International Journal of Database Theory and Application
Vol.9, No.6 (2016), pp.81-92
http://dx.doi.org/10.14257/ijdta.2016.9.6.08

PFIN: A Parallel Frequent Itemset Mining Algorithm Using Nodesets

Chen Lin and Junzhong Gu

Department of Computer Science and Technology
East China Normal University
m2linchen@gmail.com, jzgu@ica.stc.sh.cn

Abstract

Frequent Itemset Mining (FIM) is one of most fundamental techniques in data mining with extensive applications to a variety of data mining problems such as association rule mining, correlations, clustering and classification. Since the first proposal of frequent itemset mining, numerous serial algorithms have been proposed in order to improve mining performance, yet most of them cannot scale to massive datasets which are very common nowadays. In this paper, we propose a new parallel FIM algorithm named PFIN based on Nodeset which is a more efficient data structure for mining frequent itemsets. PFIN can intelligently decompose a large-scale FIM problem into a set of tasks, where each task can be executed in parallel without unnecessary communication overheads. Moreover, a hash-based load balancing strategy has been adopted to optimize resource use and maximize throughput. For evaluating the performance of PFIN, we have conduct extensive experiments on Spark which is an emerging distributed in-memory processing framework to compare it against PFP which is one of state-of-the-art parallel FIM algorithms on a range of real datasets. The experimental results demonstrate that our proposed PFIN are highly competitive with PFP in scalability performance, outperforming PFP in speed performance.

Keywords*: data mining, frequent itemset mining, distributed computing, spark*

1. Introduction

As an essential data mining task, Frequent Itemset Mining (FIM) aims to discover frequently co-occur items in a dataset. Since the problem of frequent itemset mining was firstly proposed by Agrawal *et al.* [1], extensive efforts have been devoted to improving the performance of FIM algorithms. However, the extrinsic characteristic of the exponential solution search space makes FIM remain a very time-consuming and challenging process especially when dealing with big data. Many serial algorithms [2-4] work well on small datasets, whereas most of them are not able to finish the mining task upon large-scale datasets within an acceptable time due to the limit computation capability and memory capacity of a single node.

On the other hand, parallel computing which were developed to process massive datasets offers a potential solution to this problem [5]. Early efforts focused on parallelizing the Apriori-like algorithms [6-7]. However, parallel Apriori-like algorithms inevitably suffer from repetitive I/O scans and shuffling around intermediate results of the mining process across the distributed nodes for a share-nothing environment. In recent years, parallel FP-growth algorithms [8-9] which adopt a highly compact representation of datasets named FP-tree and employ a divide-and-conquer philosophy to mine frequent itemsets without candidate generation has been proven to achieve impressive efficiency. However, the parallel FP-growth methods become inefficient when datasets are sparse due to the significant cost in constructing and traversing the FP-trees.

In addition, recent studies have shown that MapReduce framework [10] is not suitable for sophisticated data mining algorithms with intensive iterative computations since it needs to write the intermediate results into a specified location for the next iteration to read which will involves heavy I/Os and extra job start-up time.

FIN [11], firstly proposed by Zhihong Deng, is an efficient frequent itemset mining algorithm based on a novel data structure called Nodeset. Compared with FP-growth, the FIN method do not need to recursively build numerous conditional FP-trees and can directly discover frequent itemsets in a set-enumeration tree [12]. Previous studies have shown that FIN outperforms many current single-node algorithms. Therefore, it is reasonable to parallelize FIN to achieve a better performance for frequent itemset mining. To the best of our knowledge, there are no studies on this issue so far.

In this paper, we propose PFIN, a parallel FIN algorithm based on Nodesets to efficiently represent itemsets. PFIN partitions the transaction database with a hash-based load balancing strategy and divides the set-enumeration tree in such a way that each partition can be executed to discover the frequent itemsets in parallel. Such partitioning significantly reduces the unnecessary communication overheads and greatly improves the scalability and performance. Apache Spark [13] is an emerging distributed in-memory processing framework which has been proved that it is much more suitable for data mining algorithms. We have implemented PFIN on Spark and conducted extensive experiments to compare it against PFP [9] which is one of state-of-the-art parallel FIM algorithms on a range of real datasets for evaluating the performance of PFIN. The experimental results show that our proposed PFIN runs much faster than PFP and achieves near-linear scalability.

