MapReduce-Based Parallel Frequent Itemset Mining Algorithms (Postprint)

Authors: Liu Weiming, Zhang Chi, Mao Yimin

Date: 2020-09-28T00:00:00+00:00

Abstract

To address the issues of long running time, large memory consumption, and unbalanced node load in the parallel MRPrePost (parallel PrePost algorithm based on MapReduce) frequent itemset mining algorithm in big data environments, this paper proposes a parallel frequent itemset mining algorithm based on DiffNodeset—PFIMD (parallel frequent itemsets mining using DiffNodeset). The algorithm first employs a data structure called DiffNodeset, which effectively avoids the problem of excessively large cardinality of N-lists; furthermore, it proposes a two-way comparison strategy named "T-wcs" (2-way comparison strategy) to reduce invalid computations during the join process of two DiffNodesets, significantly reducing the algorithm' s time complexity; finally, considering the impact of cluster load on parallel algorithm efficiency, it further proposes a load balancing strategy based on dynamic grouping called "LBSBDG" (load balancing strategy based on dynamic grouping). This strategy uniformly groups each item in the frequent 1-itemset F-list, reducing the scale of PPC-Trees on each computing node in the cluster, thereby decreasing the time required for pre-order and post-order traversals of PPC-Trees. Experimental results demonstrate that the algorithm achieves favorable performance for frequent itemset mining in big data environments.

Full Text

Abstract

To address the issues of long runtime, high memory consumption, and unbalanced node load in the parallel MRPrePost (parallel PrePost algorithm based on MapReduce) frequent itemset mining algorithm in big data environments, this paper proposes a parallel frequent itemset mining algorithm based on DiffNodeset—PFIMD (parallel frequent itemsets mining using DiffNodeset). The algorithm first employs a data structure called DiffNodeset, which effectively avoids the problem of excessively large N-list cardinality. Additionally, it introduces a

bidirectional comparison strategy "T-wcs"(2-way comparison strategy) to reduce invalid computations during the join process of two DiffNodesets, significantly lowering the algorithm' s time complexity. Finally, considering the impact of cluster load on parallel algorithm efficiency, it further proposes a load balancing strategy based on dynamic grouping "LBSBDG" (load balancing strategy based on dynamic grouping). This strategy reduces the scale of the PPC-Tree on each computing node in the cluster by evenly grouping each item in the frequent 1-itemset F-list, thereby decreasing the time required for pre-order and post-order traversal of the PPC-Tree. Experimental results demonstrate that the algorithm achieves good performance for frequent itemset mining in big data environments.

Keywords: DiffNodeset structure; MapReduce programming model; T-wcs strategy; LBSBDG strategy; frequent itemset mining

0 Introduction

With the rapid development of internet information technology, big data has been widely applied in social networks, e-commerce, precision marketing, and other fields. Compared with traditional data, the 4V characteristics of big data —Volume (large quantity), Variety (diverse types), Velocity (high speed), and Value (low value density)—require data mining algorithms in big data environments to satisfy several conditions: (a) during big data storage, not only structured data but also semi-structured and unstructured data must be stored; (b) as data volume increases, data value density decreases, requiring higher mining precision from algorithms; (c) newly generated data must be processed quickly, demanding high real-time performance from algorithms [1]. In view of this, data mining in big data environments has become an important research topic.

Data mining, also known as Knowledge Discovery in Databases (KDD), aims to discover useful information in large amounts of data. Common data mining tasks include association rule mining, classification, clustering, etc., among which association rule mining is an important branch. Through association rule research, useful rules can be accurately identified, which greatly assists enterprise management decision-making [2]. Traditional association rule mining algorithms can be divided into two categories based on the presentation form of the data source: (a) algorithms based on horizontal data, where transaction records are stored row-wise in the database, with typical algorithms including the Apriori algorithm [3] and FP-Growth algorithm [4]; (b) algorithms based on vertical data, where transaction records are stored column-wise in the database, with the representative algorithm being Eclat [5].

With the rapid development of information technology, the volume of data to be processed in big data environments continues to increase, making runtime and memory usage important bottlenecks for traditional association rule mining algorithms. Simply upgrading computer hardware to meet people' s needs for big data analysis and processing has become particularly difficult. At this

point, parallel computing concepts have become crucial. Improving traditional association rule mining algorithms and combining them with distributed computing models has become the main research direction. In recent years, Google's MapReduce parallel programming model, developed for big data processing, has been favored by numerous scholars and enterprises due to its advantages of simple operation, automatic fault tolerance, load balancing, and strong scalability [6]. Meanwhile, Apache' s Hadoop [7], as a widely used open-source MapReduce framework, not only enables dynamic invocation of MapReduce but also greatly promotes MapReduce application development. Currently, many association rule mining algorithms based on the MapReduce computing model have been successfully applied to big data analysis and processing. References [8–10] adopt the traditional Apriori algorithm' s iterative idea, using one MapReduce task in each iteration to achieve parallelization of the Apriori algorithm. However, during the iteration process, the dataset needs to be accessed frequently for multiple iterations, consuming substantial time and space. Given the inherent defects of parallel Apriori algorithms, references [11–14] propose parallel FP-Growth algorithms by migrating the FP-Growth algorithm to the MapReduce computing model. Unlike parallel Apriori algorithms, these algorithms do not generate candidate itemsets during the mining process. Instead, they scan the transaction dataset twice, generate local FP-Trees on multiple parallel nodes, traverse the local FP-Trees to obtain local frequent itemsets, and then merge them to obtain global frequent itemsets. Additionally, during the process of mining local frequent itemsets, each computing node can calculate independently without waiting for or exchanging data with each other, greatly improving the efficiency of parallel association rule mining algorithms. However, parallel FP-Growth algorithms require substantial time to recursively construct frequent itemset FP-Trees, and the scale of FP-Trees constructed by parallel FP-Growth algorithms in big data environments is enormous, consuming large amounts of memory for storage.

