PPS: Parallel Pincer Search for Mining Frequent Itemsets Based on Spark

Krishan Kumar Sethi, Ramesh Dharavath(✉), and Samuel Nyakotey

Department of Computer Science and Engineering,
Indian Institute of Technology (ISM), Dhanbad 826004, Jharkhand, India
kksethi02@gmail.com, ramesh.d.in@ieee.org, snyakotey@gmail.com

Abstract. Association rule mining is one of prominent techniques to discover the relation between data items of a transactional data. The process of mining has been simplified by considering only the frequent itemsets. Pincer search is one of the frequent itemset mining method which combines top-down and bottom-up search techniques to get the benefits of both. Top-down approach in Pincer search reduces the number of candidates in pass of iterations and saves a lot of computing resources. In this work, we present a Parallel Pincer Search (PPS) which is based on distributed implementation on Spark framework. We have converted the search algorithm according to the Spark framework to make it run in parallel. Spark provides a lot of features for the iterative algorithm such as in-memory execution, efficient data structure, better fault tolerant method, etc. We implemented the PPS on a Spark cluster with multiple datasets and analysed the performance.

Keywords: Pincer search · Frequent itemset mining · Apriori algorithm · Maximal itemset · Spark

1 Introduction

Knowledge discovery from the large data is a primary task of data mining techniques. Data mining [1] is a complex task which consumes a lot of computation. To simplify the knowledge discovery process, a few steps of pre-processions are applied which may change the structure or dimension of the data. All organizations keep the storage of their historical data to analyse and produce some good future strategies. For a last few decades, many techniques have been discovered to fetch a great evolution in the area of data mining. Data mining algorithms such as classification, clustering, association rule mining, prediction, abnormal findings, pattern discovery, etc. have achieved a great attention by the researchers and developers. Association rule mining is a process to determine the correlation between data objects. Many decision making applications are based on the association rule mining and further can be extended for the data analysis process. Association rule mining is a time consuming task due to the examination of all possible rules. Therefore, to reduce the search space it requires to consider only

A. Abraham et al. (eds.), *Proceedings of the Eighth International Conference on Soft Computing and Pattern Recognition (SoCPaR 2016)*, Advances in Intelligent Systems and Computing 614, DOI 10.1007/978-3-319-60618-7_35

the rules with frequent itemsets. A constrain called support measure is considered to discover the frequent itemsets. Support of an itemset is the count of that item in transactional data. A frequent itemset have the support count more than the user defined support threshold. Higher the support threshold can decrease the number of frequent itemset and vice versa. Therefore, always an adequate value of support threshold is chosen.

A large number of researches has been performed in the area of frequent itemset mining. Apriori algorithm [2] is one of the classical method which approach in bottom-up manner for searching for frequent items. This iterative algorithm discovers lower cardinality of frequent items in starting passes and the process goes with the higher cardinality of frequent items in further passes. There are many limitations of the apriori algorithm such as a huge number of candidate sets are generated and large number of passes are carried out if some frequent itemsets are large in size. Therefore, this kind of approach requires a lot of computing power and memory to process on large data. Pincer search [3] is another method which combines both bottom-up and top-down search methods to reduce the deficiencies of apriori algorithm. A top-down search starts with the discovery of frequent maximal itemsets and closure property is applied to deduce the other frequent itemsets. Pincer search discovers set of frequent items in each pass same as apriori algorithm, however, at the same time it computes and accumulates the set of frequent maximal itemsets. All the subsets of frequent maximal itemset are also frequent, hence many of candidates need not require to be processed. This approach optimizes the performance drastically if in starting passes, we get some large frequent maximal itemsets. To keep the track of maximal itemsets a data structure which is known as Maximal Frequent candidate set (MFCS) is introduced.

In the recent years, a big evolution in technology and science has increased the data size up to Exabyte and Zettabyte. This data is not only large in size but also unstructured and semi-structured in nature which causes intricacy in processing. Such data are termed as 'Big Data' [4] which has got great attention in the last few years. Big data requires a large set of resources for storage and processing. So, a traditional single machine computing is not enough to deal this problem. Hence, multi machine processing came into usage to accumulate and process such large amount of data. There are many multi machine distributed frameworks available for this purpose, such as MPI [5], Hadoop [6], and Dryad [7] etc. Hadoop is an open source framework for big data processing. Hadoop utilises the MapReduce library for programming model which provides the environment to write an application and run it across a cluster of machines in a distributed manner. Although, MapReduce is not suitable for iterative algorithms because of its performance decay due to network and I/O overhead. Spark is an alternative framework which has gained popularity in batch and interactive processing and also assures the high performance over MapReduce. Spark optimizes the process execution by introducing the job caching in main memory and a better approach for fault tolerance.