The rest of this paper is organized as follows. Section 2 briefly introduces the concept of frequent itemset mining and Spark. Section 3 discusses related work on frequent itemset mining. Section 4 introduces the PFIN algorithm in detail. Section 5 evaluates the performance of PFIN. Section 6 summarizes the paper.

2. Background

2.1. Frequent Itemset Mining

Let $I = \{i_1, i_2, i_3, \dots, i_n\}$ be a set of n distinct items in the transaction database D where each transaction T in D contains a set of items from I called itemset. If an itemset contains k items, then it is called a k-itemset. The support of an itemset S is defined as the number of transactions that contain S. And we denote |S| as the number of items in S. Formally, $\mathrm{sup}(S) = |\{tid | S \subseteq T, (tid, S) \subseteq D\}|$, where tid is a transaction identifier. An itemset is said to be frequent if it has a support greater than a given threshold σ which is called minimum support or minSup in short. Given a database D, FIM algorithms aim to discover frequent itemsets meeting the user-specified minimum support.

2.2. Apache Spark Framework

Apache Spark [13], as an emerging distributed in-memory processing framework originally developed in the AMPLab at UC Berkeley, was purposely designed to perform real-time data analysis at lightning fast speed. Compared with MapReduce's two-stage paradigm, Spark running in-memory on the cluster makes it superior in iterative computations and low-latency workloads. Intermediate results will be cached by Spark in memory in case of future reuse. Whereas, MapReduce has to persist intermediate results to local disk in each iteration, requiring a great deal of extra I/Os and unnecessary computations. In consequence, many iterative data mining algorithms implemented in MapReduce run significantly slower than they do on distributed in-memory processing frameworks. Besides, Spark provides more versatile APIs for Scala, Java and Python than MapReduce which is infamous for being difficult to program.

3. Related Work

Since massive real datasets may not fit in a single workstation's memory, how to parallelize serial FIM algorithms to address frequent itemset mining becomes extremely critical. The choice of memory model determines the way of accessing and storing data, which in turn have a significant impact on the performance of a parallel algorithm. Shared memory systems offers a global memory space which can be accessed by all processes. A major advantage of shared memory system is that memory access becomes much cheaper than inter-node communication. However, to our problem of dealing with big data, scalability of a shared memory system is limited by available memory. In shared-nothing systems or distributed systems, each node of the cluster has its private memory address space. Great speedup can be easily obtained for the shared-nothing systems by scaling out to hundreds or even thousands of machines.

Most of the previously proposed parallel FIM algorithms can be classified into two categories: the parallel Apriori-like aglortihms and the parallel FP-growth algorithms. The parallel Apriori-like algorithms employ the downward closure property which is a sharp knife for pruning: the subsets of a frequent itemset must be frequent. Agrawal and Shafer [6] proposed a Count Distribution (CD) algorithm to distribute the transaction database among processes on a share-nothing system. Each process computes the support of all candidate itemsets with respect to its local partition. Based on the CD algorithm, Lin *et al*. [14] introduced three Apriori-like algorithms on MapReduce framework: Single Pass Counting (SPC), Fixed Passes Combined-counting (FPC), and Dynamic Passes Combined-counting (DPC). Qiu *et al*. [15] proposed a new implementation of a parallel Apriori-like algorithm named YAFIM on Spark. However, these parallel Apriori-like algorithms inevitably suffer from repetitive I/O scans and generation of numerous candidate itemsets.

Many studies have proved parallel FP-growth algorithms run much faster than parallel Apriori-like algorithms. Parallel FP-growth algorithms employ a compact data structure named FP-tree to compress the transaction database. Frequent itemsets can be discovered by recursively constructing and traversing conditional FP-trees. Pramudiono *et al*. [8] parallelized the FP-growth algorithm on a shared-nothing system. Zaiane [16] *et al.* proposed a MLFPT algorithm to distribute workload among processors in a more fairly manner on a shared memory system. Liu *et al.* [17] proposed cache-conscious FP-array and lock-free parallelization optimizations to improve the mining performance on a shared memory system. Li *et al.* [9] proposed a parallel FP-growth algorithm named PFP on MapReduce framework. However, the major disadvantage of parallel FP-growth methods is that FP-tree is expensive to build and traverse. In this paper, we show that our proposed PFIN extends the advantages of FIN and can efficiently address large-scale FIM problems.