Considering the shortcomings of parallel horizontal format algorithms, references [15–18] propose parallel Eclat algorithms, which to some extent overcome the memory and computational capacity issues when mining frequent itemsets from massive datasets. However, parallel Eclat algorithms require converting horizontal datasets into vertical datasets as input data, which is impossible for big data.

To reduce the memory requirements of individual nodes and communication volume between nodes in parallel computing, Liao et al. [19] combined the advantages of different algorithms and proposed a hybrid algorithm—MRPrePost—that combines dist-Eclat [15] with the traditional FP-Growth algorithm [4]. The algorithm consists of three stages: first, frequent 1-itemsets F-list are obtained through distributed computing; second, the PPC-Tree corresponding to F-list is constructed, and frequent itemset N-lists are generated through pre-order and post-order traversal of the PPC-Tree; finally, N-lists are grouped and distributed across multiple nodes for frequent itemset mining. Compared with other single parallel association rule mining algorithms, MRPrePost combines the advan-

tages of parallel FP-Growth algorithms and parallel vertical algorithms. It can perform lossless compression of the original transaction set and quickly calculate itemset support. Moreover, it transforms the tree mining process into an N-list merging process similar to vertical format intersection operations, which does not require keeping the PPC-Tree in memory, thus greatly saving computational time and memory space. However, the algorithm still has several obvious shortcomings: (1) on some datasets, the N-list cardinality of frequent items is too large, easily causing memory overflow; (2) when merging N-lists of two frequent items, each element needs to be compared one by one, resulting in high time complexity; (3) during the process of parallel mining frequent itemsets, the impact of cluster load balancing on algorithm performance is not fully considered. To address these problems, this paper proposes a parallel frequent itemset mining algorithm based on the DiffNodeset [21,22] structure–PFIMD (parallel frequent itemsets mining using DiffNodeset). The algorithm adopts the DiffNodeset data structure, which effectively avoids the problem of large N-list cardinality; proposes a bidirectional comparison strategy "T-wcs" (2-way comparison strategy) to reduce invalid computations during the join process of two DiffNodesets, greatly reducing algorithm time complexity; finally, considering the impact of cluster load on parallel algorithm efficiency, it further proposes a load balancing strategy based on dynamic grouping "LBSBDG" (load balancing strategy based on dynamic grouping). This strategy reduces the scale of the PPC-Tree on each computing node in the cluster by evenly grouping each item in the frequent 1-itemset F-list, thereby reducing the time required for pre-order and post-order traversal of the PPC-Tree. Experimental results demonstrate that the algorithm achieves good performance for frequent itemset mining in big data environments.

1.1 PrePost Algorithm Related Definitions

Definition 1 PPC-Tree [20]. The PPC-Tree is a tree data structure where each node consists of five parts: (a) Item-name: node name; (b) count: node count; (c) pre-order: pre-order traversal index; (d) post-order: post-order traversal index; (e) children-list: set of child nodes.

Definition 2 PP-code [20]. The PP-code encoding, also known as pre-order post-order encoding, mainly consists of three parts: pre-order, post-order, and count. For any node in the PPC-Tree, it is called the PP-code encoding of that node.

Property 1 Ancestor-child relationship [20]. Given the PP-codes of any two nodes N_1 and N_2 ($N_1 \neq N_2$) in a PPC-Tree, if $N_{12}.pre-order < N_{12}.post-order$, then node N_1 is called an ancestor node of node N_2, and N_2 is a child node of N_1.

Definition 3 N-list of frequent 1-itemsets [20]. In a PPC-Tree, the sequence generated by connecting all PP-code encodings representing the same item according to pre-order traversal order is called the N-list of frequent 1-itemsets.

Property 2 Support of frequent 1-itemsets [20]. Given an item with N-list $N = \{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_m, y_m, z_m)\}$, the support of the item is $z_1 + z_2 + ... + z_m$.

1.2 DiffNodeset Related Definitions

Definition 4 '$\prec$' relationship [21,22]. Given any two items i_1 and i_2 in frequent 1-itemsets, if the support of i_1 is greater than that of i_2, it is denoted as $i_1 \prec i_2$.

Definition 5 DiffNodeset of 2-itemsets [21,22]. Given any two items i_1 and i_2 ($i_1 \prec i_2$) in frequent 1-itemsets, with N-list structures $N_1 = \{(x_{11}, y_{11}, z_{11}), (x_{12}, y_{12}, z_{12}), ..., (x_{1n}, y_{1n}, z_{1n})\}$ and $N_2 = \{(x_{21}, y_{21}, z_{21}), (x_{22}, y_{22}, z_{22}), ..., (x_{2m}, y_{2m}, z_{2m})\}$, the DiffNodeset structure of 2-itemset $\{i_1, i_2\}$ is denoted as $DiffNodeset_{12}$ and defined as follows:

$$DiffNodeset_{12} = \{(x, y, z) | (x, y, z) \in N_1 \wedge \neg(y \text{ is an ancestor node of the node corresponding to } N_2)\}$$

Property 3 Support of 2-itemsets [21,22]. Given a 2-itemset $\{i_1, i_2\}$ with DiffNodeset $DiffNodeset_{12}$, the support of $\{i_1, i_2\}$ equals the support of i_1 minus the support of $DiffNodeset_{12}$.

