Coverage Pattern Mining Based on Map Reduce

by

ralla akhil, Shadaab Siddiqie, P Krishna Reddy, Anirban Mondal

in

CODS-COMAD

Report No: IIIT/TR/2020/-1

Centre for Data Engineering
International Institute of Information Technology
Hyderabad - 500 032, INDIA
January 2020

Coverage Pattern Mining Based on MapReduce

Akhil Ralla
ralla.akhil@research.iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

Shadaab Siddiqie
mashadaab.siddiqie@research.iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

P. Krishna Reddy
pkreddy@iiit.ac.in
International Institute of Information Technology
Hyderabad, Telangana, India

Anirban Mondal
anirban.mondal@ashoka.edu.in
Ashoka University
Sonipat, Haryana, India

ABSTRACT

Pattern mining is an important task of data mining and involves the extraction of interesting associations from large databases. However, developing fast and efficient parallel algorithms for handling large volumes of data is a challenging task. The MapReduce framework enables the distributed processing of huge amounts of data in large-scale distributed environment with robust fault-tolerance. In this paper, we propose a parallel algorithm for extracting coverage patterns. The results of our performance evaluation with real-world and synthetic datasets demonstrate that it is indeed feasible to extract coverage patterns effectively under the MapReduce framework.

KEYWORDS

Data Mining, Knowledge Discovery, Coverage Patterns, MapReduce

ACM Reference Format:
Akhil Ralla, Shadaab Siddiqie, P. Krishna Reddy, and Anirban Mondal. 2020. Coverage Pattern Mining Based on MapReduce. In *CODS-COMAD 2020: ACM India Joint International Conference on Data Science Management of Data, Jan 05–07, 2020, Hyderabad, India.* ACM, Hyderabad, India, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Pattern mining [1, 15] is an important task of data mining and involves the extraction of interesting associations from large databases. It has significant applications in market basket analysis, recommendation systems, and internet advertising. In pattern mining based applications, databases are typically huge; this necessitates fast and scalable pattern mining algorithms. This problem can be addressed by the development of parallel algorithms in large-scale distributed environments. In the literature, the MapReduce framework [7] has been introduced for enabling the distributed processing of huge amounts of data on a large number of machines in geographically distributed environments with robust fault-tolerance. Computations in the MapReduce framework are distributed among worker machines and are described by the *map* and *reduce* functions. The *map* function processes key-value pairs and the *reduce* function merges all the values associated with the same key.

Another useful type of pattern is the coverage pattern [15], which has several important and diverse applications in areas such as banner advertising [17], search engine advertising [4, 5] and visibility mining [8]. Given a transactional database and a set of data items, coverage pattern (*CP*) is a set of items covering a certain percentage of transactions by minimizing overlap among the transactions covered by each item of the pattern. In the literature, a level-wise *CP* mining algorithm, designated as CMine [15], and a pattern growth approach called CPPG [16] have been proposed to extract *CPs* from transactional databases.

Incidentally, MapReduce-based pattern mining approaches have been proposed for extracting frequent patterns [11, 18, 19], periodic frequent patterns [3], utility patterns [12, 14, 23] and sequential patterns [6, 10, 21]. MapReduce-based pattern mining was first studied in the context of frequent patterns by means of an iteration-based apriori MapReduce algorithm [1, 20]. In this paper, we propose a new algorithm, designated as **CMineMR**, for the parallelization of the CMine coverage pattern mining algorithm under the MapReduce framework. The results of our performance evaluation with real-world and synthetic datasets demonstrate that it is indeed feasible to extract coverage patterns effectively by using our proposed MapReduce-based CMineMR algorithm.

The remainder of this paper is organized as follows. In Section 2, we discuss background information concerning coverage patterns. In Section 3, we present the proposed approach. In Section 4, we report the performance evaluation. Finally, we conclude in Section 5 with directions for future work.

2 BACKGROUND INFORMATION

This section discusses background information concerning coverage patterns.