4. PFIN: The Proposed Method

4.1. FIN Algorithm

The FIN algorithm is composed of the following steps:

- FIN firstly scans the database once and computes a list of frequent items sorted by support in descending order, denoted as F_1;
- FIN then scans the database again and constructs a POC-tree which is an extended prefix tree structure. Each node of POC-tree consists of four fields: item, support, children, preorder;
- FIN scans the POC-tree by preorder traversal to generate the Nodesets of frequent 2-itemsets;

- FIN constructs a set-enumeration tree and employs a depth-first search strategy to directly mine frequent itemsets from it.

Table 1. A Transaction Database

TID	Transactions	Sorted Transactions
100	f, a, c, d, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, s, p	c, b, p
500	a, f, c, e, l, p, m, n	f, c, a, m, p

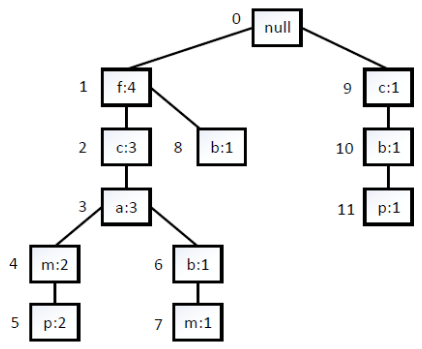

Figure 1. POC-Tree Constructed on the Database in Table 1

Table 1 shows a transaction database which contains five transactions. All items of each transaction will be sorted by F_1which is computed by counting the support of each item in the database and infrequent items will be removed. For example, in this database, suppose minSup = 3, we will get F_1: {(f:4), (c:4), (a:3), (b:3), (m:3), (p:3)} (the number after ":" indicates the support). The transaction {f, a, c, d, g, i, m, p} is pruned to {f, c, a, m, p}. A POC-tree then is constructed by inserting these pruned transactions with support = 1. The root of POC-tree is labeled as null. Pruned transactions with same prefix shares the same path of their branches and the support of each node along the same path is incremented by 1. Figure 1 shows an example of a POC-tree on database in Table 1.

The Nodeset of frequent item i is defined as a sequence of tuples in the form of $\{(\mathrm{preorder}_1 : \mathrm{support}_1), (\mathrm{preorder}_2 : \mathrm{support}_2), \ldots, (\mathrm{preorder}_m : \mathrm{support}_m)\}$ which represents all nodes registering i in the POC-tree. For any two items i_1 and i_2, $i_1 \succ i_2$ if and only if i_1 is ahead of i_2 in F_1. Given two frequent items F_1 and F_2 ($i_1 \succ i_2$), the Nodeset of 2-itemsets i_1 $i_2$denoted as Nodeset(i_1 i_2), is a subset of i_2's Nodeset, which is defined as follows: Nodeset(i_1 i_2) = $\{(\mathrm{preorder}_k{:}\mathrm{support}_k) \mid \exists$ a node, N, registering i_1, N is an ancestor of the node corresponding to $(\mathrm{preorder}_k{:}\mathrm{support}_k)\}$. For example, the Nodeset of p is {(5:2), (11:1)} and the Nodeset of fp is {(5:2)}. After the generation of Nodesets of 2-itemset, the Nodesets of (k+1)-itemset can be obtained by intersecting two Nodesets of k-itemset (k>2). A set-enumeration tree has been adopted by FIN to mine frequent k-itemsets (k>2) with a pruning technique called promotion to reduce the search space greatly. Promotion relays on the observation that given item i and itemset P ($i \notin P$), if the support of P is equal to the support of $P \cup \{i\}$, then the support of $A \cup P$, where $A \cap P = \emptyset \wedge i \notin A$, is equal to the support of $A \cup P \cup \{i\}$.

Previous studies have shown that FIN outperforms many current methods. However, it is still challenging for FIN to address large-scale FIM problems. One major challenge is that today's transaction databases may not fit in a single workstation's memory while FIN can only be executed in a stand-alone mode. Even if the transaction database can be loaded in memory, the corresponding set-enumeration tree constructed on the database may also be very huge. Hence, it is absolutely essential to parallelize FIN to achieve better scalability and performance especially over big data.