In this paper, we convey a Parallel Pincer Search (PPS) based on the Spark framework. Pincer search has performance degradation if frequent maximal itemsets are lengthy and scattered. Even for big data mining efficiency can't be achieved by Pincer search. Hence parallelize the search method is a better way to distribute the computing among multiple nodes to reduce the processing time. Apache Spark provides a lot of advantages over MapReduce and enhance the performance. Transactional data is loaded as data objects into main memory and implicitly distributed among the cluster nodes. We apply an iterative process which finds k- frequent itemsets by using the (k−1) - frequent itemsets in bottom-up manner (same as Apriori). In the same pass, we also find the set of all maximal frequent set using top-down search method. Spark framework in-memory execution and better fault tolerant approach provide us a fare computation time. We run PPS over a Spark cluster for various datasets and test the performance in terms of speedup and scale up.

The organization of this paper is as follows. Section 2 describes the background information to understand this paper, i.e., Spark framework, apriori, Pincer and important definitions. Related work is discussed in Sect. 3. Section 4 describes our proposed work in detail. Experiment details and result analysis are given in Sect. 5. Finally the manuscript is concluded in Sect. 6.

2 Background

This section includes some background knowledge and prerequisites to understand some definitions and concepts. In this, we provide the introduction of Spark along with its advantage and definitions related to frequent itemset mining algorithms i.e. Apriori and Pincer.

2.1 Spark

Spark [8] is an open source framework for big data processing and data analytics on distributed cluster computers. This framework was developed at the University of California, Berkeley's AMPLab and donated to the Apache foundation for maintenance and further advancement. Apache Spark provides many enhancements over Apache MapReduce of Hadoop framework [9], hence it is faster and efficient in terms of performance. The basic feature of Spark is in-memory processing which leads to the fast execution in iterative computation. In addition, intermediate data generated after each iteration is stored in primary memory which reduces the I/O cost. Spark has introduced an immutable data structure known as resilient distributed dataset (RDD) [10] for execution in primary memory. RDD objects are distributed on a cluster and processed in parallel in a fault tolerant manner. Spark has a better mechanism of fault tolerance where faulty RDD objects can be reconstructed by keeping the track of operations.

2.2 Prerequisites

(a) Frequent Set: We assume a transactional dataset D=$\{T_1,T_2,...,T_n\}$, where n is the number of transactions. Each transaction can be represented as a k set of items i.e. T= $\{i_1,i_2,...,i_k\}$. Let a user defined minimum support threshold s is criteria to define the frequent itemset. An itemset $X \subseteq T$ is called frequent if Support(X) $\geq \sigma$. Let n is 10 and σ=50%, an itemset X will be frequent if X appears at least 50% of transactions i.e. 5.

(b) Closure Properties: To find a frequent set from large transactional data creates a big problem as it needs to consider all possible itemsets which is a very lengthy process. Let suppose, k is the number of items then we require to check the support for each possible combination (i.e. 2k-1 itemsets) which is computationally very costly. The search space is reduced by adding an iterative procedure which does not process any infrequent itemset further. Following two closure properties are defined to reduce the candidate set in each pass of iteration.

Property 1. All the supersets of an infrequent itemsets are also infrequent.
Property 2. All the subsets of a frequent itemset are also frequent.

(c) Maximum Frequent Set: The frequent itemsets are categorized into the maximal frequent set, closed frequent set and frequent set. A frequent itemset $X \subseteq T$ is called maximal frequent set if there exist no frequent supersets. On the other hand, frequent itemset X is known as closed frequent set if there exists no frequent superset of same support count. Remaining frequent itemsets are ordinary frequent sets. A maximum frequent set (MFS) is the collection of all maximal frequent itemset. All the nonempty subsets of a frequent maximal itemset are also frequent by the property 2. Hence MFS is helpful to find the frequent itemsets very quickly. Each iteration uses the MFS to reduce the size of candidate set and number of passes.