2.1 Model of Coverage Patterns

Let $I = \{i_1, i_2 ....., i_n\}$ be the set of items, DB be the transactional database. Each transaction T in DB comprises a set of items i.e., $T \subseteq I$. $|DB|$ represents the total number of transactions in database DB. T^{i_p} represents the set of transactions, which contains the item i_p. $|T^{i_p}|$ represents the number of transactions containing i_p.

CODS-COMAD 2020, January 5-7, 2020, Hyderabad, India

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

The concept of coverage patterns incorporates the following notions: relative frequency, coverage set, coverage support and overlap ratio. We shall now discuss each of these notions.

Definition 1. Relative frequency of item i_p. *The fraction of transactions containing a item i_p is called the Relative Frequency (RF) of i_p and is computed as RF(i_p) = $\frac{|T^{i_p}|}{|DB|}$.*

An item is considered to be frequent if its $RF \geq minRF$, where *minRF* is a user-specified threshold.

Definition 2. Coverage set CSet(X) of a pattern X. *Given a pattern $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$, CSet(X) is the set of all transactions containing at least one item of pattern X, i.e., $CSet(X) = T^{i_p} \cup T^{i_q} \cup ...T^{i_r}$.*

Definition 3. Coverage support CS(X) of a pattern X. *Given $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$, CS(X) is the ratio of the size of $CSet(X)$ to $|DB|$ i.e., $CS(X) = \frac{|CSet(X)|}{|DB|}$*

Definition 4. Overlap ratio OR(X) of a pattern X. *Given $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ and $|T^{i_p}| \geq ... \geq |T^{i_q}| \geq |T^{i_r}|$, OR(X) is the ratio of the number of common transactions between $CSet(X - i_r)$ and T^{i_r} to the number of transactions having item i_r, i.e., $OR(X) = \frac{|CSet(X-i_r) \cap T^{i_r}|}{|T^{i_r}|}$*

A pattern is interesting if it has high *CS* since it covers more number of transactions. Suppose we want to increase the coverage by adding a new item i_k to the pattern X. The addition of item i_k will be more interesting if it adds more number of transactions for the coverage set $CSet(X)$ of pattern X . In essence, adding a new item i_k to pattern X could be interesting if there is a minimal overlap. Thus, a pattern having less *OR* could be more interesting.

Definition 5. Coverage pattern (*CP*). *A pattern $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ and $|T^{i_p}| \geq ... \geq |T^{i_q}| \geq |T^{i_r}|$ is called a coverage pattern if $OR(X) \leq maxOR$, $CS(X) \geq minCS$ and $RF(i_k) \geq minRF$ $\forall i_k \in X$, where maxOR and minCS are user-specified threshold values of maximum overlap ratio and minimum coverage support respectively.*

Given a set I of items, transactional database *DB*, *minRF*, *minCS* and *maxOR*, the problem of mining *CPs* is to discover the complete set of *CPs*.

About sorted closure property: The overlap ratio satisfies downward closure property if the items are ordered in descending order of their frequencies respective. Such a property is called the sorted closure property [13].

Sorted closure property. Let $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ be a pattern such that $|T^{i_p}| \geq ... \geq |T^{i_q}| \geq |T^{i_r}|$. If $OR(X) \leq maxOR$, all of the non-empty subsets of X containing i_r and having $size \geq 2$ will also have overlap ratio less than or equal to *maxOR*.

An item a is said to be a non-overlap item w.r.t. a pattern X if $OR(X, a) \leq maxOR$ and $RF(i_k) \geq minRF$ $\forall i_k \in \{X, a\}$. The notion of non-overlap pattern (*NOP*) is defined as follows.

