Apriori based Frequent Itemset Mining

A.Meiappane[#1], P.Ganesh Guru Teja[*2], Meganathan.I[*3] Nilavazhagan.K[*4]

[#1]*Associate Professor, Manakula Vinayagar Institute of Technology, Puducherry, India.*

[1]*auromei@yahoo.com*

[*]*Manakula Vinayagar Institute of Technology Puducherry, India.*

[2]*guruteja@gmail.com*

[3]*megaarjun007@gmail.com*

[4]*nilasachin10@gmail.com*

***Abstract*— Tremendous amount of data is getting generated daily. The storage, retrieval, processing are becoming a great challenge. In order to process huge data and get preferred results there is no efficient algorithm and there exists many data mining algorithms which do consume long time to mine the huge datasets in order to get the patterns that are interesting to the user. So we propose an algorithm in this paper to implement the apriori based frequent itemset mining algorithm. we also has implemented this algorithm with normal apriori algorithm.**

***Keywords*— Apriori, Big Data, Application, Data mining, Performance, Association Rule Mining**

I. INTRODUCTION

The Apriori algorithm [2] is one of famous and well-known method for mining frequent itemsets. The algorithm works within a multiple pass generation, consisting of the joining and the pruning phases to reduce the number of candidates before scanning the database for support counting. Many proposed algorithms such as the FP growth algorithm [7] and so on have proposed to overcome the weakness of Apriori algorithm level wise generations and multiple pass databases scans. Scientists are also trying to parallelize these frequent itemset mining algorithms to speed up the mining of the ever-increasing sized databases [21].

Parallel mining first to divides the mining problem into smaller ones and then solve sub problems using homogeneous nodes such that each node may be work independently and simultaneously. Although parallelization may improve the mining performance, it also raises several issues including the partitioning of the input data, the balancing of the workloads, the formation of global information from local nodes, and the minimization of the communication costs. Deploying the mining methods into a grid computing environment by decomposing the task into smaller pieces and dispatching the sub-tasks to grid nodes may also improve the performance. The assumption of all grid nodes are fail safe and all tasks can be correctly completed might be too strong enough in real grid environments. In general, the potential errors due to failure nodes increase as the number of grid nodes increases. The failure of nodes cannot be ignored since erroneous or incomplete data are introduced. Even worse, the failure may cause an endless re-execution of all the jobs. Additionally, the scalability of the grid is limited because running on hundreds of nodes is prone to error.

To overcome the above problem, the MapReduce framework [5] has been introduced. MapReduce enables the distributed processing of huge data on large clusters, with good scalability and robust fault tolerance. A scale-up of hundreds or even thousands of nodes can be smoothly established. Algorithms running in the framework are described by two major functions, *map* and *reduce*. The input data are partitioned (and possibly replicated) and stored in a number of nodes. A master node initiates and schedules the two functions for executions in the nodes. The *map function* takes (from its node) the input data as a <key, value> pair and outputs a list of <key, value> pairs in a different domain. The *reduce function* takes the sorted output of the map function as <key, list-of-values> and outputs a collection of values. Both functions (i.e. a map task and a reduce task) can be performed in parallel. Thus, MapReduce can be an efficient platform for frequent itemset mining in huge datasets of terabyte or larger scale. Using the MapReduce framework to parallelize the mining task on a cluster of cheap servers might outperform some serial mining algorithms on a powerful server.

II. RELATED WORKS

Extensive algorithms on frequent itemset mining have been proposed for the past decades [2][7]. As the size of the database increases to terabyte or petabyte scale, even a powerful computing server may handle the mining well. Thus, parallel mining algorithms [3][4][14] are proposed. However, parallel mining comes out new problems to be solved, such as workload balancing, data distribution, jobs assignment, and parameters passing between node etc., many challenges need to be handled in parallel mining. To run a mining task on a cluster, the database should be split into many sub-databases and be distributed to nodes. And the job for each node should also be decided. If the workload for each node is not balance, the total execution time will be delayed by stragglers. With a missing setting for parallel mining, many nodes may be idled and total execution time may be increased.

