DisCANTree: A Distributed Algorithm for Incremental Frequent Itemset Mining based on MapReduce

Wen Xiao \(^1\)\(^a\), Juan Hu \(^2\)\(^b\)

\(^1\)Key Laboratory of Unmanned Aerial Vehicle Development & Data Application of Anhui Higher Education Institutes, Wanjiang University of Technology, Maanshan 243000, China

\(^2\)Ma'anshan Engineering Technology Research Center for Wireless Sensor Network and IntelliSense, Wanjiang University of Technology, Maanshan 243000, China

\(^a\)cyees@163.com  \(^b\)hujuan19851985@163.com

Abstract. Frequent itemset mining is one of the most important data mining tasks. Classical frequent itemset mining algorithms need to store data in a centralized way and run in a batch way, which cannot meet the requirements of fast updating big data mining. In this paper, we propose a distributed incremental frequent itemset mining algorithm, DisCANTree, which uses CANTree to store the conditional database, achieves the load balance between nodes by grouping all items, updates the new transaction to the existing CANTree to avoid the load of tree reconstruction, and uses the efficient FP-Growth algorithm to mine CANTree to generate frequent itemsets. The popular distributed programming model MapReduce and its open source system Hadoop are used to implement the DisCANTree algorithm. The experimental results show that the DisCANTree algorithm has more advantages than the most popular PFP algorithm in performance as well as the number of transferred records between nodes, and especially suits for the fast updating sparse big data.

1. Introduction

Frequent Itemset Mining (FIM) is one of the most important data mining tasks. Its purpose is to find frequent itemsets with greater than or equal to the minimum support in transaction dataset. It is the basis of machine learning tasks such as association rule analysis and outlier monitoring. Since it was proposed [1], it has attracted more and more attention. Classical algorithms for mining frequent itemsets can be divided into three categories. The first kind of algorithm generates and counts candidate itemsets by iteration to generate frequent itemsets. Typical algorithms include Apriori[2], DHP, etc. The second kind of algorithm does not need to generate candidate itemsets. By storing the transaction dataset in a special data structure (generally a prefix tree), it generates frequent itemsets directly by traversing the prefix tree. Typical algorithms include FP-Growth[3], LP-Tree[4], FIUT[5], IFP[6], FPL/TPL[7], etc. The last kind of algorithm transforms the transaction dataset from horizontal format to vertical format firstly, then directly generates frequent itemsets through intersection operation. Typical algorithms include Eclat[8], dEclat, etc. Among the classical algorithms, FP-Growth is generally considered to have the best performance [9].

Based on the assumption that the dataset is centralized and static, the classical FIM algorithm mines the whole dataset in a batch way. However, in the real world, the dataset may be continuously updated, and new transactions are constantly inserted into the dataset, resulting in infrequent itemsets
becoming frequent and vice versa. For the new dataset after updating, if it is still mined in a batch way, its efficiency is obviously very low. If we can make full use of the results that are generated in the previous stage of mining, and only incremental mining of the newly arrived dataset will improve the efficiency significantly.

For incremental frequent itemset mining, researchers have proposed some algorithms, such as FUP[10], FUP2[11], UWEP[12], GSP[13], PreLarge[14] which are based on Apriori algorithm, AFPIM[15], CATSTree[16], CANTree[17] which are based on prefix tree. Although these algorithms for incremental frequent itemset mining based on Apriori make full use of all candidate itemsets and their counting generated in previous stage, it needs to generate candidate itemsets and count them multiple times, which is not suitable for the incremental mining of big data. The incremental frequent itemset mining algorithms based on prefix tree use the newly arrived transactions to update the prefix tree generated in the previous stage. After updating, FPGrowth can be directly used for mining, with better efficiency [17].

Frequent itemset mining is a task with high requirements for calculation and storage. In the era of big data, transaction datasets are very large in size and update very fast. Traditional centralized incremental frequent itemset mining algorithm cannot meet the needs of big data mining. There are some challenges: it is unable to store whole dataset and intermediate results generated in the mining process on a single computer; the duration of mining is too long to bear, etc. We need to design a distributed incremental frequent itemset mining algorithm to meet the above challenges.