Definition 6. Non-overlap pattern (NOP): *A pattern $X = \{i_p, ..., i_q, i_r\}$, $(1 \leq p, q, r \leq n)$ and $|T^{i_p}| \geq ... \geq |T^{i_q}| \geq |T^{i_r}|$ is called a non-overlap pattern if $OR(X) \leq maxOR$ and $RF(i_k) \geq minRF$ $\forall i_k \in X$.*

2.2 CMine Algorithm

Similar to the apriori algorithm [1], CMine [15] is an iterative multi-pass algorithm for extracting *CPs* from a given transactional database. In case of CMine, *NOPs* of size k are used to explore size k+1 patterns. As *NOPs* satisfy sorted closure property, we extract *NOPs*, which satisfy the *maxOR* constraint. Next, *CPs* are extracted by applying the *minCS* constraint.

Let C_k, NOP_k and CP_k denote the candidate, non-overlap and coverage patterns of size k respectively. At the k^{th} iteration, *NOPs* and *CPs* of size k are computed. Given *minCS*, *maxOR*, and *minRF* values, the steps of CMine algorithm for extracting *CPs* from the transactional database *DB* can be summarized as follows:

(1) **First iteration:** The frequency of each item is computed by scanning *DB*. After scanning, CP_1 and NOP_1 are computed by checking relative frequency. Item is added to NOP_1 if $RF \geq minRF$, and added to CP_1 if $RF \geq minCS$. The items in NOP_1 are sorted in descending order of their frequencies.

(2) **Second iteration and beyond:** Starting from k=2, the following step is repeated until C_k=ϕ. C_k is generated by computing $NOP_{k-1} \bowtie NOP_{k-1}$ (self-join). After scanning *DB*, NOP_k and CP_k are computed by checking *OR* and *CS* of candidate patterns in C_k accordingly.

3 PROPOSED APPROACH

This section presents the proposed approach.

3.1 Basic Idea

We distribute *DB* across N machines and extract the *CPs* in a distributed manner. Let $X = \{i_1, i_2, ..., i_n\}$ be a pattern, N be the number of machines and DB_i represent the i^{th} partition of *DB*. The basic idea is to extract the *CPs* by checking the values of *CS* and *OR* by accessing the partitions of *DB*.

The main issue is to compute the *OR* value of a candidate pattern in a distributed manner. Notably, as the value of $OR(X)$ is a fraction, i.e., $OR(X) = \frac{|CSet(X-i_n) \cap T^{i_n}|}{|T^{i_n}|}$, it cannot be computed by adding the *OR* values from the partitions of *DB*. However, $OR(X)$ can be computed efficiently by computing the numerator and the denominator of $OR(X)$ *independently* under the MapReduce framework. It can be observed that the denominator $|T^{i_n}|$ of *OR* is the frequency of a item. Hence, it is possible to compute the respective frequencies of all of the items in the first phase of MapReduce and store these frequencies in each of the N machines. Moreover, the value $CSet(X)$ can be computed by aggregating the corresponding coverage sets from the partitions of *DB* stored in each machine in a distributed manner. Once the frequency of the n^{th} item i_n is with every machine, the value of $OR(X) = \frac{|CSet(X-i_n) \cap T^{i_n}|}{|T^{i_n}|}$ can be computed in a distributed manner. Notably, the value of $CS(X)$ in *DB* can be computed in a distributed environment by adding the coverage support values in each partition of the *DB*.

The overview of the proposed approach under MapReduce is as follows. We distribute *DB* into N machines. In the first iteration, we compute relative frequency values of all items using one phase of MapReduce. We broadcast frequencies of all items to all machines. In the second iteration, we compute the *CPs* of size two by using one phase of MapReduce. From the third iteration onwards, we employ two phases of MapReduce; one phase is for generating candidate patterns, while the other phase is for computing *CPs*.

3.2 CMineMR Algorithm

Similar to the CMine algorithm, our proposed MapReduce algorithm is also an iterative algorithm. At the k^{th} iteration of the proposed CMineMR algorithm, *NOPs* and *CPs* of size *k* are computed. The input to the algorithm consists of a transactional database *DB*, *minRF*, *minCS* and *maxOR*. First, *DB* is segmented into multiple partitions and each partition is loaded onto each machine.

