC**	**DISTINCT ELEMENT**	**IPSC**
M1	4	**M1**	4
M2	1	**M2**	1
M3	4	**M3**	4
M4	1	**M4**	1
M5	4	**M5**	4
M6	1	**M6**	1
M7	2	**M7**	3
M9	1	**M10**	1
M12	1	**M13**	1
M15	1	**M14**	1`
		M16	1

Let us assume that the user defined support count is 4, the count which is less than 4 will be pruned and the final distinct element is shown in the following table 4. The elements M2, M4, M6, M9, M10, M11, M12, M13, M14, M15, M16 are pruned away and the remaining elements are shown in the table 4.

Table 4: Final distinct element after pruning

DISTINCT ELEMENT	MPSC
M1	8
M3	8
M5	8
M7	5

Property 1

"If an element's minimum support count is less than the user-specified minimum Support threshold value, it is deemed to be an infrequent element."

The final dataset is shown in the table 5 after pruning the infrequent elements whose support count value is less than the user specified,

Table 5:

Top Down partition – P1			Bottom Up partition – P2	
TID	**Items**		**TID**	**Items**
1	M1, M3, M5		**1**	M1, M3, M5, M7
2	M1, M3, M7		**2**	M1, M3, M5, M7
3	M5		**3**	M1, M3
4	M1, M3, M5, M7		**4**	M1, M3, M5
5	M1, M3, M5, M7		**5**	M5

Pruned sample dataset

Creating a bit vector table that corresponds to the database that has been pruned is the next step in the technique. The element in dataset will be indicated as "1" if it is present in the transactional row and "0" if it is not present.

The distinct elements are confined into four elements {M1, M3, M5, M7} and these distinct values are marked for its present in a row with "1" and "0" if it is not present. The first element M1 is marked in partition P1 as shown in the following section,

Table 6: Transactional Data Representation for Element M1

Element	**Transactional Row**				
	1	**2**	**3**	**4**	**5**
M1	1	1	0	1	1

Here the element M1 is not present in the row 3 if partition P1 and it is represented by "0" whereas all other rows contains "1" as it is present.

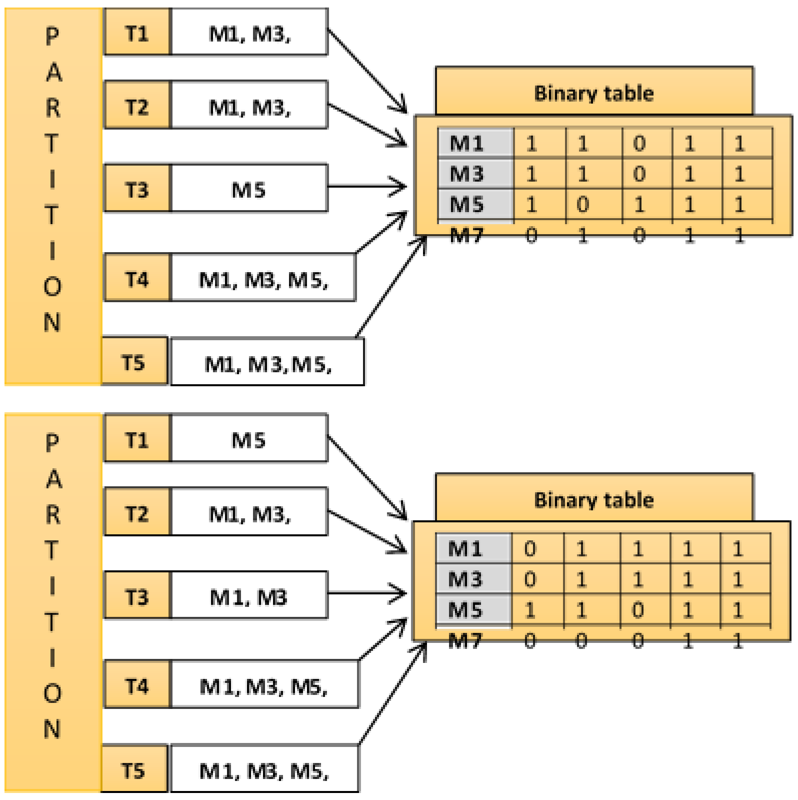


Figure 2: Bit vector table creation for P1 and P2

Next the elements are formed using some properties and definitions shown below,

Property 2

"An element or itemset might be infrequent in one partition but after merge that element or itemset might be frequent."[7]