MapReduce is an easy-to-use and powerful distributed programming model proposed by Google, which has the advantages of automatic fault tolerance, data localization optimization, node scalability, etc., and has been widely used in the field of distributed data mining. Researchers have proposed many algorithms [18-31] based on MapReduce for distributed frequent itemsets mining, which have achieved good function. The execution flow chart of a MapReduce job is shown in Fig 1.

Hadoop[32] is an open source implementation system of MapReduce, and provides a distributed file system named HDFS. HDFS is used to store big datasets, which can be automatically partitioned. HDFS automatically stores the data partitions in different nodes. The Map() method in MapReduce job is executed on the node where the data is located. The purpose of reducing I/O and improving efficiency is achieved through data localization.

In this paper, we propose a distributed incremental frequent itemset mining algorithm, named DisCANTree, which uses Ca-nonical-Order Tree (CANTree) to store the conditional database, achieves the load balance between nodes in the mining stage by grouping all items, updates the newly arrived transactions into the existing CANTree to avoid the load of tree reconstruction, and uses the efficient FPGrowth algorithm to find frequent itemsets by traversing CANTree. The popular distributed programming model MapReduce and its open source system Hadoop are used in the implementation of DisCANTree, which is especially suitable for mining big data with fast update and sparse features.

This paper is organized as follows: in Section 2, the problem and symbol are defined, and the related work is discussed; An distributed frequent itemset mining algorithm based on MapReduce named DisCANTree is proposed in Section 3; The experimental results are discussed in Section 4; We summarize the paper in Section 5.
2. Preliminary and Related Work

2.1. Problem Definition and Symbols

Let \( I = \{i_1, i_2, \ldots, i_n\} \) be a set of items. Let \( X = \{i_1, i_2, \ldots, i_k\} \subseteq I \) be an itemset. Let \( T \) be a transaction, \( T = \{\text{tid}, X'\} \) in which \( \text{tid} \) is an identifier and \( X' \) is an itemset. Given an itemset \( X \subseteq I \), a transaction \( T \) contains \( X \) if and only if \( X \subseteq X' \). Let \( D = \{T_1, T_2, \ldots, T_n\} \) be a database of transactions. \(|D|\) is the number of transactions in \( D \).

For an itemset \( X \), its support is the percentage of transactions in \( D \) which contains \( X \), and its support count, denoted by \( X.\text{sup} \), is the number of transactions in \( D \) containing \( X \). An itemset \( X \) is frequent if its support is no less than the minimum support \( s \). The set of frequent items in \( D \) denoted \( \text{FList} = \{i | i \in I \land i.\text{sup} \geq |D| \times s\} \).

The symbols used in this paper are shown in Table 1.

| Symbol | Description |
|--------|-------------|
| \( D \) | Original Transaction Dataset |
| \(|D|\) | Number of Transactions in \( D \) |
| \( D_i \) | A Data Partition of \( D \) |
| \( N \) | Newly Arrived Transaction Dataset |
| \(|N|\) | Number of Transactions in \( N \) |
| \( N_i \) | A Data Partition of \( N \) |
| \( U \) | Union of \( D \) and \( N \), \( U = D \cup N \) |
| \(|U|\) | Number of Transactions in \( U \) |
| \( s \) | Minimum Support |
| \( DB_i \) | A Conditional Database |
| \( T_i \) | A Transaction |
| \( x_i \) | An Item in A Transaction |
| \( \text{IList} \) | All Items |
| \( \text{GList} \) | All Groups of Items |
| \( G_i \) | A Group of Items |
| \( \text{gid} \) | NO. of A Group of Items |
| \( \text{CT}_i \) | A CanTree |
| \( \text{GL}_i \) | Frequent Itemsets in \( DB_i \) |

The problem of incremental frequent itemsets mining is to make full use of the results of \( D \) in the previous stage, to mine \( N \) incrementally, and to efficiently find frequent itemsets with support degree which is no less than \( s \) in \( U \).

2.2. \textit{CATSTree}

Cheung et al. designed a special tree structure named Compressed Arrayed Transfer Sequence Tree (CATSTree) [16] which was used for incremental frequent itemsets mining.