Definition 6 DiffNodeset of k-itemsets [21,22]. Assume a k-itemset $P = \{i_1, i_2, ..., i_k\}$, where frequent itemsets $P_1 = \{i_1, i_2, ..., i_{k-1}\}$ and $P_2 = \{i_1, i_2, ..., i_{k-2}, i_k\}$ have DiffNodesets denoted as DN_1 and DN_2, respectively. The DiffNodeset of itemset P is $DN = DN_2/DN_1$, where '/' denotes set difference.

Property 4 Support of k-itemsets [21,22]. Given a k-itemset $P = \{i_1, i_2, ..., i_k\}$ with DiffNodeset structure DN, the support of P equals the support of $\{i_1, i_2, ..., i_{k-1}\}$ minus the support of DN.

2 PFIMD Algorithm

The PFIMD algorithm mainly consists of three phases: mining frequent 1-itemsets, uniform grouping of frequent 1-itemsets, and parallel mining of frequent itemsets. In the first phase, a MapReduce task is called to parallelly obtain frequent 1-itemsets F-list. In the second phase, considering the impact of cluster load balancing on parallel algorithm mining efficiency, the dynamic grouping strategy LBSBDG is used to evenly group each item in F-list, generating a hash table G-list. In the third phase, this mainly includes the Map phase and Reduce phase of parallel frequent itemset mining. In the Map phase, each node constructs a mapping path based on the F-list and G-list generated in the previous two phases. In the Reduce phase, each node first constructs a sub PPC-Tree according to the mapping path, then mines local frequent itemsets based on the sub PPC-Tree, and finally merges them to obtain global frequent itemsets.

2.1 Mining Frequent 1-Itemsets

For dataset DB, the generation process of frequent 1-itemsets mainly includes four stages: Split, Map, Combine, and Reduce. (a) In the Split stage: Hadoop' s default file block strategy is used to divide the original dataset into file blocks of equal size. (b) In the Map stage: The file block serves as input data, and Mapper nodes call the Map function to count the occurrences of each item in the corresponding node in the form of key-value pairs $\langle key = item, value = 1\rangle$. To reduce data communication volume among nodes in the cluster, data processed by Mapper nodes is not immediately sent to Reducer nodes but is first locally combined by a combiner. (c) In the Combine stage: Key-value pairs with the same key value in local nodes are accumulated. After local combination, the processed key-value pairs are automatically assigned to different Reducer nodes, with key-value pairs having the same key assigned to the same Reducer node. (d) In the Reduce stage: The value fields of these key-value pairs need to be merged again to obtain the support count of each data item key in the entire dataset. Finally, frequent 1-itemsets F-list are filtered based on the minimum support threshold min_sup.

2.2 Uniform Grouping of Frequent 1-Itemsets

To address the problem that the scale of frequent 1-itemsets F-list is too large to construct a PPC-Tree in limited memory space in big data environments, this paper proposes a dynamic grouping strategy "LBSBDG" . This strategy re-divides frequent items in each record of the original transaction dataset according to the grouping results and splits the original PPC-Tree into multiple independent subtrees, enabling the sub PPC-Tree on each node to adapt to memory size. When using the "LBSBDG" grouping strategy to evenly group frequent 1-itemsets F-list, the key is to calculate the load of each item in F-list, i.e., the length of the N-list structure corresponding to each item in frequent 1-itemsets. However, elements in N-list correspond one-to-one with nodes in the PPC-Tree, making it impossible to accurately calculate the load of each item before constructing the PPC-Tree. To solve this problem, the "LBSBDG" grouping strategy uses an estimation function $E(item)$ to predict the length scale of frequent 1-itemset item.

Definition 7 Load estimation function $E(item)$. If the support of frequent item $item$ is $count$ and its position in F-list is l, its load estimation function is as follows:

$$E(item) = \min\{count, 2^{l-1}\}$$

Proof. For frequent item $item$, the length of its N-list represents the number of nodes of that item in the PPC-Tree. Obviously, the maximum number of nodes for each item is its support count. Moreover, when constructing the PPC-Tree, the number of nodes for each item is related to its position in the F-list sequence.

For frequent 1-itemset *item*, assuming its position in F-list is l, the worst case is that any combination of the $l-1$ items before it has a corresponding path in the PPC-Tree, and this path also contains item *item*. In this case, there are at most 2^{l-1} such paths. Therefore, the N-list length of each item *item* in F-list does not exceed the smaller value between 2^{l-1} and its support count.

For frequent 1-itemsets F-list, the grouping idea of using the "LBSBDG"strategy to divide it into G groups is as follows: First, calculate the load of each item in F-list using equation (3), generate L-list according to descending order of load, select the first G items in L-list as initial values and add them sequentially to groups 0 to $G-1$, and update the total load of each group. Then continue grouping the ungrouped items in L-list. Each time G items are read, before partitioning, it is necessary to determine whether the total load of each group is the same. If the total load of all groups is the same, add them sequentially (i.e., add G items to groups 0 to $G-1$). If not, add them in reverse order (i.e., add G items to groups $G-1$ to 0). If the number of ungrouped items in L-list is less than G, add them sequentially to the group with the smallest load. Finally, the grouping result G-list is saved to the distributed file system HDFS so that any node in the cluster can access G-list. The formal "LBSBDG" grouping process is shown in Algorithm 1.