{(M1, M2), IPSC} ≠ {(M1, M2), MPSC}

Property 3

"It is possible for an infrequent candidate in one partition to be a frequent candidate in another, and vice versa for frequent candidates in different partitions."[8]

```
Procedure CanFORM(Partioned data P, DistinctElement E)
INPUT: data P, Distinct element E
OUTPUT: Itemsets
BEGIN
Load the partitioned data P
Load the Distinct element E
∀ Element i ∊ P do
∀ Item  Di ∊ E do
   IF [Di not present in Pi AND Di > Pi] then
       Append Pi and Di →RES
       Close IF
  Close For
Close For
Return RES
END
```

This process is carried out in each and every level and then merged to get the overall itemset with MPSC value. The procedure shown above is used to form the itemsets with 2 element, 3 element and n elements. The first partition P1 is fed as an input which comprises of the following elements

{M1, M3, M5, M7}

- ***First iteration:*** Item M1 is fetched, and concatenated with the other elements to form {M1, M3}, {M1, M5}, {M1, M7}
- ***Second iteration:*** Item M3 is fetched and concatenated with other elements to form {M3, M5}, {M3, M5}
- ***Third iteration:*** Item M5 is fetched and concatenated with other elements to form {M5, M7}

The IPSC is found using the bit table values formed in the individual partition shown in the figure 4.

M1 = 1 1 0 1 1

M3 = 1 1 0 1 1 &

M1, M3 = 1 1 0 1 1 $\rightarrow$ IPSC = 4

M1 = 1 1 0 1 1

M5 = 1 0 1 1 1 &k

M1, M5 = 1 0 0 1 1 $\rightarrow$ IPSC = 3

M1 = 1 1 0 1 1

M7 = 0 1 0 1 1 &

M1, M7 = 0 1 0 1 1 $\rightarrow$ IPSC = 3

M3 = 1 1 0 1 1

M5 = 1 0 1 1 1 &

M3, M5 = 1 0 0 1 1 $\rightarrow$IPSC = 3

M3 = 1 1 0 1 1

M7 = 0 1 0 1 1 &

M3, M7 = 0 1 0 1 1 $\rightarrow$IPSC = 3

M5 = 1 0 1 1 1

M7 = 0 1 0 1 1 &

M5, M7 = 0 0 0 1 1 $\rightarrow$IPSC = 2

Table 7: Individual partition support count values found

Partition P1	IPSC	Partition P1	IPSC
M1 M3	4	M1 M3	4
M1 M5	3	M1 M5	3
M1 M7	3	M1 M7	2
M3 M5	3	M3 M5	3
M3 M7	3	M3 M7	2
M5 M7	2	M5 M7	2

Table 8: Merged partition support count values found

MERGED	MPSC	BINARY
M1 M3	8	1101101111
M1 M5	6	1001101011
M1 M7	5	0101100011
M3 M5	6	1001101011
M3 M7	5	0101100011
M5 M7	4	0001100011

The three element candidate is found using simple AND operation as shown in the following operation,

M1, M3 = 1101101111

M5 = 1011111011&

M1 M3 M5 = 1011101011→ IPSC = 7

M1, M5 = 1001101011

M7 = 0101100011&

M1 M5 M7 = 1001101011→ IPSC = 4

M3, M5 = 1001101011

M7 = 0101100011&

M3 M5 M7 = 1001101011→ IPSC = 4

M1 M3 M5 = 1011101011

M7 = 0101100011&

M1 M3 M5 M7 = 0001100011 → IPSC = 4

The frequent pattern output formed with the user defined minimum support count value are found and they are {M1, M3, M5, M7, M1M3, M1M5, M1M7, M3M5, M3M7, M5M7, M1M3M5M7}