It only needs to scan the dataset once to construct CATSTree, and insert all the items in every transaction in the dataset into the tree sequentially. When inserting \( T_i \), it will start from the root node in CATSTree and compare the items in \( T_i \) with each child of the current node. If they are the same, it will merge them, change the current node to the merged node, and compare the next item in turns. If there are no same items, it will insert the items in \( T_i \) into CATSTree as a new path. If the support of a node in the tree is higher than that of its ancestor, it will exchange it with its ancestor to ensure that the support of each node is not greater than that of its ancestor. After CATSTree construction is completed, FP-Growth algorithm can be used to mine frequent itemsets by traversing CATSTree. From the above discussion, we can see that CATSTree is similar to FPTree [3], the nodes on each path are sorted by
support descendingly, but CATSTree stores all items in the transaction, while FPTree only stores frequent items in the transaction.

Although CATSTree stores all items in transactions of dataset, and has the characteristics of simple construction and easiness to insert transaction, it also has two obvious disadvantages: firstly, there are operations such as searching common items, exchanging and merging nodes with the change of support in the construction of CATSTree, which will cause much load; secondly, according to the characteristics of CATSTree, when using FP-Growth algorithm for mining, it is necessary to traverse up and down to generate a conditional database of an item, which will also cause a lot of load.

2.3. CANTree
Leung et al. proposed Canonical Order Tree (CANTree) [17] to overcome the above disadvantages of CATSTree. Like CATSTree, CANTree also stores all the items in the transaction of dataset. Before the transaction is inserted, all the items in the transaction will be arranged in a predefined order (either dictionary order or a certain attribute of the item) so that the node order in the tree is independent of the support of item, and no additional node exchange operation is required, while greatly reduces the load of searching common items in the path. It only needs to traverse CANTree from the up direction to reduce the load.

For the newly arrived transaction dataset N, the above method can be used to easily insert the new transaction into the CANTree for incremental frequent itemset mining. For the constructed CANTree, the FP-Growth algorithm can also be used for mining. Masome et al. proposed a new algorithm for mining on CANTree [33], which is similar to FP-Growth. The difference is that only frequent items on the path are selected when creating the conditional database of items, reducing the construction cost of CANTree corresponding to the conditional database.

According to the characteristics of CANTree, it stores all items in the dataset, and it is very convenient to insert new transactions, which is very suitable for incremental frequent itemset mining. CANTree only needs to scan dataset once to construct, and there is no additional operation of exchanging nodes in the construction process, so the efficiency of searching common items and constructing CANTree is further high. Similar to FPTree, when FP-Growth algorithm is used to mine CANTree, an independent conditional database can be generated for each item. The mining of all conditional databases is independent of each other, which can realize distributed mining, and can be realized by using MapReduce distributed computing model conveniently.

3. Proposed DisCANTree

3.1. Algorithm Framework
Given D and N and s, it is assumed that there are p nodes in the distributed system. The framework of the DisCANTree algorithm is shown in algorithm 1.

**Algorithm 1. DisCANTree**

| Input: |
|--------|
| · D: Original Transaction Dataset |
| · N: Newly Arrived Transaction Dataset |
| · s: minimum support |

| Output: |
|--------|
| · All frequent itemsets in U |

| Procedure: |
|------------|
| (1) Distributed counting all items in D to get the set of all items IList |
| (2) Balanced grouping of IList to get n groups GList={G1,…Gn} |
| (3) Generating conditional database of all groups {DB1,…DBn}, redistributing conditional databases to nodes |
| (4) Constructing CANTree of DBi on each node, and get CTList = {CT1,…CTn} |
(5) Each node using s and FPGrowth algorithm to CTi,
(6) Each node retaining CTi
repeat
(7) Distributed counting of all items in N, update IList
(8) Updating GList according to the updated IList
(9) Redistributing the transactions in N according to the updated GList
(10) Each node updating the CTi, with the newly arrived transactions
(11) Each node using s and FPGrowth algorithm to CTi, generating GLi
(12) Each node retaining CTi
until No new transaction data

From the pseudo description of the DisCANTree algorithm, we can see that step 1-6 is used for processing D and step 7-12 is used for processing N incrementally. When N arrives, the iteration is to be executed in step 7-12.

Whether mining D or N, it mainly includes five basic steps: (1) counting items in a distributed way (row 1, 7); (2) to achieve load balancing, all items are grouped according to the load balancing principle (row 2, 7); (3) constructing conditional database according to the GList to ensure that the size of conditional database allocated by each node is basically identical (row 3, 9); (4) using FPGrowth algorithm on CANTree which has been constructed or updated for independent mining to generate frequent itemsets (row 5, 11); (5) each node retaining CANTree for incremental mining in the next stage (row 6, 12).