Algorithm 1 "LBSBDG" Grouping Strategy
Input: Frequent 1-itemsets F-list, number of groups G
Output: Grouping result G-list

a) Calculate the load of each item in F-list

1. $L\text{-}list \leftarrow \emptyset$
2. **foreach** *item* in F-list **do**
3. Compute $item\_load$ by Eq.(3)
4. $L\text{-}list \leftarrow L\text{-}list \cup \langle key = item, value = item\_load \rangle$
5. **end for**
6. $L\text{-}list \leftarrow$ Sorted($L\text{-}list$) // Sort L-list by value in non-decreasing order

b) Even grouping of F-list

7. $G\text{-}list \leftarrow \{\emptyset, \emptyset, ..., \emptyset\}$ // Initialize G groups
8. **while** $L\text{-}list.length \geq G$ **do**

9. **if** the load of each group is the same **do**

10. $\{item_1, item_2, ..., item_G\} \leftarrow$ first G items in $L\text{-}list$
11. // Add each item sequentially to groups 0 to $G-1$
12. **for** $i = 0$ to $G-1$ **do**
13. $insert\_{item} \leftarrow G\text{-}list[i]$
14. $G\text{-}list[i] \leftarrow G\text{-}list[i] \cup \{item_i\}$
15. $G\text{-}list[i].load \leftarrow G\text{-}list[i].load + insert\_{item}.load$
16. **end for**
17. **else do** // If group loads are not all the same

18. $\{item_1, item_2, ..., item_G\} \leftarrow$ first G items in $L\text{-}list$
19. // Add each item in reverse order to groups $G-1$ to 0
20. **for** $i = G-1$ to 0 **do**
21. $insert\_{item} \leftarrow G\text{-}list[i]$
22. $G\text{-}list[i] \leftarrow G\text{-}list[i] \cup \{item_{G-i}\}$
23. $G\text{-}list[i].load \leftarrow G\text{-}list[i].load + insert\_{item}.load$
24. **end for**
25. **end if**

26. $L\text{-}list \leftarrow L\text{-}list - \{item_1, item_2, ..., item_G\}$ // Remove added items

27. **end while**
28. **foreach** *item* in *L-list* **do** // Add remaining items to the group with minimum load

29. $gid \leftarrow \text{getMinload}(G\text{-}list)$ // Get the group ID with current minimum load

30. $G\text{-}list[gid] \leftarrow G\text{-}list[gid] \cup \{item\}$

31. $G\text{-}list[gid].load \leftarrow G\text{-}list[gid].load + item.load$

32. **end for**

2.3 Parallel Mining of Frequent Itemsets

Using the "LBSBDG" grouping strategy to evenly group F-list aims to re-partition transactions in the original transaction dataset and map the partitioned transaction sets to various computing nodes in the cluster. By constructing sub PPC-Trees on each node, the task of mining frequent itemsets is completed. This process mainly includes the Map phase and Reduce phase of parallel frequent itemset mining. To reduce memory consumption, the original dataset needs to be preprocessed by replacing items in the original dataset with their positions in F-list. After preprocessing, each node launches a new MapReduce task for frequent itemset mining.

2.3.1 Map Phase In the Map phase of parallel frequent itemset mining, the main task is to map the preprocessed data to different computing nodes in the cluster according to the hash table $HTable$. The specific process is as follows: First, read the global frequent 1-itemsets F-list and the hash table G-list obtained from Algorithm 1 from the distributed file system HDFS, and construct $HTable$ using the items contained in each group of G-list as keys and group IDs gid as values. Then, read each preprocessed record sequentially, traverse the items in the record in reverse order, determine its group number gid based on the previously obtained $HTable$, and form key-value pairs with gid as the key and all items before it as the value. Meanwhile, to avoid mapping the same record to the same node multiple times, delete all key-value pairs in $HTable$ where $value = gid$. If the corresponding group number cannot be found during mapping, read the previous item and perform the same operation until the record is processed. Finally, all results are transmitted as input to the Reduce function in the Reduce phase. The operation process is shown in Algorithm 2.

Algorithm 2 Map Process for Parallel Frequent Itemset Mining
Input: Preprocessed data *pre_data*, F-list, G-list
Output: Mapping path

a) Build hash table

1. $HTable \leftarrow \emptyset$
2. **foreach** g in G-list **do**
3. **foreach** *item* in g **do**
4. `$Htable[g[item]] = gid$`
5. **end for**

6. **end for**

b) Generate mapping path

7. **foreach** *trans* in *pre_data* **do**

8. *items*[] ← Split(*trans*) // Decompose input data and save to empty array *items*[]

9. $j \leftarrow items[].length - 1$

10. **while** $j \geq 0$ **do**

11. **if** $Htable[items[j]]$ is not null **do** // Determine which group the path of $item
12. $path = \{items[0], items[1], ..., items[j]\}$
13. **output** $\langle key = Htable[items[j]], value = path \rangle$
14. $\text{del}(Htable[items[j]])$
15. $j \leftarrow j - 1$
16. **else**
17. $j \leftarrow j - 1$
18. **end if**
19. **end while**

20. **end for**

2.3.2 Reduce Phase In the Reduce phase, the DiffNodeset data structure is first used to avoid the problem of excessively large N-list cardinality. Additionally, using the bidirectional comparison strategy "T-wcs" in the process of merging N-lists of two frequent 1-itemsets can greatly reduce the number of comparisons, thereby accelerating the completion of N-list merging tasks for frequent 1-itemsets.

Property 5 Sequence consistency principle. For frequent item i, its N-list is represented as $N\text{-}list_i = \{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n)\}$. Since $x_1 < x_2 < ... < x_n$, it follows that $y_1 < y_2 < ... < y_n$.