4.2. PFIN Outline

The PFIN algorithm consists of six steps as follows:

- **Partitioning**: Partitioning refers to dividing a large-scale dataset into distinct independent parts on different nodes. In the implementation of PFIN, the dataset is represented by a RDD and it has already been partitioned due to the properties of RDDs;
- **Parallel Counting**: This step counts the support of each items in parallel and computes a list of frequent items sorted by support in descending order, denoted as F_1;
- **Grouping Items with Load Balancing**: All items in F_1 are divided into Q groups called G-List and each group is given a unique group id. A hash-based load balancing strategy has been adopted to distribute workloads across nodes of the cluster more evenly;
- **Generating Conditional Transactions**: For minimize the amount of communication between nodes of the cluster, a technique called projection is adopted which maps each transaction to one or more conditional transactions according to group id. This step is described in detail in Section 4.4;
- **Constructing Local POC-trees and Generating the Nodesets of 2-itemset**: In this step, all conditional transactions with same group id are grouped into one partition. A local POC-tree is constructed on each partition. PFIN then scans these local POC-trees by preorder traversal to generate Nodeset($i_1 i_2$), where i_2's group id equals the group id of current partition;
- **Constructing Local Set-enumeration Trees and Mining Frequent Itemsets**: For each partition, a task is activated to construct a local set-enumeration tree and mine frequent itemsets from it. All outputs of tasks are aggregated by key in one RDD and cached for future use.

4.3. Parallel Counting

For counting the support of each item in parallel, a mapPartitions() function will be applied on a RDD which is created by referencing the transaction dataset from HDFS. In the mapPartitions() function, each partition of this RDD activates a task to count the support of items. The results of mapPartitions() function are aggregated by applying a reduceByKey() function which combines the intermediate results at the map side before the shuffle process to achieve a better performance than directly using groupByKey() function. Another way to finish the parallel counting job is to use map() function in stead. However, empirical results show that mapPartitions() function is significantly faster than map() function due to map() function needs to initialize more objects than mapPartitions() function. Frequent items can be obtained by applying filter() function. After that, this RDD will be collected to the driver program and sorted by support in descending order.

4.4. Grouping Items with Load Balancing

Load balancing is one of open problems for parallel frequent itemset mining. In our proposed PFIN algorithm, how to group items has a remarkable impact on the workload of each task. However, researchers had not come up with a good enough way to estimate the workload of each subtask. A hash-based load balancing strategy has been adopted to achieve better performance in this paper. The group id of each item can be computed as follows: $gid = rank \bmod numGroups$, where gid is the group id of a item, rank is the index of a item in F_1 and numGroups is specified by users. Group id can be used to generate conditional transactions which is discussed in detail in next subsection. It is supposed that those items with lower ranks generate conditional transactions with longer length and take more execution time to discover frequent itemsets. Based on this hash-based load balancing strategy, PFIN can distribute workloads more fairly across nodes of the cluster.

4.5. Generating Conditional Transactions

This is the key step of our proposed PFIN algorithm. As the G-List is usually small, it can be broadcast to each node of the cluster rather than copying it to all tasks. A technique called projection is discussed as follows. For each transaction, frequent items are filtered out and sorted by support in descending order. These ordered items will be scanned from tail to head. If it is the first time the gid of current item appeared in one transaction, then PFIN outputs all items in front of it, otherwise, it continues to scan. For example, suppose F_1 are divided into three groups and given G-List: {(f:0), (a:0), (c:1), (b:1), (m:2), (p:2)} (the number after ":" indicates the gid), the transaction {f, a, c, d, g, i, m, p} is pruned to {f, c, a, m, p} with infrequent items removed. Then it is scanned from tail to head and outputs three conditional transactions: (0, {f, c, a, m, p}), (1, {f, c, a}), (2, {f, c}), where key is a gid and value is a conditional transaction in a key-value pair.