The efficiency improvement of the DisCANTree algorithm compared with the batch way algorithms is mainly reflected in the following two aspects: First, it does not need to redistribute U but only redistributes the transactions in N, reducing the number of records that need to be transferred between nodes; Second, it reduces the load of reconstructing the CANTree. Because the tree construction is a recursive operation, avoiding the reconstruction of CANTree can greatly improve the efficiency of the mining stage.

3.2. Implementation based on MapReduce
According to the pseudo description of DisCANTree, the distributed storage of D and N can be realized automatically through HDFS. Two MapReduce jobs are required to complete the distributed mining of D and incremental mining of N respectively. The flow chart of the algorithm is shown in Fig 2.
The implementation of DisCANTree is mainly divided into two phases, each of which uses two MapReduce jobs.

The first phase is InitMining. The goal of this phase is to process D. Job 1 is used to count all items in D in a distributed way to get IList. Job 2 is used to complete the distributed mining of D. The Map() method generates conditional database \{DB_1, DB_2, ..., DB_n\} according to GList and redistribute them to nodes, FPGrowth algorithm is used for mining after constructing CANTree of DB_i in Reduce() method.

The second phase is IncrementalMining. Job 3 is similar to Job 1 in that it counts the items in N in a distributed way. Job 4 is similar to Job 2 in that it redistributes the transactions in N and updates CANTree before mining. The first run of DisCANTree requires two phases, and the next new dataset N only needs a second phase.

Job 1 and Job 3 finish distributed counting of all items in D or N as described in Algorithm 2.

### Algorithm 2. ItemDistributedCounting

**Input:** \(T_i \in D\) or \(N\)

**Output:** IList

| Line | Code |
|------|------|
| 1    | function Map(key offset, values \(T_i\)) |
| 2    | for each \((x_i \in T_i)\) |
| 3    | output\((<x_i, 1>)\) |
| 4    | end for |
| 5    | end function |
| 6    | function Reduce(key \(x_i\), values list(\(x_i\))) |
| 7    | sum\(\leftarrow 0\) |
| 8    | for each \((x_i \in \text{list}(x_i))\) |
| 9    | sum\(\leftarrow \text{sum} + 1\) |
The balanced grouping of items in IList to generate Glist is the basis of dataset redistribution. Its purpose is to ensure that the size of conditional database corresponding to each group is nearly same, so that the load of each node in mining is close.

DisCANTree algorithm uses the classical Round-Robin load balancing method to group items. First, all items in IList are arranged in descending order according to the support, and then IList is divided into n groups. Let Ii be the i-th item in IList after sorting, and Gi is the number of the group Ii is divided into as follows:

\[ G_i = \begin{cases} n \times (i \mod n = 0) \\ i \mod n \end{cases} \]  

(eq. 1)

The process of updating GList in IncrementalMining phase is similar to the method of calculating Gi above. First, it updates the count of items generated from N to IList. If \( x_i \in G_i \), the count of \( x_i \) in \( G_i \) will be updated; If \( x_i \) does not exist in any \( G_i \), it will be reassigned to n groups according to Round-Robin method to get a new GList.

Job 2 is similar to Job 4 in that it redistributes the transactions in D or N according to GList to get the conditional database \{DB1, DB2, ..., DBn\} in Map() method. Each node constructs or updates the corresponding CTi according to the conditional database, and then generates frequent itemsets in Reduce() method. The specific process is described in algorithm 3.

**Algorithm 3. RedistributeAndMining**

**Input:** \( T_i \in D \text{ or } N \), s, GList  
**Output:** GLi

```
(1) function Map(key offset, values Ti)
(2)   for each (xi in Ti)
(3)     for each (gid in GList)
(4)       if Ggid contains xi then
(5)         output(<gid, Ti>)
(6)       endif
(7)   end for
(8) end for
(9) end function
(10) function Reduce(key gid, values list(Ti))
(11)   DBi ← list(Ti)
(12)   CTi ← ConstructCANTree(DBi)
(13)   // CTi ← UpdateCANTree(CTi, DBi)
(14)   GLi ← FPgrowth(CTi, s)
(15)   for each (xi in GLi)
(16)     output(<xi, xi.sup>)
(17) end for
(18) end function
```

The CTi in Job 2 is constructed from the conditional database (row 13), while the CTi in Job 4 is obtained by updating the CTi in the previous stage with the new conditional database (row 14).