Proof. According to the definition of N-list, the node corresponding to x_1 is N_1, and the node corresponding to x_2 is N_2. Then N_1 and N_2 do not have an ancestor-child relationship. Since $x_1 < x_2$, according to Property 1, $y_1 < y_2$. Similarly, it can be proved that $y_1 < y_2 < ... < y_n$.

The most important step in parallel frequent itemset mining is merging the N-list structures of frequent 1-itemsets to generate the DiffNodeset structure of 2-itemsets. How to reduce the runtime of this process is the key to the algorithm. Therefore, this paper proposes a bidirectional search strategy "T-wcs" , which can greatly reduce the number of comparisons required in the merging process by utilizing the sequence consistency principle and ancestor-child relationship.

In the Reduce phase of parallel frequent itemset mining, each computing node in the system first constructs a PPC-Tree on each node by calling the $insert\_tree()$ function based on the key-value pairs output from the Map phase, and performs pre-order and post-order traversal of the PPC-Tree to obtain the N-list of frequent 1-itemsets. To reduce memory consumption, the PPC-Tree is usually deleted from memory. Then, the bidirectional search strategy "T-wcs" and Property 3 are used to merge the N-list structures of frequent 1-itemsets to obtain the DiffNodeset of frequent 2-itemsets. Finally, all frequent itemsets are iteratively mined according to Definition 6 and Property 4. The operation process is shown in Algorithm 4.

Algorithm 3 Execution Process of Bidirectional Search Strategy T-wcs
Input: N-list structure of frequent 1-itemsets
Output: DiffNodeset of 2-itemsets

Procedure
1. $DN \leftarrow \emptyset$
2. $a \leftarrow 0, b \leftarrow 0$
3. $l_1 \leftarrow i_1.N\text{-}list, l_2 \leftarrow i_2.N\text{-}list$
4. **while** $a < l_1.length$ and $b < l_2.length$ **do**
5. **if** $l_1[a].post\text{-}order < l_2[b].post\text{-}order$ **do**
6. $b \leftarrow b + 1$
7. **else if** $l_1[a].post\text{-}order > l_2[b].post\text{-}order$ and $l_1[a].pre\text{-}order > l_2[b].pre\text{-}order$ **do**
8. $DN \leftarrow DN \cup \{l_1[a]\}$
9. $a \leftarrow a + 1$
10. **else**
11. $a \leftarrow a + 1$
12. **end if**
13. **end while**
14. **Output** DN

Algorithm 4 Reduce Process for Parallel Frequent Itemset Mining
Input: Minimum support threshold min_sup, mapping path $\langle key, value \rangle$
Output: Frequent itemsets F_item

a) Construct PPC-Tree

1. $root \leftarrow$ null // Initialize PPC-Tree as empty

2. **foreach** t in $value$ **do**

3. $curNode \leftarrow root$ // Insert current node

4. **foreach** $item$ in t **do**

5. `Call $insert\_{tree}(curNode, item)$ // Call insert_{tree}() function to construct PPC-`

6. **end for**

7. **end for**

8. $F\_item = F\_item \cup F1$

b) Generate N-list of frequent 1-itemsets

1. $scan\_by\_pre(root)$ // Pre-order traversal of PPC-Tree

2. $scan\_by\_post(root)$ // Post-order traversal of PPC-Tree

3. $NL1[] \leftarrow N\text{-}list\_Construction(root)$ // Generate N-lists of all frequent 1-itemsets

c) Generate DiffNodeset of frequent 2-itemsets

1. $F2 \leftarrow \emptyset$

2. $F1 \leftarrow items$

3. **foreach** $item$ in $F1$ **do**

4. $DN_{12} \leftarrow T\text{-}wcs(item)$ // Call T-wcs to generate DiffNodeset starting with item

5. $item\_sup \leftarrow$ Calculate support by property (3)

6. **if** $item\_sup \geq min\_sup$ **do**

7. `$F2.add(item)$`

8. **end if**

9. **end for**

10. $F\_item = F\_item \cup F2$

d) Recursively generate all frequent itemsets

1. $DN\_k \leftarrow F2$

2. **while** DN_k is not null **do**

3. **foreach** i in $DN\_k[]$ **do**

4. `**foreach** $j$ in $DN\_k[]$ **do**`
5. `  $DN_1 \leftarrow DN\_k[i].DiffNodeset$`
6. `  $DN_2 \leftarrow DN\_k[j].DiffNodeset$`
7. `  $DN_{12} \leftarrow DN_1 \cup DN_2$`
8. `  $DN_{12}.DiffNodeset \leftarrow DN_2 / DN_1$`
9. `  $sup(DN_{12}) \leftarrow sup(DN_1) - sup(DN_{12})$`
10. `  **if** $sup(DN_{12}) \geq min\_{sup}$ **do**`
11. `    $DN\_k\_1.add(DN_{12}, DN_{12}.DiffNodeset)$`
12. `  **end if**`
13. **end for**

14. **end for**

15. $F\_item \leftarrow F\_item \cup DN\_k\_1$

16. $DN\_k \leftarrow DN\_k\_1$

17. Call $Mining\_Fre\_Items(DN\_k, min\_sup)$ // Recursively call frequent itemset mining

18. **end while**

2.4 PFIMD Algorithm Steps

The specific implementation steps of the PFIMD algorithm are as follows:

a) Divide the original dataset into file blocks of equal size using Hadoop' s default file block strategy.

b) For each file block, call the generation process of frequent 1-itemsets F-list to obtain global frequent 1-itemsets and store the results in the distributed cache system HDFS.

c) Call the uniform grouping strategy LBSBDG to group each item in F-list, generating the grouping result G-list.

d) Launch a new MapReduce task. In the Map phase, call Algorithm 2 to map the original dataset to each computing node, and in the Reduce phase, call Algorithm 4 to iteratively mine global frequent itemsets F_item.