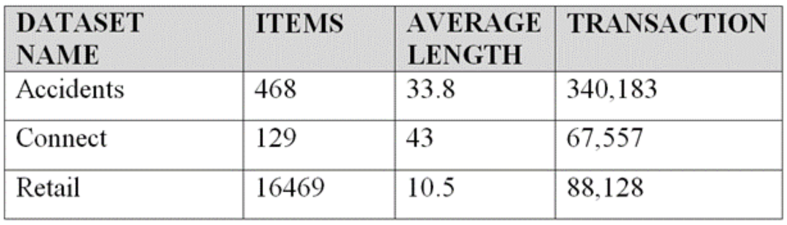
```
ALGORITHM SWAA( INPUT Data Đ, MinSup M)

INPUT: RAW data Đ
OUTPUT: Frequent Itemset
BEGIN:
1.    Load the input data Đ
2.    PP=TopDownBottomUp(D)
3.    DiscoverDistinct(PP)
4.    Can= CanForm(D, Distinct)
5.    Calculate IPSC
6.    Merge data from different partition
7.    Calculate MPSC
8.    If (MPSC >= M)
9.    Store the candidate→RES
10.   End IF
11.   Return RES
```

```
END
```

2. Experimantal Evaluation

The SPMF tool, which is based on the Java programming language, is used for coding, and large synthetic datasets [2] as well as benchmarked real datasets are utilized in tests to assess the performance of the proposed technique. For this experiment, six nodes are employed, and each system setup consists of an Intel Core I7 CPU, 8GB RAM, and a 1TB SDD drive. The benchmarked dataset used is shown in the table 8

Table 9: Dataset used



DATASET NAME	ITEMS	AVERAGE LENGTH	TRANSACTION
Accidents	468	33.8	340,183
Connect	129	43	67,557
Retail	16469	10.5	88,128

The Experiments are carried out for the number of candidate generated during the process and the candidate produced are noted and compared with the state of the art existing algorithms as shown in the tables below,

Table 10: Assessment of the number of candidates generated by experimentation on the Accident dataset

CANDIDATE GENERATION ASSESSMENT					
DATASET NAME - ACCIDENT					
Algorithm Name	User defined Minimum Support Values				
	0.10	0.20	0.25	0.35	0.40
BIGFIM	986187	532210	468127	220921	168657
PARECLAT	798423	412392	376915	201125	167218
SPC	575327	318175	259117	189890	162035
SWAA	420947	278616	200196	177618	145838

Table 11: Assessment of the number of candidates generated by experimentation on the CONNECT dataset

CANDIDATE GENERATION ASSESSMENT					
DATASET NAME - CONNECT					
Algorithm Name	User defined Minimum Support Values				
	0.15	0.20	0.25	0.30	0.45
BIGFIM	136127	112072	88878	68107	42137
PARECLAT	121398	105981	82826	59021	38681
SPC	99370	86596	72090	51536	38403
SWAA	81146	78757	68709	45765	34267

Table 12: Assessment of the number of candidates generated by experimentation on the RETAIL dataset

CANDIDATE GENERATION ASSESSMEMT					
DATASET NAME - RETAIL					
Algorithm Name	**User defined Minimum Support Values**				
	0.25	**0.30**	**0.35**	**0.40**	**0.55**
BIGFIM	186325	172927	163675	89327	72618
PARECLAT	212089	206916	172615	116673	75575
SPC	186878	175113	166124	90912	74581
SWAA	171595	160991	153067	83268	72147

From the above tables it is quite obvious that the proposed SWAA algorithm out- performed the other three existing algorithms with respect to candidate generation and produced fewer candidates during the execution which reduces the runtime considerably. The runtime comparison for varying support value and for varying number of processors is shown in the following figures.