**4. Analysis of Experimental Results**

Experiments use Hadoop 2.6.5 as the platform to implement DisCANTree algorithm. A high-performance workstation is used to simulate a Hadoop cluster consisting of one master node and eight slave nodes. The MapReduce 2.0 (yarn) scheduling method is used. The virtual machine configuration in the cluster is shown in Table 2.
Table 2. Virtual machine configuration of Hadoop cluster

| Node     | Cores | Memory | OS     | Physical Configuration               |
|----------|-------|--------|--------|--------------------------------------|
| Master   | 2     | 8GB    | CentOS7.5 | Intel Xeon E5-2620Cores 64GB RAM     |
| Slave01-08 | 2    | 6GB    | CentOS7.5 | V4@2.10GHz,8 Cores 64GB RAM          |

The DisCANTree algorithm is written in Java language (JDK 1.8.0). We choose the most popular MapReduce-based distributed frequent itemset mining algorithm PFP [26] for comparison, use the real dataset retail and the composite dataset T10I4D100K generated by IBM generator from FIMI[34] as the test datasets, and the attributes of the experimental datasets are shown in Table 3.

Table 3. Attributes of datasets used in Experiments

| Name       | transactions | items | Avg.Length | Size(KB) |
|------------|--------------|-------|------------|----------|
| T10I4D100K | 100,000      | 870   | 10         | 3.928    |
| retail     | 88,162       | 16,470| 10         | 4,070    |

4.1. Experiment of Tree Structures

The PFP algorithm uses FPtree to store conditional database, while the DisCANTree algorithm uses CANTree to store the conditional database. Both of them use FPgrowth as the frequent itemset mining algorithm. In the independent mining stage, both algorithms include two steps: constructing tree structure and mining on tree. In this group of experiments, the construction and mining time of the two tree structures are shown in Fig 3.

(a) T10I4D100K  

(b) Retail  

Fig.3 Construction and mining time comparison of two tree structures under different minimum support
From Fig 3 we can draw the following conclusions:

1. Whether FP Tree or CAN Tree, its construction load is far greater than the mining load. Incremental mining only needs to update the constructed CAN Tree, which can greatly reduce the load of tree reconstruction.

2. The load of constructing FP Tree is inversely proportional to the minimum support. The lower the minimum support is, the more transactions and frequent items need to be inserted into FP Tree, the higher the load is. Because all transactions and items are stored in CAN Tree, constructing it has nothing to do with the minimum support, and its construction load is only proportional to the number of transactions and items. In general, the load of constructing CAN Tree is larger than that of constructing FP Tree, but with the decrease of minimum support, the gap between them will gradually decrease.

3. Using the same FP Growth algorithm, FP Tree is more efficient than CAN Tree, because FP Tree only contains frequent items, while CAN Tree contains all items in order to realize incremental mining.

4. As can be seen from Table 3, the retail is sparser than the T10I4D100K. For sparse dataset, the load of construction tree is more than that of mining, so incremental mining can achieve better results.

4.2. Experiment of Load Balance
Dis CAN Tree uses Round-Robin load balance method to group the items in D or N according to the support. Its purpose is to make each conditional database \{DB_1, DB_2, ..., DB_n\} have mostly same size, which is conducive to load balancing in the mining stage. In this group of experiments, Round-Robin method is used to group all items of the two datasets according to the support. The number of groups is set to 4, 8 and 16 respectively. In these three cases, the number of transactions in each conditional database is shown in Table 4.