2.3 Apriori Algorithm

Apriori algorithm is the basic algorithm to mine the frequent patterns in a set of transactions. It was developed by R. Agrawal and R. Srikant in 1994. The concept of the algorithm is to iteratively scan the transactions and find the frequent items. The frequent itemsets generated in one iteration are used to find larger itemsets in the next iteration. Although, searching of all possible subsets is a complex task. To reduce the search space a property called anti-monotone is used. This is also termed the apriori property as described below:

- All the nonempty subsets of a frequent set are also frequent.
- If a set of item is not frequent then its superset also can't be frequent.

An itemset is called frequent if it qualifies the minimum support threshold. Frequent itemset is represented by L_i for i^{th} -Itemset. There are two major operations in apriori algorithm.

Join Operation: A set of candidates are generated in each iteration by joining the frequent itemsets of last iteration. So L_k is generated by joining of L_{k-1} by itself, which is called candidate set denoted by C_k. At the time of join operation, apriori property is applied to reduce search space.

Pruning Operation: The candidate set C_k goes through a pruning phase to find the frequent itemsets L_k.

Each iteration applies these two operations on data to process in a bottom-up manner and produces frequent sets.

2.4 Pincer Search

Pincer search technique for frequent itemset mining is a bi-directional method, which includes the advantages of both bottom-up and top-down search. Each pass of iteration discovers frequent itemset in bottom-up manner like the apriori algorithm. Also, each pass finds the set of maximal frequent itemset in a top-down manner. If some of large MFS are searched out, then all the subsets of frequent maximal itemsets are pruned from the list of candidates. It saves a lot of computation and also decreases the number of passes if all larger itemsets are already discovered in earlier passes.

The two way search proceeds by using a new data structure called maximum frequent candidate set (MFCS). The MFCS accumulates the list of all maximal itemsets which can either be frequent or not. At the start of the search, all the frequent items are part of MFCS. Obviously, MFCS is a superset of MFS, which is updated after each pass. The Database is scanned to find the frequency check of each of maximal itemset in MFCS.

3 Related Work

A extensive research works have been carried out in the area of frequent itemset mining. Many algorithms are presented which are proven to work well for small scale of transactional data. Some of the popular algorithms such as apriori [2], EClat [11] and FP-growth [12] work in bottom-up fashion. There are some other algorithms which work in top-down fashion like maxclique [13], max-miner [14] and Pincer [15]. Pincer algorithm is a hybrid algorithm which combines both bottom-up and top-down approaches in each iteration.

When we introduce big data, all frequent itemset mining algorithms does not fit for processing such large data. All these algorithms are proposed before more than a decade, hence generated data used to be small in size and can be processed on a single machine. Nowadays, the single machine system is not enough to process such huge data. Hence, we need to switch towards multi machine computing system. Distributed computing using Hadoop and Spark frameworks are popular because of their parallel processing. Many research works adopted the Hadoop framework with MapReduce programming engine for frequent itemset mining on big data. Ye [16] introduced a distributed approach for processing big data using the apriori algorithm. This methodology comprises

a trie data structure to enhance the speed of frequent itemset mining. Lin [17] has proposed apriori implementation using Hadoop in three different algorithms, namely SPC, FPC and DPC. SPC is a single pass counting algorithm in which each map and reduce phase finds k-frequent itemsets while FPC combines a fixed number of passes (default is 3) in a map and reduce phase. On the other hand, DPC selects the number of passes to combine dynamically. A parallel version of apriori algorithm is presented by Li [18], which iteratively produces k-frequent itemsets for huge data. Although it has adopted only the basic functionality of apriori algorithm like SPC. Yu [19] suggested another MapReduce based apriori implementation which processes the data in a single phase and produce all possible frequent itemsets. Each cluster node requires a huge memory to store all possible candidates, otherwise, the job might fail. Another frequent itemset mining algorithm known as ECLAT has been also implemented in the distributed environment of Hadoop to process on big data [20]. There is more Hadoop based implementation of apriori algorithm has been given in [21,22] which are almost similar to the above work.

All the algorithms, given above, are implemented in MapReduce over Hadoop framework either in single stage or multistage. Although Hadoop is not suitable for iterative algorithms as it requires to save intermediate data to HDFS and read it back, which takes a high I/O cost. Spark is a cluster computing framework, which deals with an iterative algorithm in an efficient way and performs in-memory process to faster the execution. In recent years, many Spark based implementations of apriori algorithms are carried out. Implementation of basic Apriori algorithm in Spark is presented in YAFIM [23]. Yang [24] has introduced an innovative matrix based pruning stage in apriori which reduces the number of candidate sets generated. There are other Spark based implementation of apriori algorithms are presented in [25,26].