4.6. Constructing Local POC-trees and Generating the Nodesets of 2-itemset

All conditional transactions with same group id are grouped into one partition in the form of <key = group id, value = conditional transactions>. A local POC-tree is constructed on each partition in the same way as the FIN method and no further communication overheads required during the process of mining frequent itemsets. Note that these grouped conditional transactions are ended with items whose group id equals the group id of current partition, Compared with FIN method, PFIN scans the local POC-tree by preorder traversal to only generate the Nodesets($i_1 i_2$), where i_2's group id equals the group id of current partition. For example, suppose F_1 is divided into 3 groups, m and p are both in the same group 2. The conditional transactions of group 2 are <key = 2, value = {f, c, a, m, p}, {f, c, a, b, m}, {c, b, p}, {f, c, a, m, p}>. Then a local POC-tree is constructed on these conditional transactions. The Nodesets of 2-itemset in group 2 generated from this local POC-tree are shown in Figure

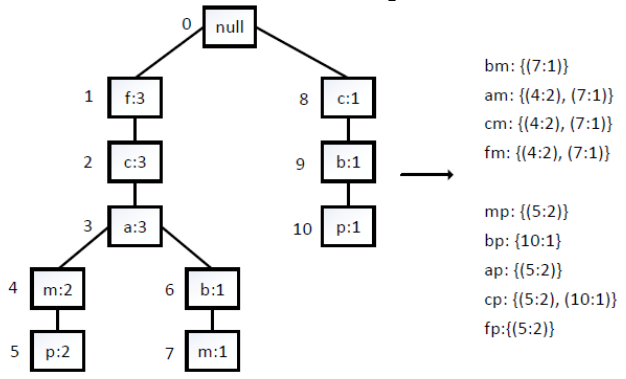

Figure 2. Constructing Local POC-Tree and Generating the Nodesets of 2-Itemset

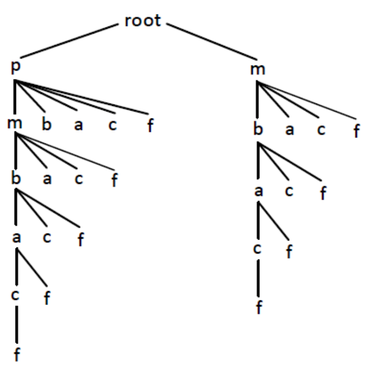

Figure 3. Constructing a Local Set-Enumeration Tree of Group 2

4.7. Constructing Local Set-enumeration Trees and Mining Frequent Itemsets

Different from the FIN method, PFIN divides the set-enumeration tree as follows. PFIN constructs a local set-enumeration tree on each partition to mine frequent itemsets in parallel. The local set-enumeration tree is considered as a subtree of a set-enumeration tree. The root of a local set-enumeration tree is labeled as null and the child nodes of the root only consists of items of current group. During the construction of local set-enumeration trees, all frequent itemsets can be discovered and aggregated from different partitions. Figure 3 presents the local set-enumeration tree generated from the conditional transactions of group 2.

5. Experiments

5.1. Experiment Setup

In this section, we evaluate the performance of our proposed PFIN algorithm in terms of runtime and scalability by comparing it against PFP and FIN on a range of real datasets. PFP is an implementation of the parallel FP-growth algorithm on a distributed system and it is one of state-of-the-art parallel FIM algorithms. PFIN, PFP and FIN are all implemented in Scala. All the experiments of PFIN and PFP were performed on a Spark 1.3.0 cluster of 7 nodes including one master and six workers, where each node has 32 Intel Xeon E5-2660 CPUs running at 2.2GHz, 192GB memory and 1TB disk. FIN was performed on a single node of the cluster. Note that the runtime here includes input time and execution time. We persist the output of PFIN and PFP in memory and hold the output of FIN in a StringBuffer.

Four real datasets with various characteristics and domains were used in our experiments. These datasets are available from the FIMI repository ({http://fimi.ua.ac.be}). The chess and connect datasets are generated from different game steps. The mushroom dataset describes the characteristics of poisonous and edible mushrooms. The webdocs dataset was built from a spidered collection of web html documents. Table 2 summaries the properties of these datasets including the average transaction length(#Avg.Length), the number of items (#Items), the number of transactions (#Trans) and the type of each dataset.