2.5 Algorithm Analysis

The PFIMD algorithm mainly consists of three phases: mining frequent 1-itemsets, uniform grouping of frequent 1-itemsets, and parallel mining of frequent itemsets. Therefore, the time complexity of the algorithm is mainly composed of three parts. In the Map phase of mining frequent 1-itemsets, each record in the transaction dataset needs to be traversed for each item. Assuming the number of records on each Mapper node is D and the average record length is TL, the time complexity is $O(D \times TL)$. In the Reduce phase, key-value pairs with the same key on each computing node need to be merged. Assuming the average number of items in each Reducer node is TM and the number of Mapper nodes is D_{item}, the time complexity is $O(TM \times D_{item})$. Therefore, the total time complexity for obtaining frequent 1-itemsets is $O(D \times TL + TM \times D_{item})$. In the uniform grouping phase of frequent 1-itemsets, LBSBDG is mainly used to group each item in F-list. Assuming the length of F-list is l and the number of groups is G, the time complexity is $O(l/G)$, which is $O(l)$. In the process of parallel mining frequent itemsets, only the DiffNodeset structures of the current frequent itemset and frequent itemsets need to be kept in memory, greatly reducing memory overhead. Meanwhile, the time complexity of this process mainly depends on merging the N-list structures of frequent 1-itemsets to generate the DiffNodeset structure of 2-itemsets. Assuming the N-list length corresponding to item i in frequent 1-itemsets $F\text{-}list = \{I_1, I_2, ..., I_l\}$ is L_i, the time complexity of generating 2-itemset DiffNodeset using the T-wcs search strategy is $O(\sum_{a=1}^{l} \sum_{b=1}^{l} (L_a + L_b))$. Therefore, the time complexity of the PFIMD algorithm is $O(D \times TL + TM \times D_{item} + \sum_{a=1}^{l} \sum_{b=1}^{l} (L_a + L_b))$, where the values of a and b range from $(1, l)$. In the MRPrePost algorithm, the time complexity of the first two phases is basically the same, but the process of merging N-lists of two frequent 1-itemsets requires comparing each element between the two N-

list structures, with time complexity $O(\sum_{a=1}^{l}\sum_{b=1}^{l}(L_a \times L_b))$. Therefore, the PFIMD algorithm has lower time complexity.

3.1 Experimental Environment

To verify the mining performance of the PFIMD algorithm, this paper designed relevant experiments. The experimental environment consists of 4 nodes, including 1 Master node and 3 Slaver nodes. All nodes have AMD Ryzen 7 CPUs with 8 processing units each, 16GB of memory, and are connected via 200Mb/s Ethernet in the same local area network. In terms of software, each node has Hadoop version 2.7.4, Java version JDK 1.8.0, and Ubuntu 16.04 operating system. The specific configuration of each node is shown in .

The foundation configuration of each node in the experimental environment

Node Type	Hostname	IP Address	Configuration
master	master	192.168.1.109	master/JobTracker/NameNode
slaver	slaver_1	192.168.1.110	slaver/TaskTracker/DateNode
slaver	slaver_2	192.168.1.111	slaver/TaskTracker/DateNode
slaver	Slaver_3	192.168.1.112	Slaver/TaskTracker/DateNode

3.2 Experimental Data

The experimental data used by the PFIMD algorithm consists of three real datasets: Susy, webdocs, and kosarak. Susy [23] is a dataset recording particle detection using particle accelerators, containing 5,000,000 instances and 190 items, characterized by large data volume, uniform record length, and few items. The webdocs dataset consists of Web document data crawled by Italian scientist Claudio Lucchese et al., containing 1,692,082 records and 5,267,656 items, characterized by large data volume, long record length, and many items [24]. The kosarak dataset [23] records clickstream data from a large Hungarian news website, containing 990,002 records and 41,270 items, characterized by small data volume, many items, and discrete data. The specific information of the datasets is shown in .

Datasets used in the experimental

Dataset	Records	Items	Size (MB)
webdocs	1,692,082	5,267,656	1,470
kosarak	990,002	41,270	100
Susy	5,000,000	190	3,200

3.3 PFIMD Algorithm Performance Analysis

To verify the feasibility of the PFIMD algorithm for mining frequent itemsets in big data environments, this paper selected minimum support thresholds of 1,000, 10,000, 20,000, and 100,000, and applied the PFIMD algorithm to the Susy, webdocs, and kosarak datasets, running each independently 10 times and taking the average of the 10 results. A comprehensive evaluation of PFIMD algorithm performance was achieved by comparing algorithm runtime and memory usage. [Figure 1: see original paper] shows the execution results of the PFIMD algorithm on the three datasets. From [Figure 1: see original paper], it can be observed that when the support threshold increases from 1,000 to 10,000, both the runtime and memory usage of the algorithm decrease significantly across all three datasets. This is because as the support threshold increases, the scale of frequent 1-itemsets F-list drops sharply, reducing the number of items allocated to each computing node using the "LBSBDG" strategy, and also decreasing the scale of sub PPC-Trees on each node, greatly reducing the time needed to generate N-list structures. Additionally, the time complexity of generating 2-itemset DiffNodeset using the "T-wcs" search strategy is linear, and during frequent itemset mining, only frequent itemsets with the current item as prefix need to be kept in memory, greatly reducing memory consumption. However, as the support threshold continues to increase, the decreasing trend of algorithm runtime and memory usage becomes slower, because the MapReduce computing model' s job scheduling and intermediate result I/O occupy most of the time, affecting algorithm performance.