| Table 4. The number of transactions in each conditional database |
|---------------------|---------------------|---------------------|---------------------|---------------------|
|                     | T10I4D100K          | Retail             |                     |                     |
|                     | 4                   | 8                   | 16                  | 4                   | 8                   | 16                  |
| DB_1                | 91654               | 71783               | 47388               | 78662               | 49876               | 33772               |
| DB_2                | 91194               | 72308               | 45539               | 73531               | 71044               | 32418               |
| DB_3                | 90454               | 72034               | 49771               | 69916               | 64539               | 63653               |
| DB_4                | 91480               | 71392               | 45282               | 70066               | 55895               | 33877               |
| DB_5                | -                   | 71040               | 48228               | -                   | 55812               | 56177               |
| DB_6                | -                   | 71662               | 46818               | -                   | 54010               | 32764               |
| DB_7                | -                   | 71317               | 48252               | -                   | 49337               | 42124               |
| DB_8                | -                   | 70049               | 45194               | -                   | 49913               | 33561               |
| DB_9                | -                   | -                   | 48074               | -                   | -                   | 41104               |
| DB_10               | -                   | -                   | 45482               | -                   | -                   | 33650               |
| DB_11               | -                   | -                   | 47538               | -                   | -                   | 40431               |
| DB_12               | -                   | -                   | 45982               | -                   | -                   | 32273               |
| DB_13               | -                   | -                   | 47147               | -                   | -                   | 33678               |
| DB_14               | -                   | -                   | 44793               | -                   | -                   | 32388               |
| DB_15               | -                   | -                   | 46913               | -                   | -                   | 33926               |
| DB_16               | -                   | -                   | 45597               | -                   | -                   | 32230               |

From Table 4, it can be seen that the Round-Robin load balance method is used to achieve a better load balancing effect. The size of each conditional database is basically the same, ensuring that the load of each node in the mining stage is very close, realizing the load balancing of distributed mining. Because T10I4D100K is more intensive, the size of each conditional database is more balanced, but
for the retail which is sparse, there are still some conditional databases whose size is quite different from other conditional databases.

4.3. Experiment of Incremental Proportion
This group of experiments compares the performance of PFP and DisCANTree and the number of records transferred between nodes under different incremental proportion. The minimum support is fixed at 0.1%. The experimental results are shown in Fig 4.

It can be seen from Fig 4 (a) that, although PFP algorithm has good linear acceleration, DisCANTree has better performance than PFP in two datasets. The main reason for the performance improvement of the DisCANTree algorithm is that it avoids the heavy load caused by the reconstruction of the corresponding CANTree of the conditional database. For sparser dataset retail, the performance improvement is more significant because the load of constructing tree of sparse datasets is heavier. It can also be seen from this figure that when the incremental proportion is at the value of 25%, the performance improvement of DisCANTree is the most obvious compared with that of PFP algorithm; with the increase of incremental proportion, the gap between |D| and |N| is smaller and smaller, and the performance improvement is also smaller and smaller. Therefore, the DisCANTree algorithm has better efficiency in the case of |D|>>|N|.

Reducing the number of records that need to be transferred between nodes is an important goal of designing distributed algorithms. As can be observed from Fig 4 (b) that in terms of the number of records that need to be transferred between nodes, the DisCANTree algorithm has a significant improvement over the PFP algorithm. The reason is that the PFP algorithm needs to redistribute all the transactions in U to construct the conditional database, while the DisCANTree algorithm only needs to
redistribute the transactions in N. Similarly, when the incremental proportion is 25%, the gap between the two algorithm is the largest, which shows that the DisCANTree algorithm has better efficiency in the case of |D|>>|N|. The DisCANTree algorithm is particularly suitable for the needs of fast changing big data mining.

4.4. Experiment of Minimum Support

Minimum support is the most important parameter in frequent itemset mining. This group of experiments compares the performance and the number of records transferred between nodes of PFP and DisCANTree algorithm under different minimum support. The incremental proportion is fixed at 100%. The experimental results are shown in Fig 5.

![Performance](image1)

(a) Performance

![Number of records transferred between nodes](image2)

(b) Number of records transferred between nodes

Fig.5 The performance and the number of records transferred between nodes of the two algorithms under different minimum support

As can be seen from Fig 5 (a), in T10I4D100K which is relatively dense, the performance optimization of DisCANTree is relatively stable compared with that of PFP algorithm, which shows that both algorithms have good minimum support scalability. In the sparse dataset retail, when the support is small (s = 0.01% and 0.025%), the performance of the DisCANTree algorithm is better than that of the PFP algorithm; when the support is large (s = 0.05% and 0.1%), the performance of the DisCANTree algorithm is worse than that of the PFP algorithm. Therefore, DisCANTree has the best performance improvement under the small minimum support of sparse dataset, and is especially suitable for incremental mining of big data.