4 Proposed Work

We propose a Parallel Pincer Search which contributes the model and implementation of Pincer search in a distributed environment. Classical Pincer search is written to work in standalone mode. With the rapid growth of data, we require to process big data in parallel on multi machine setup. Hence, for frequent itemset mining from big transactional data, we cast the Pincer search to work in the distributed environment of Spark. PPS is divided into 2 passes, where pass 1 searches the 1-frequent itemset along with the MFCS and MFS. Pass N iteratively searches the frequent itemset of more than 1 cardinality. In addition, each iteration makes the efficient use of MFS of last iteration and update the MFCS. Both the passes are described as follows.

4.1 Pass 1

This phase is responsible for search of 1-frequent itemset in a bottom-up manner and also finds the MFS at the same time in top-down manner. Initially, data is

loaded in RDD and distributed on cluster nodes. We separate the items from the transaction ID and extract each item using flatMap() method. Each item is mapped in (key, value) form where a key is an item and value is the count of the item in transaction i.e. 1. Count of each item is added using the reduceByKey() which gives the another (key, value) pair, where key is an item and value is the count of that item in the entire dataset. A frequency check using the minimum support is applied to each item using the filter() method. All the frequent items are accumulated into MFCS which is the maximal candidate set. We further scan the entire data to count the support of MFCS. Itemset from MFCS is assigned to MFS, if it is frequent. The algorithmic instance of pass1 is shown in Algorithm 1.

Algorithm 1. Pass 1

Require: D: Transactional dataset
min_sup : Minimum support threshold
Ensure: $MFCS$: RDD of maximum frequent candidate set
MFS: RDD of maximum frequent set
$FrequentSet$: RDD of Frequent

1: **for each** transaction t in D **do**
2: **flatMap** ($trans\_ID$, t)
3: **for each** item I in t **do**
4: $itemCount$ = map(I, 1)
5: **end foreach**
6: **end foreach**
7: $sumCount$= $itemCount$.reduceByKey()
8: $frequentSet$=$sumCount$.filter(min_sup)
9: $MFCS$=$frequentSet$.keys()
10: Scan the transactions to determine support for MFCS
11: **for each** maximal item MIS in $MFCS$ **do**
12: **if** support(MIS) $\geq$ min_sup **then**
13: MFS=$MFS \cup MIS$
14: **end if**
15: **end foreach**
16: output ($MFCS$, MFS, $frequentSet$)

4.2 Pass N

Pass N of PPS is an iterative procedure which discovers the frequent itemsets more than 1 cardinality i.e. 2-frequent itemset, 3-frequent itemset and so on. Each iteration takes the input, i.e. MFCS and MFS from the last iteration and utilise MFS to reduce the candidate itemsets. All subsets of a maximal itemset in MFS are separated from the potential candidates list. Moreover, each iteration also finds the frequent set in the bottom-up manner. Pass N of PPS is described in the form of pseudo code in Algorithm 2. Initially, data is loaded into RDD and distributed among the nodes. Steps from 1 to 7 in algorithm are working to accumulate all distinct itemsets. All possible combinations of certain cardinality

(according to the value of the pass) of itemsets are created using the combinationGenerator() function. The set of all maximum frequent itemsets (MFS from the last iteration) are distributed among all the nodes using broadcast variable. We check, if MFS is not empty, then a list of itemsets are separated which are available as a subset of one of maximal itemset in shared_MFS.

Each candidate from the finalCandidates: RDD is processed to compute support count and mapped to (candidate, count) where count is the support for that candidate. Each candidate is pruned using the filter method. Itemsets which qualifies min_sup threshold are assigned into frequentIS:RDD and rest are assigned into nonfrequentIS:RDD. The current MFCS and nonfrequentIS is passed to MFCSGen() function which updates the MFCS. The MFCS_updated is pruned and assigned to MFS_updated which is further used in the next iteration.