(i) **First iteration:** In this iteration we explain the generation of NOP_1 and CP_1. Each *mapper* reads each transaction of the data partition and maps each item to 1. *Reducer* groups all the item counts of each item into a list, which we designate as *count-list*. Then item frequencies are computed by adding counts in *count-list* of an each item. Algorithm 1 depicts the procedure to compute frequencies of size one itemsets. The NOP_1 and CP_1 are computed by comparing the relative frequencies with *minRF* and *minCS* respectively. The frequencies of NOP_1 are broadcast among all machines; this is used in the subsequent iterations.

Algorithm 1 First iteration-Computing CP_1, NOP_1 (DB)

```
procedure MAP(key = null,value = DB_i)
    for each t_i in DB_i do:
        for each i_k in t_i do:
            output < i_k, 1 >
procedure REDUCE(key = i_k, value = count-list(i_k))
    for each count in count-list(i_k) do:
        i_k.freq += count
```

(ii) **Second iteration:** In this iteration, we explain the generation of NOP_2, CP_2. In the second iteration, the candidate patterns are computed by joining the non-overlap patterns of the first iteration. The C_2 are broadcast across all machines. The *OR*, *CS* of C_2 are computed using one MapReduce phase.

For each transaction (t_i) and candidate pattern (P), *mapper* maps the *P* to [x,y] of the form: <P,[x,y]>. The first component x is 1 if t_i has at least one item of the *P*. The second component y is 1 if t_i has the least frequent item of *P* and at least one item among the remaining items. *Reducer* groups all the counts of each pattern into a list, which we designate as *counts-list*. Then the corresponding integers of each P are added. Algorithm 2 depicts the procedure to compute the size of coverage set and the numerator of overlap ratio of candidate patterns of size *k* $(k > 1)$. After reduction, the *CS* of *P* is computed by dividing the first component with the total number of transactions. The *OR* of *P* is computed by dividing the the second component with the frequency of the least frequent item (broadcast in the first iteration).

(iii) **Third iteration and beyond:** In this iteration, we explain the generation of NOP_k, CP_k $(k > 2)$. From the third iteration onwards, C_k are generated using one MapReduce phase. The *OR* and *CS* of C_k are computed using another MapReduce phase. This procedure of two MapReduce phases is repeated until no new candidate patterns are generated.

For each pattern *P* in NOP_{k-1}, *mapper* maps the pattern having the first *k*-2 items of *P* to the least frequent item of *P*. *Reducer* groups all the least frequent items based on the key into a list, which we designate as *item-list*. For each size *k*-2 pattern, C_k are generated

Algorithm 2 k^{th} iteration-Computing CP_k, NOP_k (DB, C_k)

```
procedure MAP(key = null,value = DB_i)
    for each t_i in DB_i do:
        for each P = {i_1, i_2, ..., i_k} in C_k do:
            if ∃i_m, m ∈ [1, k − 1] : i_m ∈ t_i and i_k ∈ t_i then
                output < P, [1, 1] >
            else if ∃i_m, m ∈ [1, k] : i_m ∈ t_i then
                output < P, [1, 0] >
procedure REDUCE(key = P, value = counts-list(P))
    for each count in count-list(P) do:
        P.count[0] += count[0]
        P.count[1] += count[1]
```

by iterating over the *item-list* as shown in Algorithm 3. Algorithm 3 depicts the procedure to compute C_k. The value of C_k is broadcast across all machines. The *CS* and *OR* of C_k are computed by another MapReduce operation, which is similar to the Second iteration.