From the discussion in Section 3.2, we can see that the number of records transferred between nodes by the DisCANTree algorithm is only related to |N| and has nothing to do with the minimum support. It can also be observed from Fig 5 (b).
From Fig 5 (b), it can be seen that on the dense dataset T10I4D100K, the number of records transferred between nodes by either PFP algorithm or DisCANTree algorithm is relatively stable, almost unaffected by the change of minimum support, and has a relatively stable optimization; On the sparse dataset retail, the number of records transferred between nodes of PFP algorithm decreases with the increase of minimum support. It can be inferred that when the minimum support increases to a certain value, the number of records transferred between nodes of PFP algorithm will be smaller than that of the DisCANTree algorithm, but in the case of the minimum support is small, the number of records transferred between nodes of the DisCANTree algorithm will undoubtedly be smaller.

5. Conclusion
In order to improve the efficiency of distributed incremental frequent itemsets mining in frequently updated transaction big data, this paper proposes a distributed incremental frequent itemset mining algorithm named DisCANTree. This algorithm uses CANTree to store the conditional database. When the new transaction data N arrives, it only needs to use the transaction in N to update the constructed CANTree without reconstruction, which significantly improves the performance and reduces the number of transferred records between nodes. The experimental results show that the Round-Robin method can achieve good load balance by grouping items according to the support, and the DisCANTree algorithm has better performance than the most popular PFP algorithm, especially in the case of |D| >> |N| when the minimum support is small and the dataset is sparse, the DisCANTree algorithm has significant performance improvement.

Acknowledgement
This work supported by the natural science foundation of the universities in Anhui province under Grant NO. KJ2019A1274

References
[1] Agrawal, Rakesh, Tomasz Imieliński, and Arun Swami. "Mining association rules between sets of items in large databases." Acm sigmod record. Vol. 22. No. 2. ACM, 1993.
[2] Agrawal, Rakesh , and R. Srikant . "Fast Algorithms for Mining Association Rules in Large Databases." Proceedings of the 20th International Conference on Very Large Data Bases IEEE, 1994.
[3] Han, Jiawei , et al. "Mining Frequent Patterns without Candidate Generation: A Frequent- Pattern Tree Approach." Data Mining and Knowledge Discovery 8.1(2004):53-87.
[4] Pyun, Gwangbum, Unil Yun, and Keun Ho Ryu. "Efficient frequent pattern mining based on linear prefix tree." Knowledge-Based Systems 55 (2014): 125-139.
[5] Tsay, Yuh Jiuan, T. J. Hsu, and J. R. Yu. "FIUT: A new method for mining frequent itemsets." Information Sciences 179.11(2009):1724-1737.
[6] Lin, Ke Chung , I. E. Liao , and Z. S. Chen . "An improved frequent pattern growth method for mining association rules." Expert Systems with Applications 38.5(2011):5154-5161.
[7] Tseng, Fan Chen . "An adaptive approach to mining frequent itemsets efficiently." Expert Systems with Applications 39.18(2012):13166–13172.
[8] Zaki, Mohammed Javeed, et al. "New Algorithms for Fast Discovery of Association Rules." International Conference on Knowledge Discovery & Data Mining 1997.
[9] Aggarwal, Charu C. . Frequent Pattern Mining. Springer New York, 2014.
[10] Cheung, D. W. , et al. "Maintenance of discovered association rules in large databases: an incremental updating technique." Proceedings of the Twelfth International Conference on Data Engineering IEEE, 2002.
[11] Cheung, David W. , S. D. Lee , and B. Kao . " [WORLD SCIENTIFIC Proceedings of the Fifth International Conference on Database Systems for Advanced Applications - Melbourne, Australia (1 – 4 April 1997)] Database Systems for Advanced Applications "'97 - A General Incremental Technique for Maintaining Discovered Association Rules." :185-194.
[12] Ayan, Necip Fazil, A. U. Tansel, and E. Arkun. "An Efficient Algorithm To Update Large Itemsets With Early Pruning." Acm-sigkdd Intl Conf on Knowledge Discovery & Data Mining 1999.

[13] Srikant, Ramakrishnan, and R. Agrawal. "Mining Sequential Patterns: Generalizations and Performance Improvements." International Conference on Extending Database Technology 1996.

[14] Hong, Tzung Pei, C. Y. Wang, and Y. H. Tao. A new incremental data mining algorithm using pre-large itemsets. IOS Press, 2001.