Table 2. The Summary of the Used Datasets

Dataset	#Avg.Length	#Items	#Trans	Type
chess	37	76	3,196	Dense
connect	43	129	67,557	Dense
mushroom	23	119	8,124	Dense
webdocs	177	5,267,656	1,692,082	Sparse

5.2. Speed Performance Analysis

In this subsection, we compare PFIN against PFP and FIN in terms of runtime with various values of minimum support. Figure 4 shows the runtime of all algorithms on dataset chess. For high minimum support, the performance of PFIN and PFP are very close. In situations with lower minimum support, the solution search space can be so enormous that both PFIN and PFP run much more slowly. However, we observe that the runtime of PFP increases much faster than that of PFIN. For all experiments on dataset chess, FIN performs worse than PFIN and PFP due to the limit computation capability and memory capacity of a single node. Figure 5 shows the runtime of all algorithms on dataset mushroom. The results demonstrate that PFIN nearly outperforms PFP by a factor of 2 or 3 for all values of minimum support. FIN performs well with the high minimum support. However, the performance difference between FIN and other parallel algorithms was enlarged significantly with lower minimum support. Figure 6 shows that PFIN runs much faster than PFP and FIN on dataset connect. Figure 7 shows the runtime of all algorithms on dataset webdocs. PFIN still performs best for each minimum support. FIN performs worst and is much slower than other parallel algorithms.