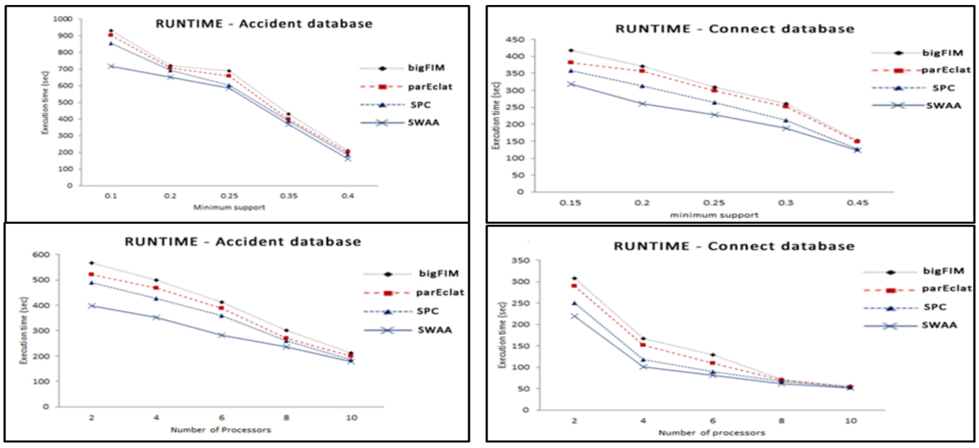


Figure 3: Runtime comparison for varying support and number of processors used

The above figure clearly indicates the effective and efficient performance of the proposed SWAA algorithm with respect to the speed of execution and out-performed the other algorithms by a good margin for varying user defined minimum support value as well as for varying number of nodes or processors. Similarly the memory usage is compared for the proposed SWAA with the existing algorithms and the results are showcased in the following figures and the result proved to be exemplary for the proposed SWAA algorithm.

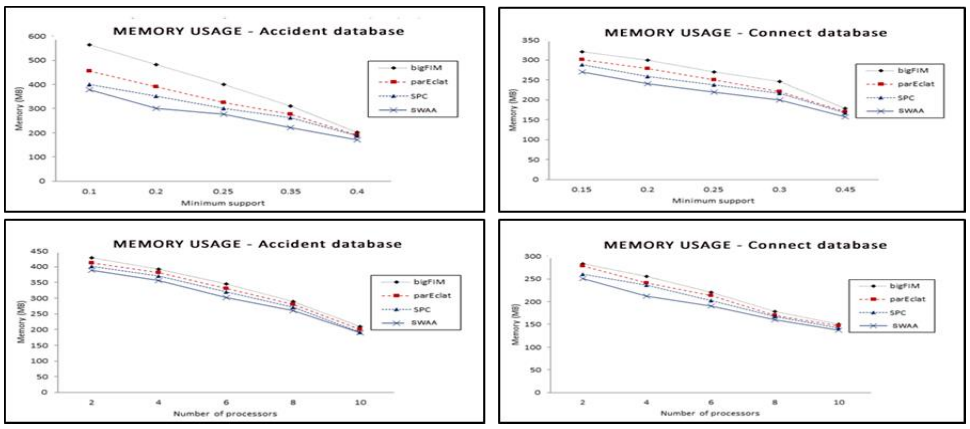


Figure 4: Memory usage comparison for varying support and number of processors used

3. Conclusion

The proposed SWAA algorithm is showcased in this paper and the important flaws that are prevalent in the existing algorithms are eradicated and from the experimental result, it is proved that the proposed algorithm produced fewer numbers of candidates due to the efficient pruning mechanism and this thereby reduces the runtime, memory consumption and increases the speed of the execution.

Refrences

[1] R. Agarwal and R. Srikant.Mining Sequential Pattern. In Proc. 1995 Int. Conf. Data Engineering, pages 3-10, 1995.
[2] Quest Data Mining Project, available at http: www.almaden.ibm.com/ cs/quest/syndata.html., IBM Almaden Research Center, San Jose, CA 95120.
[3] Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. Machine Learning, 42(12):31-60, January 2001
[4] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In Proceedings of the Sixth International Conference on Ubiquitous Information Management and Communication, ICUIMC ’12, pages 76:1 -76:8, New York, NY, USA, 2012.
[5] Sandy Moens, EminAksehirli, and Bart Goethals.Frequent itemset mining for big data. In 2013 IEEE International Conference on Big Data, pages 111-118. IEEE, 2013.
[6] I. Ali, "Application of Mining Algorithm to find Frequent Patterns in a Text Corpus," International Journal of Software Engineering and its Applications, vol. 6, no. 3, pp. 127-134, 2012.
[7] E. Ozkural, B. Ucar, and C. Aykanat. Parallel frequent item set mining with selective item replication. IEEE Trans. Parallel Distrib.Syst., pages 1632–1640, 2011.
[8] P. G. SanjeevRao, "Implementing Improved Algorithm Over Apriori Data Mining Association Rules Algorithm," International Journal of Computer Science and Technology, vol. 3, pp. 489-493, 2012.