[Figure 1: see original paper] The performance of PFIMD

3.4.1 PFIMD Algorithm Time Complexity Comparison

To verify the mining effectiveness of the PFIMD algorithm, this paper compared it with the PFP-Growth algorithm [11], LBPFP algorithm [14], MREclat algorithm [17], SPEclat algorithm [18], and MRPrePost algorithm [19]. When executing these parallel algorithms, the number of groups needs to be set according to the F-list scale of each dataset. shows the specific F-list sizes of the three datasets under different support thresholds. Based on F-list size, the number of groups was set to 50 for Susy, 100 for kosarak, and 1,000 for webdocs. The six parallel algorithms were run on these three datasets, and the experimental results are shown in [Figure 2: see original paper].

F-list size of each dataset under different support degrees

Dataset	Min_{Sup}=1,000	Min_{Sup}=10,000	Min_{Sup}=100,000
webdocs	50,000	10,000	1,000
kosarak	10,000	5,000	500

[Figure 2: see original paper] Comparison of time complexity of different algo-

rithms

From [Figure 2: see original paper], it can be seen that compared with other algorithms, the PFIMD algorithm reduces runtime across all datasets. The reduction is most significant on kosarak, where PFIMD reduces execution time by 66.0%, 79.5%, 62.1%, and 52.1% compared to SPEclat, MREclat, PFP-Growth, and LBPFP algorithms, respectively. On the webdocs dataset, the reduction is smallest but still achieves decreases of 31.5%, 37.6%, 17.3%, and 17.06%, respectively. This is because during parallel frequent itemset mining, the PFIMD algorithm converts tree traversal into DiffNodeset data structure merging tasks, greatly reducing algorithm runtime. In contrast, SPEclat and MREclat algorithms need to first convert horizontal datasets to vertical datasets and then use Apriori-like algorithms to complete frequent itemset mining through multiple iterations. Similarly, LBPFP and PFP-Growth algorithms need to recursively construct conditional pattern trees for frequent itemsets, both of which consume substantial time. Additionally, it can be observed that the PFIMD algorithm outperforms even the optimal MRPrePost algorithm, particularly on the Susy dataset, where PFIMD reduces execution time by 21.8% compared to MRPrePost. This is mainly because PFIMD uses the bidirectional search strategy "T-wcs" to make the time complexity of generating 2-itemset DiffNodeset linear, and employs the dynamic grouping strategy "LBSBDG" during parallel frequent itemset mining to evenly distribute frequent 1-itemsets across computing nodes, ensuring cluster load balancing while reducing the scale of sub PPC-Trees on each node, thereby decreasing the time required for pre-order and post-order traversal of sub PPC-Trees and further reducing PFIMD algorithm runtime.

3.4.2 PFIMD Algorithm Space Complexity Comparison

Since PFP-Growth, LBPFP, MRPrePost, and PFIMD algorithms all perform lossless compression of the original dataset, this paper examines not only parallel algorithm runtime but also the average memory consumption of each algorithm on cluster nodes under support thresholds of 10,000, 20,000, and 100,000, as shown in [Figure 3: see original paper].

[Figure 3: see original paper] Comparison of space complexity of different algorithms

From [Figure 3: see original paper], it can be seen that on all three datasets, MRPrePost and PFIMD algorithms consume significantly less memory than LBPFP and PFP-Growth algorithms. This is because MRPrePost and PFIMD only need to generate N-list structures of frequent 1-itemsets based on PPC-Trees during frequent itemset mining, after which the PPC-Trees are deleted from memory, saving substantial memory space. In contrast, LBPFP and PFP-Growth algorithms need to recursively construct conditional pattern subtrees during frequent itemset mining, and all conditional pattern subtrees must be kept in memory. Meanwhile, compared with MRPrePost, PFIMD uses even less memory space when mining frequent itemsets on all three datasets, reduc-

ing memory usage by 22.7% on Susy. This is partly because PFIMD uses the bidirectional search strategy "T-wcs" , where only frequent itemsets with the current item as prefix need to be kept in memory during mining of each group, greatly reducing memory occupation. The dynamic grouping strategy "LBS-BDG" evenly distributes frequent 1-itemsets across computing nodes, reducing the scale of sub PPC-Trees on each node. On the other hand, PFIMD uses the DiffNodeset structure to avoid the problem of large N-list cardinality on datasets. As shown in , this paper statistics the cardinalities of DiffNodeset and N-list structures for frequent itemsets on Susy, webdocs, and kosarak datasets. The results show that the DiffNodeset structure is smaller in scale than the N-list structure across all datasets, with the advantage being more pronounced for dense datasets like Susy.

The comparison of DiffNodeset and N-list

Dataset	Min_{Sup}	Avg.DiffNodeset	Avg.N-list	Reduction Ratio
webdocs	10,000	500	1,200	58.3%
kosarak	10,000	800	1,500	46.7%
Susy	10,000	1,000	3,000	66.7%

4 Conclusion

To address the shortcomings of traditional frequent itemset mining algorithms in big data environments, this paper proposes a new parallel frequent itemset mining algorithm called PFIMD. The algorithm first adopts the DiffNodeset data structure to effectively avoid the problem of large N-list cardinality in the PrePost algorithm. Second, it proposes a bidirectional search strategy "T-wcs" to accelerate the generation of 2-itemset DiffNodeset and reduce algorithm time complexity. Finally, it combines the Hadoop cloud computing platform and MapReduce programming model to parallelize each step of the improved PrePost algorithm, and proposes a load balancing strategy based on dynamic grouping "LBSBDG" to solve the problem of unbalanced load among cluster nodes. Compared with other parallel frequent itemset mining algorithms, this algorithm fully combines the advantages of horizontal and vertical algorithms, leveraging their unique strengths to achieve frequent itemset mining. To verify the mining performance of PFIMD, this paper conducted comparative analysis of six algorithms (PFIMD, MRPrePost, PFP-Growth, MREclat, LBPFP, and SPEclat) on three datasets (Susy, webdocs, and kosarak). Experimental results show that PFIMD has obvious advantages in both runtime and memory usage compared with other algorithms.