Algorithm 3 $k^{th}(k > 2)$ iteration-Computing C_k (NOP_{k-1})

```
procedure MAP(key = null,value = NOP_{k-1})
    for each X = {i_1, i_2, ..i_{k-1}} in NOP_{k-1} do:
        output < {i_1, i_2, .., i_{k-2}}, i_{k-1} >
procedure REDUCE(key = X, value = item-list(X))
    for each i_m in item-list(X) do:
        for each i_n in item-list(X) do:
            if Freq(i_m) < Freq(i_n) then
                {i_1, i_2, .., i_{k-2}, i_m, i_n}
```

4 PERFORMANCE EVALUATION

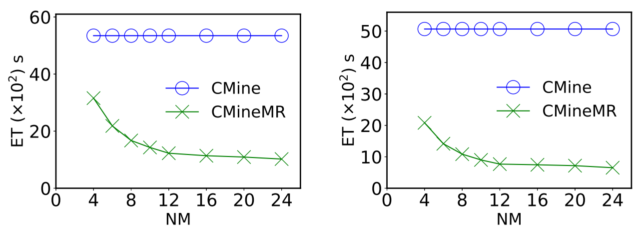
We have conducted experiments by implementing our proposed CMineMR algorithm as well as the reference CMine algorithm in Python 2.7. The CMineMR algorithm is written using Apache Spark architecture [22] and it is performed in a cluster of 24 machines, with 2 GB memory each. The experiments on the reference CMine algorithm [15] are performed in one machine of the cluster.

Table 1: Parameters used in our experiments

Dataset	Parameter	Default value	Variations	step-size
BMS-POS	N/of Machines (*NM*)	8	[4,6,8,10, 12,16,20,24]	-
	|DB|	515,596	-	-
	minRF	0.065	[0.065, 0.095]	0.01
	minCS	0.5	[0.1, 0.9]	0.1
	maxOR	0.6	[0.1, 0.9]	0.1
Synthetic	N/of Machines (*NM*)	8	[4,6,8,10, 12,16,20,24]	-
	|DB|	100,000	-	-
	minRF	0.045	[0.045, 0.06]	0.0025
	minCS	0.3	[0.1, 0.9]	0.1
	maxOR	0.3	[0.05, 0.5]	0.05

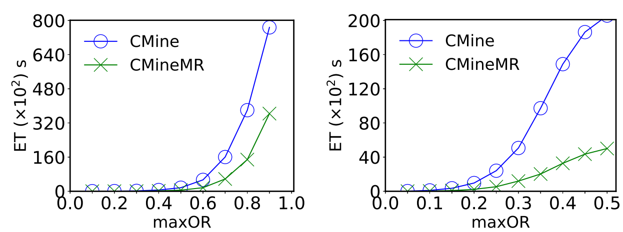
The experiments were conducted on two datasets. The first dataset is BMS-POS [9] dataset, which is a click-stream dataset of an e-commerce company; this dataset has 515,596 transactions and 1656 distinct items. The second dataset is the T10I4D100K, which is a synthetic dataset [2] generated by a dataset generator. This dataset has 100,000 transactions and 870 distinct items.

Our experiments are conducted by varying the number of machines (*NM*), *maxOR*, *minCS* and *minRF*. Table 1 summarizes the parameters used in our experiments. As the performance metric, we use execution time (*ET*), which is the total processing time (in seconds) for extracting *CPs* during the course of the experiment.