Algorithm 2. Pass N

Require: $D\_frequent$: Transactional Dataset with frequent itemsets only
$MFCS$: RDD of maximum frequent candidate set
MFS: RDD of maximum frequent set
min_sup: Minimum Support Threshold
$pass$: Pass of iteration
Ensure: $frequentIS$: RDD of Frequent itemsets
$MFCS\_updated$: RDD of updated MFCS
$MFS\_updated$: RDD of updated

1: **for each** transaction t in $D\_frequent$ **do**
2: **flatMap**($trans\_ID$, t)
3: **for each** item I in t **do**
4: $itemList$= I.distinct
5: **end foreach**
6: **end flatMap**
7: **end foreach**
8: $candidates$=combinationGenerator($pass$, $itemList$)
9: $shared\_MFCS$=broadcast($MFCS$)
10: $shared\_MFS$= broadcast(MFS)
11: **if** (!($shared\_MFS$.value.isEmpty())) **then**
12: $finalCandidates$= $candidates$.filter($shared\_MFS$)
13: **else**
14: $finalCandidates$=$candidates$
15: **end if**
16: **for each** itemset IS in $finalCandidates$ **do**
17: $supportIS$= map($IS \Rightarrow$ (IS, supportCount(IS, $D\_frequent$)))
18: **end foreach**
19: $frequentIS$= $supportIS$.filter(min_sup).keys()
20: $nonfrequentIS$= $supportIS$.filter(min_sup).keys()
21: $MFCS\_updated$= MFCSGen($shared\_MFCS$, $nonfrequentIS$)
22: $MFS\_updated$=prune($MFCS\_updated$, min_sup)
23: output($MFCS\_updated$, $MFS\_updated$, $frequentIS$)

5 Experimental and Result Analysis

This section describes the implementation of PPS on a cluster of Spark and analyzes the performance for different datasets. Pincer search is already proved better than apriori algorithm, hence we do not compare PPS with distributed implementation of apriori. Instead of this, we consider different datasets and performance metrics, i.e. execution time, number of rounds and scalability. We prefer Spark framework in place of Hadoop because of its fast execution of iterative algorithms. As per our knowledge, PPS is the first implementation of Pincer search in a distributed environment. We have set up a Spark cluster and run PPS over it and measured the results.

5.1 Cluster Setup

We implemented the PPS in the distributed environment of Spark framework. A cluster of 4 nodes is set up which is hedgerows in nature. Three nodes of the cluster are configured on virtual machines on server which have Intel Xeon CPU E5-24070 clocked at 2.20 GHz and one core to each VM. One more node is configured on a workstation which consists Core(TM) i7-4510U CPU clocked at 2 GHz and 4 cores. We allot 7 GB of RAM to each node and connect them via 100 Mbps Ethernet switch. Each node is configured with Spark version 2.0.0 and Scala version 2.11.8. Program code of PPS is written in Scala programming language and tested the cluster for different data sets.

5.2 Datasets

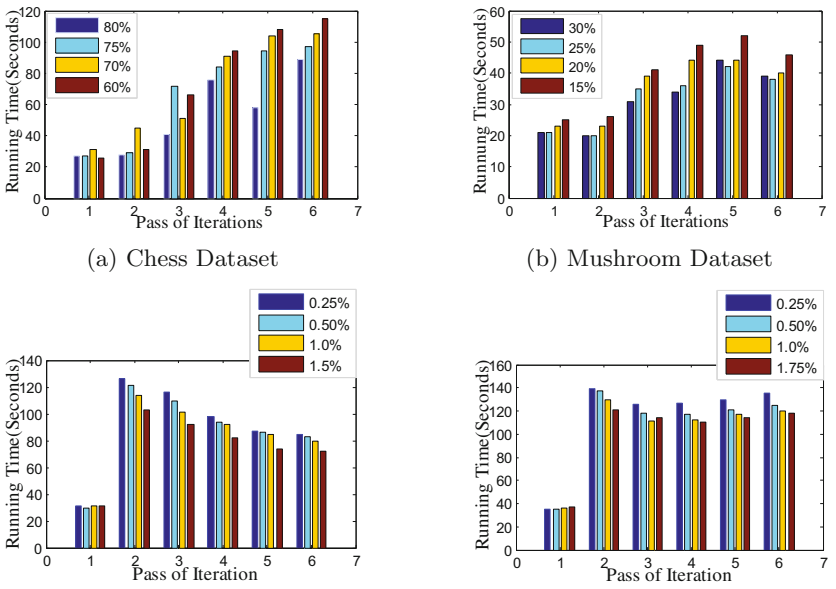
We run PPS for 4 different benchmark datasets and analyzed the results. First dataset is mushroom dataset [27] which comprises the various attributes of 23 species of grilled mushroom. Second dataset is a chess game dataset [27] which includes the end positions of the game for king vs king and rook. Third one is a synthetic dataset which is generated by data generator of IBM [28] and known as T1014D100k. Last dataset is a retail dataset [29] of a UK-based online store which enlist all the transactions occurred between 01-12-2010 and 09-12-2011. All four datasets are depicted in Table 1.