With the rise of cloud computing, MapReduce [5] becomes one of most important technique for cloud computing, it hiding the problems like data distribution, fault tolerance, and workload balance and users just focus on algorithms. MapReduce has been widely researched in many areas, such as database [1][10], text similarity search [12][19], and data mining [8][11]. Moreover, the performance of MapReduce is also discussed and many enhancing techniques have been proposed [6][9][13]. In this paper, we also proposed three mining algorithms to analyse the impact on performance of different implementations, and summarize a technique for enhancing the performance of mining algorithms using MapReduce.

III. PREPARATION

1. The Apriori Algorithm

Three algorithms are proposed to investigate the Apriori-like algorithms in the MapReduce paradigm. The Apriori algorithm mines all the frequent itemsets in a transactional database, where each transaction *ti* contains a set of items called *itemset*. An itemset having *k* items is called a *k*-itemset and its *length* is *k*. An itemset *X* is frequent if its support, which is the fraction of transactions containing *X* in the database, is at least certain user- specified minimum support *min_sup*. Let L_k denote the frequent itemsets of length *k* and C_k denote the candidate itemsets of length *k*. The Apriori algorithm joins L_{k-1} to generate C_k, counts the supports of C_k, and determine the L_k in *k-th* database scanning. The algorithm terminates when no C_k or L_k is generated. Note that usually the discovery of frequent 1-itemsets is accomplished by a simple counting of items in the first pass of database scanning (pass-1). Starting from pass-2, the hash-trees are used for arranging C_k to facilitate fast support counting. The pruning of candidates using the downward closure property is effective for candidates of length larger than two, starting from pass-3.

2. The MapReduce Paradigm

MapReduce [5] is proposed to support distributed computing, i.e. a share nothing architecture, on huge data and Hadoop [23] is one of the implementations of the MapReduce frameworks. The features of MapReduce include the following. First, MapReduce partitions data into equal sized blocks with some replicas and distributes blocks evenly to the distributed file systems (such as HDFS [16]) automatically so that users are free from the locations and the distributions of data. Second, MapReduce re-executes a crashed task without re-executions of the other tasks and achieves good fault-tolerances. Third, MapReduce increases total throughputs by re-assigning un-finished tasks of slower or busy nodes to idle nodes (which have accomplished their tasks) in a heterogeneous cluster.

In a MapReduce cluster, one node is designated as the *master*, who schedules tasks for execution among nodes, and other nodes are the *workers*. The master and the workers can be located at the same node. Computations in the MapReduce framework are distributed among nodes and are described by the map function and the reduce function. Programmers address the required computations to be executed in parallel by designing map tasks and reduce tasks. The map tasks (also the reduce tasks) will be executed concurrently by the configured workers. The input data file is evenly divided into disjoint chunks and stored in the nodes in the distributed file system like GFS or HDFS. Usually, workers having input chunks are assigned with map tasks to reduce the required data transmissions for input. Map tasks and reduce tasks can be assigned to the same worker.

The master schedules the map tasks and the reduce tasks to the configured cluster nodes when a job is initialized. Each map task then reads one block from the file system, and outputs the <key, value> pairs specified by the map function. Each reduce task receives the sorted <key, value-list> pairs of the associated keys. All the pairs of the same key are sent to one assigned worker responsible for that key. The reduce task performs the operations specified in the reduce function and finally outputs the results to a file. Additionally, the *combine* function is specified, usually at the end of the map task, to collect local <key, value> pairs at a map worker to reduce communications between map tasks and reduce tasks. The combine function merges the <key, value> pairs of the same key in the map worker into one <key, value> pair and finally outputs the pair.

Algorithms in the MapReduce framework may comprise several map-reduce phases. The master can collect the results from a map- reduce phase in a file, then schedule workers for the next map- reduce phase. The *DistributedCache* is used when the entire content of a file is to be read by all the map workers or reduce workers. Moreover, outputs of both a map and a reduce task are written to files so that the failure of a map-reduce phase can be recovered from such checkpoints without having to re-execute the job from the beginning when an error occurs.