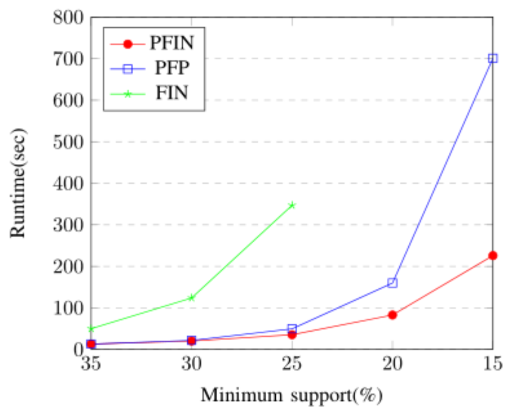

Figure 4. Runtime on Dataset Chess

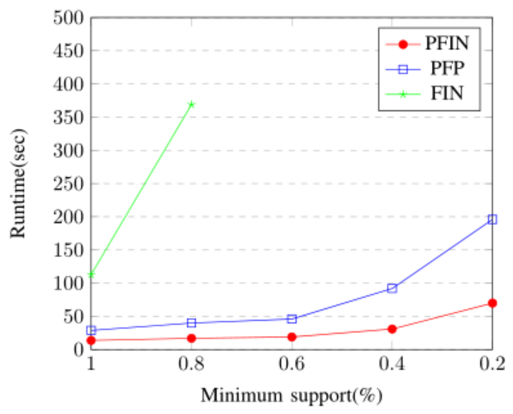

Figure 5. Runtime on Dataset Mushroom

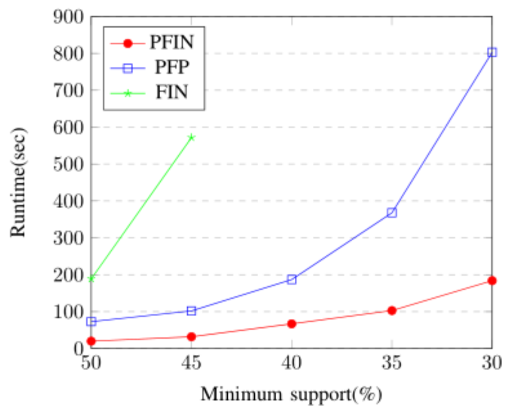

Figure 6. Runtime on Dataset Connect

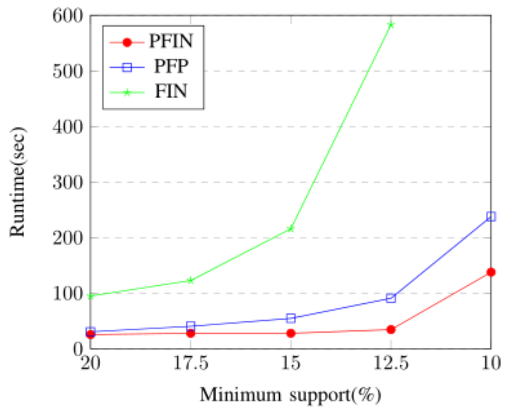

Figure 7. Runtime on Dataset Webdocs

The above results have shown that our proposed PFIN extends all advantages of the FIN method and achieves a significant performance improvement on different real transaction datasets. Compared with FIN, the performance difference between PFIN and FIN is close with high minimum support and is enlarged rapidly with lower minimum support. PFIN employs a technique called projection to partition the transaction dataset so that PFIN is able to handle large-scale datasets which may not fit into a single workstation. Moreover, PFIN distributes the computation to many tasks by dividing the set-enumeration tree into local set-enumeration trees which can be executed to mine frequent itemsets in parallel. Our experimental results show that this dividing strategy achieves ideal performance and near-linear scalability. Compared with PFP, PFIN adopts a more efficient data structure called Nodeset to efficiently represent itemsets. Based on Nodeset, PFIN avoids the time consuming process of constructing and traversing numerous conditional FP-trees by simply Nodeset intersection. In addition, PFIN adopts a pruning strategy called promotion to greatly reduce the search space and extremely improve the mining performance.

5.3. Scalability Performance Analysis

In this subsection, we compare PFIN against PFP with respect to sizeup. To measure the performance of sizeup, we hold the number of cores to 224 and grow the size of the datasets by replicating original datasets to different times in size. Figure 8, Figure 9, Figure 10, Figure 11 show the sizeup performance for difference datasets. We can see that as the size of transaction dataset increases, the execution time costs for PFP and PFIN both go near-linear. However, our proposed PFIN still performs better than PFP on all transaction datasets. The experimental results indicate that PFIN can achieve virtually linear scalability in performance.

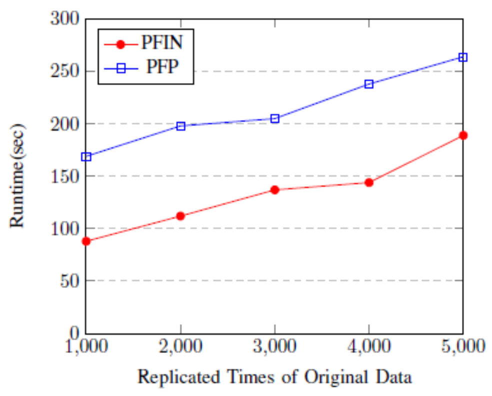

Figure 8. Sizeup Performance Evaluation on Dataset Chess (minSupp = 20%)

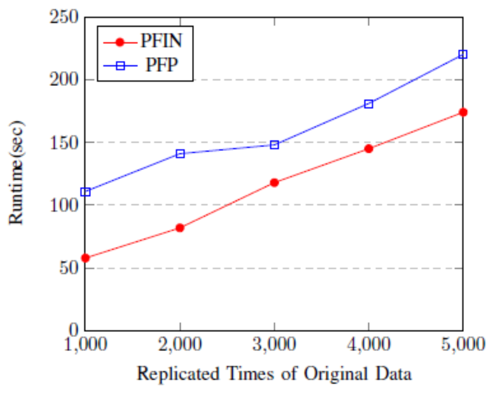

Figure 9. Sizeup Performance Evaluation on Dataset Mushroom (minSupp = 0.4%)

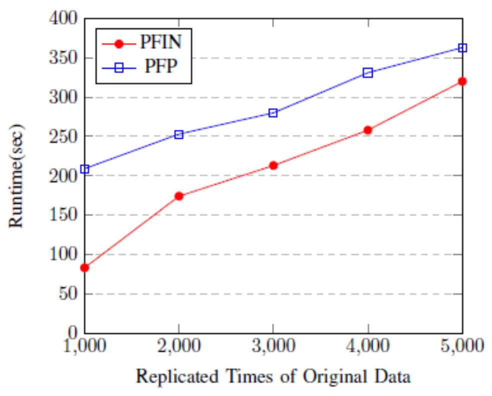

Figure 10. Sizeup Performance Evaluation on Dataset Connect (minSupp = 40%)

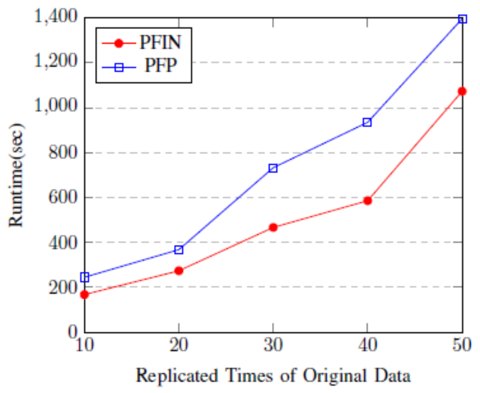

Figure 11. Sizeup Performance Evaluation on Dataset Webdocs (minSupp = 15%)