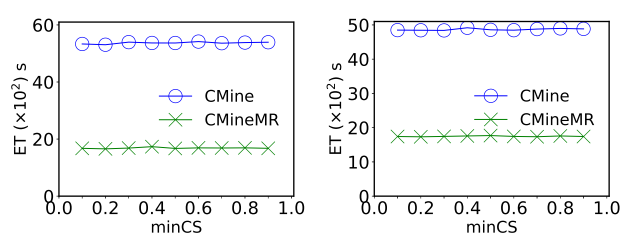
(a) BMS-POS (b) Synthetic

Figure 1: Effect of variations in NM



(a) BMS-POS (b) Synthetic

Figure 2: Effect of variations in maxOR



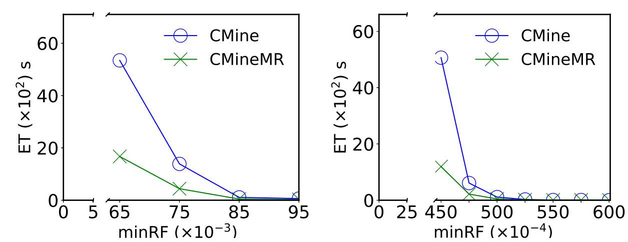
(a) BMS-POS (b) Synthetic

Figure 3: Effect of variations in minCS

(i) **Effect of variations in NM:** Figure 1 depicts the effect of variations in *NM*. For BMS-POS, the results are shown in Figure 1(a). The *ET* of CMineMR decreased rapidly till *NM*=8 due to a large amount of parallel computation in extracting *CPs*. However, the change in *ET* decreases with increase in *NM* and reaches saturation when 16 machines are used due to increase in the communication cost. The proposed CMineMR algorithm is 3.2 times faster than CMine algorithm when *NM* is 8. Similar trend is observed in Synthetic dataset as shown in Figure 1(b).

(ii) **Effect of variations in maxOR:** Figure 2 depicts the effect of variations in *maxOR*. For BMS-POS, the results are shown in Figure 2(a). The *ET* of CMine and CMineMR increases with the increase in *maxOR*, as the number of non-overlap patterns generated increases, thereby eventually increasing the runtime of the algorithms. The *ET* of CMineMR is 2.1 times faster than that of CMine algorithm when *maxOR* is 0.9 due to a significant amount of parallel computation in extracting *CPs*. The results for Synthetic dataset are shown in Figure 2(b).

(iii) **Effect of variations in minCS:** Figure 3 depicts the effect of variations in *minCS*. For BMS-POS, the results are shown in Figure 3(a). Notably, the *CPs* in each iteration are generated by checking *minCS* of *NOPs*, thereby leading to no significant changes in *ET* for CMine and CMineMR due to variations in *minCS*, as shown in Figure 3(a). The results for Synthetic dataset are depicted in Figure 3(b).



(a) BMS-POS (b) Synthetic

Figure 4: Effect of variations in minRF

(iv) **Effect of variations in minRF:** Figure 4 depicts the effect of variations in *minRF*. The results for BMS-POS, Synthetic datasets are depicted in Figures 4(a) and 4(b) respectively. The decrease in *ET* for CMine and CMineMR with increase in *minRF* represents the decrease in the number of size-one frequent itemsets (items satisfying *minRF*). For BMS-POS, the gradual decrease in *ET* indicates that there are small changes in the number of size-one frequent itemsets with increase in *minRF*. However, for Synthetic dataset, there is a sudden fall in *ET*, which indicates that most of the items are having comparable frequencies.

5 CONCLUSION

In pattern mining, developing fast and efficient parallel algorithms handling large volumes of data becomes a challenging task. In this paper, we have introduced the problem of parallel mining in the context of coverage patterns and proposed the CMineMR algorithm for efficiently extracting the knowledge of coverage patterns. The results of our performance evaluation with real-world and Synthetic dataset demonstrate that it is indeed feasible to extract coverage patterns effectively using our proposed CMineMR algorithm under the MapReduce framework. As part of future work, we plan to develop parallel algorithms for pattern growth approach towards extracting coverage patterns. Furthermore, we plan to investigate the parallel coverage pattern extraction by considering issues such as skew in transactional databases and load-balancing.

ACKNOWLEDGEMENTS

This work is supported by Kohli Center on Intelligent Systems, IIIT-Hyderabad. The authors are thankful to the anonymous referees for their useful comments.

REFERENCES

[1] Rakesh Agarwal, Ramakrishnan Srikant, et al. 1994. Fast algorithms for mining association rules. In *Proc. VLDB*. 487–499.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD*, Vol. 22. 207–216.

[3] Alampally Anirudh, R Uday Kiran, P Krishna Reddy, Masashi Toyoda, and Masaru Kitsuregawa. 2017. An efficient map-reduce framework to mine periodic frequent patterns. In *Proc. DaWaK*. 120–129.