IV. PROPOSED ALGORITHMS

The fundamentals of parallelizing the Apriori algorithm in the MapReduce framework is to design the map and the reduce functions for candidate generations and support counting. The first proposed algorithm, Single Pass Counting, finds out frequent *k*-itemsets at *k*-th pass of database scanning in a map-reduce phase. The second proposed algorithm, Fixed Passes Combined-counting, finds out frequent *k*-, (*k+1*)-, …, and (*k+m*)-itemsets in a map-reduce phase. In this paper, FPC discovers frequent *k*-, (*k+1*)-, and (*k+2*)-itemsets. The third proposed algorithm, Dynamic Passes Combined-counting , considers the workloads of nodes and finds out as many frequent itemsets of various lengths as possible in a map-reduce phase. For convenience, a map task is called a *mapper*, and a reduce task is called a *reducer* in the following context.

In general, the number of mappers is larger than the number of reducers in MapReduce. With the size of cluster increasing, the more mappers can be used for processing data, and the problem can be divided into smaller granularity. In all of our algorithms, each mapper calculates counts of each candidate

from its own partition, and then each candidate and corresponding count are output. After map phase, candidates and its counts are collected and summed in reduce phase to obtain partial frequent itemsets. By using count distribution between map phase and reduce phase, the communication cost can be decreased as much as possible. Since frequent 1-itemsets are found in pass-1 by simple counting of items. Phase-1 of all the three algorithms is the same, as shown in Figure 1. The mapper outputs <item, 1> pairs for each item contained in the transaction. The reducer collects all the support counts of an item and outputs the <*item*, *count*> pairs as a frequent 1-itemset to the file *L1* when the *count* is no less than the minimum support count. At first, different map-reduce functions are designed to characterize the features of the three algorithms, starting from pass-2. The huge number of *C2* in common executions, however, may overload nodes of map functions if candidates of length larger than two are combined for support counting. Thus, the same phase-2 is applied to all the three algorithms. Figure 2 shows phase-2 of the proposed algorithms. The apriori-gen () function, subset() function, and the hash-trees in Figure 2 are the same as those in Apriori [2]. In fact, the reduce function in phase- 2 is the same for all subsequent phases in all the three algorithms.

Map(key, value = itemset in transaction t_i) :
Input: database a database partition D_i
1. foreach transaction $t_i \in D_i$ do
2. foreach item $i \in t_i$ do
3. output <*i*, 1> ;
4. end
5. end

Reduce (key=item, value=count) :
1. foreach key *y* do /* Initial *y*.count = 0 */
2. foreach value *v* in *y*'s value list do
3. *y*.count += *v* ;
4. end
5. if y.count ≥ minimum support count
6. output <*y*, *y*.count> ; /* collected in L_1 */
7. end
8. end

In phase-*k* (*k* t 2) of SPC, each mapper first reads *Lk-1* from the *DistributedCache* to generate *Ck* in a hash tree. It then performs the support counting and outputs the <*itemset*, *count*> pairs as a frequent *k*-itemset to the file *Lk* if the *count* passes the minimum threshold. Note that the combine function in Hadoop MapReduce framework is invoked for all the mappers in all the proposed algorithms to reduce the communications between mappers and reducers. Recall that the input data is evenly divided into disjoint chunks in Hadoop so that each mapper can reads only part of the whole database.

We use an example to illustrate the executions of SPC as follows. A database of six transactions, *min_sup* = 33%, three mappers, and two reducers are shown in Figure 4. In phase-1, the mapper handles transaction *t1* by outputting <A, 1>, <B, 1>, and <C, 1> pairs, and handles transaction *t2* by outputting <A, 1> and <C, 1> pairs. The *combine* function then is invoked to sum up the counts and outputs frequent 2-itemsets into *L2*. Similarly, phase-3 of SPC is invoked, <C D E, 2> is outputted, and SPC terminates

10B

Being a level-wised algorithm, SPC iterates *k* times of the map- reduce phase when the maximum length of the frequent itemsets is *k*. The number of candidates often is small so that the utilization of the workers is low in the last few phases. The scheduling of mappers and reducers becomes an overhead in comparison to the workload of a worker for these phases. Moreover, the number of database scans required is the number of map-reduce phases performed in SPC. Thus, the cost of database loading in the beginning of each phase is relatively high if only few candidates are counted in a phase. Therefore, FPC combines candidates from several phases in SPC and performs the support counting in a single map-reduce phase. Counts, and outputs <A, 2>, <B, 1>, and <C, 2> finally. The reducer sums up the counts associated with each key (i.e. item-id) and outputs items having counts at least 2 into *L1*.