6. Conclusion

In this paper, we propose PFIN, a new parallel frequent itemset mining algorithm based on a novel data structure called Nodeset to efficiently represent itemsets. This algorithm employs a strategy called projection to partition the transaction database which virtually eliminates further communication overheads across nodes of the cluster and divides the set-enumeration tree in such a way that each partition can be executed to discover the frequent itemsets in parallel. During the construction of local set-enumeration trees, PFIN employs a pruning strategy called promotion to greatly reduce the search space. Moreover, a hash-based load balancing strategy has been adopted to optimize resource use and maximize throughput. We have conducted extensive experiments in terms of runtime and scalability to compare our proposed algorithm against PFP which is one of state-of-the-art parallel FIM algorithms. Empirical results show that PFIN achieve outstanding efficiency and we believe PFIN can be used to address large-scale FIM problems in the context of big data.

Reference

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large database", Proceedings of SIGMOD, **(1993)**, pp. 207-216.
[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases", Proceedings of VLDB, **(1994)**, pp. 487-499.
[3] J.-S. Park, M. S. Chen, and P. S. Yu, "An Effective Hash-based Algorithm for Mining Association Rules", ACM SIGMOD Conference, **(1995)**.
[4] H. Toivonen, "Sampling large databases for association rules", VLDB Conference, **(1996)**.
[5] D. C. Anastasiu, J. Iverson and S. Smith, "Big Data Frequent Pattern Mining", Springer International Publishing, **(2014)**, pp. 225-259.
[6] R. Agrawal and J. C. Shafer, "Parallel mining of association rules", IEEE Transactions on Knowledge and Data Engineering, vol. 8, no. 6, **(1996)**, pp. 962–969.
[7] Y. Ye and C. C. Chiang, "A Parallel Apriori Algorithm for Frequent Itemsets Mining", Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06), **(2006)**, pp. 87-93.
[8] I. Pramudiono and M. Kitsuregawa, "Parallel FP-Growth on PC cluster", PAKDD Conference, **(2003)**.
[9] H. Li, Y. Wang, D. Zhang, M. Zhang and E. Y. Chang, "PFP: parallel fp-growth for query recommendation", RecSys '08 Proceedings of the 2008 ACM conference on Recommender systems, New York, NY, USA, **(2008)**, pp.107-114.
[10] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", Proceedings of OSDI. USENIX Association, **(2004)**.
[11] Z. H. Deng, and S. L. Lv, "Fast mining frequent itemsets using Nodesets", Expert Systems with Applications, **(2014)**, pp. 4505–4512.
[12] R. Rymon, "Search through systematic set enumeration", Proceeding of International Conference on principles of knowledge representation and reasoning, **(1992)**, pp. 539–550.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing", Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, **(2011)**.
[14] M. Y. Lin, P. Y. Lee and S. C. Hsueh, "Apriori-based frequent itemset mining algorithms on mapreduce", Proceedings of the Sixth International Conference on Ubiquitous Information Management and Communication, New York, NY, USA, **(2012)**, pp.76–76.
[15] H. Qiu, R. Gu, and Y. Huang, "YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark", Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, Phoenix, AZ, **(2014)**, pp.1664-1671.
[16] O. R. Zaiane, M. E. Hajj and P. Lu, "Fast parallel association rule mining without candidacy generation", ICDM Conference, **(2001)**.
[17] L. Liu, E. Li, Y. Zhang, and Z. Tang, "Optimization of frequent itemset mining on multiple-core processor", VLDB '07 Proceedings of the 33rd international conference on Very large data bases, **(2007)**, pp. 1275-1285.

Authors

Chen Lin, he is a postgraduate student and pursuing for master degree in the department of computer science and technology at East China Normal University. His current research interests on big data and data mining.

Junzhong Gu, he is a Lifetime Professor in the department of computer science and technology at East China Normal University. He has over 200 publications in the form of journal papers, conference papers, and books. His research interests mainly focus on big data, data mining, text mining, ontology and knowledge discovery.