[15] Koh, Jia Ling, and S. F. Shieh. "An Efficient Approach for Maintaining Association Rules Based on Adjusting FP-Tree Structures." Database Systems for Advances Applications, 9th International Conference, DASFAA 2004, Jeju Island, Korea, March 17-19, 2004, Proceedings DBLP, 2004.

[16] Cheung, William, and O. R. Zaane. "Incremental Mining of Frequent Patterns without Candidate Generation or Support Constrain." (2003).

[17] Leung, K. S., Q. I. Khan, and T. Hoque. "CanTree: A tree structure for efficient incremental mining of frequent patterns." Fifth IEEE International Conference on Data Mining (ICDM 2005) IEEE, 2005.

[18] Rathee, Sanjay, M. Kaul, and A. Kashyap. "R-Apriori: An Efficient Apriori based Algorithm on Spark." (2015).

[19] Farzanyar, Zahra, and N. Cercone. "[ACM Press the 2013 IEEE/ACM International Conference - Niagara, Ontario, Canada (2013.08.25-2013.08.28)] Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining - ASONAM \"13 - Efficient mining of frequent itemsets in social network data based on MapReduce framework." (2013):1183-1188.

[20] Luna, José Maria, et al. "Apriori Versions Based on MapReduce for Mining Frequent Patterns on Big Data." IEEE Transactions on Cybernetics PP.99(2017):1-15.

[21] Farzanyar, Zahra, and N. Cercone. "Accelerating Frequent Itemsets Mining on the Cloud: A MapReduce -Based Approach." IEEE International Conference on Data Mining Workshops 2014.

[22] Li, Ning, et al. "Parallel Implementation of Apriori Algorithm Based on MapReduce." Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on IEEE, 2012.

[23] Singh, Sudhakar, R. Garg, and P. K. Mishra. "Performance optimization of MapReduce-based Apriori algorithm on Hadoop cluster." Computers & Electrical Engineering (2017):S0045790617308534.

[24] Liu, Sheng Hui, et al. "IOMRA - A High Efficiency Frequent Itemset Mining Algorithm Based on the MapReduce Computation Model." 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE) IEEE, 2014.

[25] Djenouri, Youcef, et al. "Frequent Itemset Mining in Big Data with Effective Single Scan Algorithms." IEEE Access (2018):1-1.

[26] Li, Hao yuan, et al. "Pfp: parallel fp-growth for query recommendation." Proceedings of the 2008 ACM conference on Recom-mender systems ACM, 2008.

[27] Xun, Yang, J. Zhang, and X. Qin. "FiDoop: Parallel Mining of Frequent Itemsets Using MapReduce." IEEE Transactions on Systems Man & Cybernetics Systems 46.3(2017):313-325.

[28] Liang, Yen Hui, and S. Y. Wu. "Sequence-Growth: A Scalable and Effective Frequent Itemset Mining Algorithm for Big Data Based on MapReduce Framework." 2015 IEEE International Congress on Big Data (BigData Congress) IEEE, 2015.

[29] Thakare, Sanket, S. Rathi, and R. R. Sedamkar. "An Improved PrePost Algorithm for Frequent Pattern Mining with Hadoop on Cloud." Procedia Computer Science 79(2016):207-214.
[30] Moens, Sandy, E. Aksehirli, and B. Goethals. "[IEEE 2013 IEEE International Conference on Big Data - Silicon Valley, CA, USA (2013.10.6-2013.10.9)] 2013 IEEE International Conference on Big Data - Frequent Itemset Mining for Big Data." Big Data, 2013 IEEE International Conference on IEEE, 2013:111-118.

[31] Duong, Khanh Chuong, et al. "MapFIM: Memory Aware Parallelized Frequent Itemset Mining in Very Large Datasets." International Conference on Database and Expert Systems Applications - DEXA Springer, Cham, 2017.

[32] Apache, Welcome to Apache Hadoop!. Retrieved October 20, 2019, from http://hadoop.apache.org/

[33] Hoseini, Masome Sadat, M. N. Shahraki, and B. S. Neysiani. "A new algorithm for mining frequent patterns in Can Tree." International Conference on Knowledge-based Engineering & Innovation 2016.

[34] FIMI'03, Frequent Itemset Mining Dataset Repository. Retrieved October 20, 2019, from http://fimi.uantwerpen.be/data/