Table 1. Datasets with attributes

Dataset	Number of items	Number of transactions
Mushroom	119	8124
Chess	75	3196
T1014D100k	870	100000
Retail	16470	87988

5.3 Result Analysis

We run PPS for each dataset for different minimum support threshold and recorded the running time of each pass of iteration. We have chosen minimum support as 80%, 75%, 70% and 60% of total transactions for chess dataset. Execution time for initial passes are lower and keeps grow for later passes as depicted in Fig. 1a. The Number of candidate sets is not affected by MFCS and MFS in pass 1. Execution time for the pass 2 may be reduced if a larger maximum frequent itemsets are identified which significantly reduces the number of candidates to be processed. PPS is also run for mushroom dataset and recorded the execution time with respect to the threshold value of 30%, 25%, 20% and 15% as shown in Fig. 1b. Synthetic dataset T1014D100k has larger number of transactions which consumes highest running time in pass 2 and then goes lower in further passes as shown in Fig. 1c. Execution time of PPS for retail dataset with various support threshold 0.25%, 0.50%, 1.0% and 1.75% is illustrated in Fig. 1d It demonstrates that drops after 2nd pass is impact of large MFS. Running time goes a bit higher after 4th pass, due to no larger itemset remains in MFS.



(a) Chess Dataset

(b) Mushroom Dataset

(c) T1014D100k dataset

(d) Retail Dataset

Fig. 1. Performance of PPS with various datasets for different support threshold

We also check the performance of PPS for different scale of nodes. We run the PPS for mushroom dataset, synthetic data set T1014D100k and retail data set with 15%, 1.5% and 1.75% of the support threshold respectively on the cluster by varying the number of cores on each node. We found that by scaling the configuration of nodes the performance does increase significantly as depicted in Fig. 2.

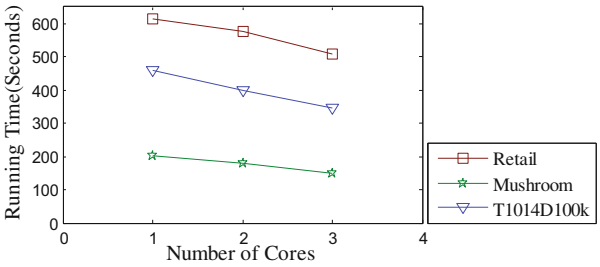


Fig. 2. Execution time by varying number of cores

6 Conclusion

This paper discusses about the frequent pattern mining for big transactional data. We choose one of the appropriate algorithm, i.e. Pincer search for searching frequent items in both top-down and bottom-up manner. We derive a Parallel Pincer Search (PPS) which is suitable to run parallel in the distributed environment of Spark. Original Pincer search is developed for single machine system and acquired great advantages over the apriori algorithm. In addition to apriori, Pincer also searches the frequent maximal itemsets which may prune a lot of candidate sets in each iteration and generate the frequent itemsets of larger cardinality. Two data structures MFCS and MFS contribute to process the data in bottom-up manner. Spark is one of the suitable framework over Hadoop, which runs iterative algorithms very rapidly. Spark inherent data structure, i.e. RDD and in-memory processing provide a better performance. We tested the performance of PPS with respect to multiple dataset on a Spark cluster. We analysed the speedup and scale up of PPS is accurate for different size of data and cluster conditions.

References

1. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. Elsevier, Amsterdam (2011)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceeding VLDB 1994, Proceedings of 20th International Conference on Very Large Data Bases, pp. 487–499 (1994)
3. Lin, D.-I., Kedem, Z.M.: Pincer-search: an efficient algorithm for discovering the maximum frequent set. IEEE Trans. Knowl. Data Eng. **14**(3), 553–566 (2002)
4. Chen, C.L.P., Zhang, C.Y.: Data-intensive applications, challenges, techniques and technologies: a survey on Big Data. Inf. Sci. (Ny) **275**, 314–347 (2014)
5. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufman, Burlington (1997)
6. Apache Hadoop. http://hadoop.apache.org
7. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS Oper. Syst. Rev. **41**(3), 59–72 (2007)