[4] Amar Budhiraja, Akhil Ralla, and P Krishna Reddy. 2018. Coverage pattern based framework to improve search engine advertising. *International Journal of Data Science and Analytics* (2018), 1–13.

[5] Amar Budhiraja and P Krishna Reddy. 2017. An improved approach for long tail advertising in sponsored search. In *Proc. DASFAA*. 169–184.

[6] Chun-Chieh Chen, Chi-Yao Tseng, and Ming-Syan Chen. 2013. Highly scalable sequential pattern mining based on MapReduce model on the cloud. In *Proc. BigData Congress*. 310–317.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[8] Lakshmi Gangumalla, P Krishna Reddy, and Anirban Mondal. 2019. Multi-location visibility query processing using portion-based transactional modeling and pattern mining. *Data Min Knowl Disc* 33, 5 (2019), 1393–1416.

[9] Bart Goethals and Mohammed Javeed Zaki. 2004. Advances in frequent itemset mining implementations: Report on FIMI'03. *SIGKDD Explorations* 6, 1 (2004), 109–117.

[10] Jen-Wei Huang, Su-Chen Lin, and Ming-Syan Chen. 2010. DPSP: Distributed progressive sequential pattern mining on the cloud. In *Proc. PAKDD*. 27–34.

[11] Ming-Yen Lin, Pei-Yu Lee, and Sue-Chen Hsueh. 2012. Apriori-based frequent itemset mining algorithms on MapReduce. In *Proc. ICUIMC*. 76:1–76:8.

[12] Ying Chun Lin, Cheng-Wei Wu, and Vincent S Tseng. 2015. Mining high utility itemsets in big data. In *Proc. PAKDD*. 649–661.

[13] Bing Liu, Wynne Hsu, and Yiming Ma. 1999. Mining association rules with multiple minimum supports. In *Proc. ACM SIGKDD*. 337–341.

[14] Krishan Kumar Sethi, Dharavath Ramesh, and Damodar Reddy Edla. 2018. P-FHM+: Parallel high utility itemset mining algorithm for big data processing. *Procedia Comput. Sci* 132 (2018), 918–927.

[15] P Gowtham Srinivas, P Krishna Reddy, S Bhargav, R Uday Kiran, and D Satheesh Kumar. 2012. Discovering coverage patterns for banner advertisement placement. In *Proc. PAKDD*. 133–144.

[16] P Gowtham Srinivas, P Krishna Reddy, and AV Trinath. 2013. CPPG: Efficient mining of coverage patterns using projected pattern growth technique. In *Proc. PAKDD*. 319–329.

[17] AV Trinath, P Gowtham Srinivas, and P Krishna Reddy. 2014. Content specific coverage patterns for banner advertisement placement. In *Proc. DSAA*. 263–269.

[18] Chen-Shu Wang and Jui-Yen Chang. 2019. MISFP-Growth: Hadoop-Based Frequent Pattern Mining with Multiple Item Support. *Applied Sciences* 9, 10 (2019).

[19] Yaling Xun, Jifu Zhang, and Xiao Qin. 2015. Fidoop: Parallel mining of frequent itemsets using mapreduce. *IEEE Trans. Syst. Man Cybern. Syst.* 46, 3 (2015), 313–325.

[20] Xin Yue Yang, Zhen Liu, and Yan Fu. 2010. MapReduce as a programming model for association rules algorithm on Hadoop. In *Proc. ICIS*. 99–102.

[21] Xiao Yu, Qing Li, and Jin Liu. 2019. Scalable and parallel sequential pattern mining using spark. *WWW* 22, 1 (2019), 295–324.

[22] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[23] Morteza Zihayat, Zane Zhenhua Hut, Aijun An, and Yonggang Hut. 2016. Distributed and parallel high utility sequential pattern mining. In *Proc. IEEE Big Data*. 853–862.