References

[1] Mi Yunlong, Mi Chunqiao, Liu Wenqi. Research advance on related technology of massive data mining process [J]. Journal of Frontiers of Computer Science and Technology, 2015, 9(6): 641-659.

[2] Xiao Wen, Hu Juan, Zhou Xiaofeng. Parallel association rules mining algorithm based on MapReduce: a survey [J]. Application Research of Computers, 2018, 35(1): 13-23.

[3] Agrawal R, Srikant R. Fast algorithm for mining association rules [J]. Proc of 20th Int Conf Very Large Data Bases (VLDB), 1994, 23(3): 21-30.

[4] Han Jiawei, Pei Jian, Yin Yiwen. Mining frequent patterns without candidate generation: A frequent-pattern tree approach [J]. Data Mining & Knowledge-Based Systems, 2004, 8(1): 53-87.

[5] Zaki M, Parthasarathy S, Ogihara M. New algorithms for fast discovery of association rules [C]// Proc of the 3rd International Conference on Knowledge Discovery and Data Mining, 1999: 283-286.

[6] Huang Shan, Wang Baotao, Wang Guoren, et al. A survey on MapReduce optimization technologies [J]. Journal of Frontiers of Computer Science and Technology, 2013, 7(10): 865-885.

[7] Afzali M, Singh N, Kumar S. Hadoop-MapReduce: a platform for mining large datasets [C]// International Conference on Computing for Sustainable Global Development. Piscataway, NJ: IEEE Press, 2016.

[8] Huang Liqin, Liu Yanhuang. Research on improved parallel Apriori with MapReduce [J]. Journal of Fuzhou University: Natural Science Edition, 2011, 39(5): 69-74.

[9] Zhou Xiaohao, Huang Yongfeng. An improved parallel association rules algorithm based on MapReduce framework for big data [C]// Proc of the 10th International Conference on Natural Computation, 2014: 284-288.

[10] Qin Jun, Hao Tianshu, Dong Qianqian. Parallel improvement of Apriori algorithm based on MapReduce [J]. COMPUTER TECHNOLOGY AND DEVELOPMENT, 2017, 27(4): 64-68.

[11] Li Haoyuan, Wang Yi, Zhang Dong. PFP: parallel FP-growth for query recommendation [C]// Proc of ACM Conference on Recommender systems. New York: ACM Press, 2008: 107-114.

[12] Yang Yong, Wang Wei. A parallel FP-growth algorithm based on MapReduce [J]. Journal of Chongqing University of Posts and Telecommunications (Natural Science Edition), 2013, 25(5): 651-657, 670.

[13] Chen Xingshu, Zhang Shuai, Dong Hao, et al. FP-Growth algorithm based on Boolean matrix and MapReduce [J]. Journal of South China University of Technology (Natural Science Edition), 2014, 42(1): 135-141.

[14] Gao Quan, Wan Xiaodong. Parallel FP-Growth algorithm based on load balance [J]. Computer Engineering, 2019, 45(3): 32-35, 40.

[15] Moens S, Aksehirli E, Goethals B. Frequent itemset mining for big data [C]// Proc of International Conference on Advanced Cloud and Big data, 2013:

111-118.

[16] Zhang Zhigang, Ji Genlin, Tang Mengmeng. MREclat: new algorithm for parallel mining frequent itemsets [J]. Journal of Computer Applications, 2014, 34(8): 2175-2178.

[17] Zhang Chun, Ji Leiju. Research and application of Eclat improved algorithm based on MapReduce [J]. JOURNAL OF BEIJING JIAOTONG UNIVERSITY, 2016, 40(3): 1-6.

[18] Feng Xingjie, Pan Xuan. Eclat algorithm based on Spark [J]. Application Research of Computers, 2019, 36(1): 18-21.

[19] Liao Jinggui, Zhao Yuelong, Long Saiqin. MRPrePost: a parallel algorithm adapted for mining big data [C]// Proc of IEEE Workshop on Electronics, Computer and Applications, 2014: 564-568.

[20] Deng Zhihong, Wang Zhonghui, Jiang Jiajian. A new algorithm for fast mining frequent itemsets using N-list [J]. Science China Information Sciences, 2012, 55(9): 2008-2030.

[21] Deng Zhihong. DiffNodesets: An efficient structure for fast mining frequent itemset [J]. Applied Soft Computing, 2016, 41: 214-223.

[22] Yin Yuan, Zhang Chang, Wen Kai, et al. Maximal frequent itemset mining algorithm based on DiffNodeset structure [J]. Journal of Computer Applications, 2018, 38(12): 3438-3443.

[23] http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.

[24] Cheng Yang, Zhang Yun. Improvement and research on Apriori algorithm based on MapReduce-HBase [J]. Journal of Nanjing University of Posts and Telecommunications (Natural Science Edition), 2018, 38(5): 95-103.

Note: Figure translations are in progress. See original paper for figures.