8. Karau, H., et al.: Learning Spark: Lightning-fast Big Data Analysis. O'Reilly Media Inc., Newton (2015)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM. **51**(1), 107–113 (2008)
10. Zaharia, M., Chowdhury, M., Das, T., Dave, A.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI12 Proceedings of 9th USENIX Conference Networked Systems Design and Implementation, p. 2 (2012)
11. Zaki, M.J., et al.: Parallel algorithms for discovery of association rules. Data Min. Knowl. Disc. **1**(4), 343–373 (1997)
12. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. ACM Sigmod Rec. **29**(2), 1–12 (2000). ACM
13. Zaki, M.J., et al.: New Algorithms for Fast Discovery of Association Rules. KDD **97**, 283–286 (1997)
14. Bayardo Jr., R.J.: Efficiently mining long patterns from databases. ACM Sigmod Rec. **27**(2), 85–93 (1998)
15. Lin, D.-I., Kedem, Z.M.: Pincer-search: a new algorithm for discovering the maximum frequent set. In: International Conference on Extending Database Technology, pp. 103–119. Springer, Berlin (1998)
16. Ye, Y., Chiang, C.-C.: A parallel apriori algorithm for frequent itemsets mining. In: Fourth International Conference on Software Engineering Research, Management and Applications (SERA 2006), pp. 87–94. IEEE (2006)
17. Lin, M.-Y., Lee, P.-Y., Hsueh, S.-C.: Apriori-based frequent itemset mining algorithms on MapReduce. In: Proceedings of 6th International Conference on Ubiquitous Information Management and Communication- ICUIMC 2012, p. 76. ACM (2012)
18. Li, N., Zeng, L., He, Q., Shi, Z.: Parallel implementation of Apriori algorithm based on MapReduce. In: 13th ACIS International Conference on Software Engineering Artificial Intelligence, Networking and Parallel/Distributed Computing, pp. 236–241 (2012)
19. Yu, R.-M., et al.: An efficient frequent patterns mining algorithm based on MapReduce framework. In: Software Intelligence Technologies and Applications and International Conference on Frontiers of Internet of Things, pp. 1–5 (2014)
20. Moens, S., Aksehirli, E., Goethals, B.: Frequent itemset mining for big data. In: IEEE International Conference on Big Data, pp. 111–118 (2013)
21. Lin, X.: MR-Apriori: association rules algorithm based on MapReduce. In: 2014 5th IEEE International Conference on Software Engineering and Service Science (ICSESS), pp. 141–144 (2014)
22. Yang, X.Y., Liu, Z., Fu, Y.: MapReduce as a programming model for association rules algorithm on Hadoop. In: 2010 3rd International Conference on Information Sciences and Interaction Sciences (ICIS), pp. 99–102. IEEE (2010)
23. Qiu, H., Gu, R., Yuan, C., Huang, Y.: YAFIM: a parallel frequent itemset mining algorithm with spark. In: Proceedings of International Parallel and Distributed Processing Symposium IPDPS, pp. 1664–1671 (2014)
24. Yang, S., Xu, G., Wang, Z., Zhou, F.: The parallel improved Apriori algorithm research based on spark. In: Proceedings of 2015 9th International Conference on Frontier of Computer Science and Technology FCST 2015, pp. 354–359 (2015)
25. Rathee, S., Kaul, M., Kashyap, A.: R-Apriori: an efficient apriori based algorithm on spark. In: Proceedings of the 8th Workshop on Ph.D. Workshop in Information and Knowledge Management, pp. 27–34. ACM (2015)

26. Gui, F., Ma, Y., Zhang, F., Liu, M., Li, F., Shen, W., Bai, H.: A distributed frequent itemset mining algorithm based on Spark. In: IEEE 19th International Conference on Computer Supported Cooperative Work in Design, vol. 18, pp. 271–275 (2015)
27. Asuncion, A., Newman, D.: UCI machine learning repository. http://archive.ics.uci.edu/ml/
28. Srikant, R.: Synthetic data generation code for association and sequential patterns. Available from the IBM Quest Web site http://www.almaden.ibm.com/cs/quest
29. Brijs, T.: Retail market basket data set. In: Workshop on Frequent Itemset Mining Implementations (FIMI03). http://fimi.ua.ac.be/data/retail.dat