In phase-2, as shown in Figure 5, each mapper reads *L1* and generates *C2* in a hash tree. The mapper *Map-3*, for example, reads *t5* but outputs nothing, then reads *t6* and outputs <C D,1>, <C E, 1>, and <D E, 1> for the reducers. The reducer sums up the

The FPC algorithm is shown in Figure 7 and the map-reduce phase in the FPC algorithm is shown in Figure 6. Phase-1 and phase-2 of FPC are the same as that of SPC. Starting from phase-3, FPC combines candidates from a fixed number of database passes for support counting in a map-reduce phase. In this paper, the fixed number is three so that candidates of every three consecutive lengths are counted in one phase. Thus, FPC counts the supports of *C3*, *C4*, and *C5* in phase-3, that of *C6*, *C7*, and *C8* in phase-4, and so on. As shown in Figure 6, the mapper reads *Lk-1* from *DistributedCache* to generate *Ck*. *Ck+1* and *Ck+2* are generated using *Ck* and *Ck+1*, respectively. These candidates are then placed in a prefix-tree for support counting. The reduce function in fact is the same as that in SPC. Therefore, in comparison to SPC, FPC reduces the number of map-reduce phases required and has better utilizations of the workers in the last few database passes. The number of database scans consequently is reduced.

Nevertheless, the number of candidates from combined passes may overload the workers, especially when the number of passes to be combined is fixed. The number of candidates generated at earlier passes, such as *C3*, can be too large to be combined with candidates of longer length. The reduction in database passes turns out to increase the number of false-positive candidates that overloads the mapper. The resulting performance of FPC would be worse than that of SPC. An example is shown in the experimental result in Figure 11(b). The weakness of FPC thus is the inability to prune candidates due to fixed combined counting without flexibility. Therefore, algorithm DPC is proposed to dynamically determine the candidates to be merged in a map- reduce phase.

IV. PERFORMANCE EVALUATION

Extensive experiments were conducted to assess the performance of the proposed algorithms. All the experiments were performed in a Hadoop 0.21.0 cluster of four nodes, where each node contains an Intel Pentium Dual Core E6500 2.93GHz CPU, 4GB RAM, and a 500GB hard disk running Ubuntu 15.04. All of the experiments were configured with 7 map tasks and 1 reduce task. All of the three algorithms are implemented in Java and the JDK version is 1.6.0_23.

Both real and synthetic datasets were used in the experiments. Real datasets BMS-POS and BMS-WebView-1 from FIMI were used [22]. The synthetic datasets were generated by the IBM dataset generator. The number of distinct items (|N|) is 10,000 and the average length of transactions (|T|) is 10. The results of experiments using different settings of |T|, |N|, and |I| are consistent.

V. CONCLUSION

We have proposed the algorithm named Apriori algorithm based on frequent pattern mining using the mapreduce programming model, investigate the performance of the Apriori-like algorithms in a MapReduce framework in this paper. It is a simple conversion of the serial Apriori algorithm into the distributed MapReduce version. SPC finds the frequent *k*-itemsets in *k*-th database scan (map-reduce phase), using mappers to generate candidates' supports and reducers to collect global supports. FPC improves SPC by using a mapper to count the candidate *k*-, (*k+1*)-, and (*k+2*)-itemsets altogether in a map-reduce phase. Consequently, FPC effectively reduces the number of map-reduce phases. Nevertheless, the performance of FPC might be worse when too many false-positive candidates are collected for counting by mappers. DPC is proposed to strike a balance between reducing the number of map-reduce phases (by combining variable-length candidates) and increasing the number of pruned candidates. DPC dynamically collects candidates of variable lengths for counting by mappers according to the number of candidates and the execution time of previous map-reduce phases.

REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D.J. Abadi, A. Rasin, and A. Silberschatz: HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In: Proceedings of the VLDB Endowment (PVLDB), 2(1): 922-933, 2009

[2] R. Agrawal and R. Srikant: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of the Twentieth International Conference on Very Large Databases (VLDB), pp. 487-499, 1994

[3] R. Agrawal and J.C. Shafer: Parallel Mining of Association Rules, In: IEEE Transactions on Knowledge and Data Engineering (TKDE), 8(6): 962-969, 1996

[4] S. Cong, J. Han, J. Hoeflinger, and D. Padua: A Sampling-based Framework for Parallel Data Mining. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 255-265, 2005

[5] J. Dean and S. Ghemawat: Mapreduce: 6LPSOLILHG ▼DWD Processing on Large Clusters. In: Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI), pp. 137-150, 2004

[6] J. Dean and S. Ghemawat: MapReduce: A Flexible Data Processing Tool. In: Communications of the ACM (CACM), 53(1):72-77, 2010

[7] J. Han, J. Pei, and Y. Yin: Mining Frequent Patterns without Candidate Generation. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 29(2):1-12, 2000

[8] J.-W. Huang, S.-C. Lin, and M.-S. Chen: DPSP: Distributed Progressive Sequential Pattern Mining on the Cloud. In: Proceedings of the 14th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), pp. 27- 34, 2010

[9] D. Jiang, B.C. Ooi, L. Shi, and S. Wu: The Performance of MapReduce: An In-depth Study. In: Proceedings of the VLDB Endowment (PVLDB), 3(1): 472-483, 2010

[10] J.L. Johnson: SQL in the Clouds. In: Computing in Science and Engineering, 11(4):12-28, 2009

[11] H. Li, Y. Wang, D. Zhang, M. Zhang, and E.Y. Chang: PFP: Parallel FP-Growth for Query Recommendation. In Proceedings of the 2008 ACM Conference on Recommender Systems, pp. 107-114, 2008

[12] R. Li, L. Ju, Z. Peng, Z. Yu, and C. Wang: Batch Text Similarity Search with MapReduce. In: Proceedings of the 13th Asia-Pacific Web Conference on Web Technologies and Applications, pp. 412-423, 2011

[13] R. McCreadie, C. Macdonald, and I. Ounis: MapReduce Indexing Strategies: Studying Scalability and Efficiency. In: Information Processing and Management, pp. 1-16, 2011

[14] A. Mueller: Fast Sequential and Parallel Algorithms for Association Rule Mining: A Comparison. In: Tech. Report CS-TR-3515, University of Maryland, College Park, Md., 1995

[15] A. Pavlo, E. Paulson, A. Rasin, D.J. Abadi, D.J. DeWitt, S. Madden, and M. Stonebraker: A Comparison of Approaches to Large Scale Data Analysis. In: Proceedings of SIGMOD, pp. 165-178, 2009

[16] T. Shintani and M. Kitsuregawa: Hash Based Parallel Algorithms for Mining Association Rules. In: Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, pp. 19-31, 1996

[17] K. Shvachko, H. Kuang, S. Radia, and R. Chansler: The Hadoop Distributed File System. In: Proceedings of the Mass Storage Systems and Technologies (MSST), pp. 1-10, 2010

[18] M. Stonebraker, D.J. Abadi, D.J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin: MapReduce and Parallel DBMSs: Friends or Foes? In: Communications of the ACM (CACM), 53(1):64-71, 2010

[19] R. Vernica, M. J. Carey, C. Li: Efficient Parallel Set-Similarity Joins Using MapReduce. In: Proceedings of SIGMOD, pp. 495-506, 2010

[20] K.-M. Yu, J. Zhou, T.-P. Hong, and J.-L. Zhou: A Load-Balanced Distributed Parallel Mining Algorithm, Expert Systems with Applications, 37(3):2459-2464, 2009

[21] M. J. Zaki: Parallel and Distributed Association Mining: A Survey. In: IEEE Concurrency, 7(4):14-25, 1999

[22] Z. Zheng, R. Kohavi, and L. Mason: Real world performance of association rule algorithms. In: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 2001, pp. 401-406, 2001.

