Closed Frequent Itemset Mining over Fast Data Stream Based on Hadoop

Shan Jicheng, Liu Qingbao
Science and Technology on Information Systems Engineering Laboratory
National University of Defense Technology
Changsha, China
email: sjcheng2007@126.com, liuqingbao@nudt.edu.cn

***Abstract*—Mining closed frequent itemsets provides complete and condensed information for non-redundant association rules generation. Online mining of closed frequent itemsets over streaming data is one of the most important issues in mining data streams. In this paper, we extend two types of methods to MapReduce platform to mine closed frequent itemset over fast data streams. Experiments show that both methods have performance improvement with more mapper nodes and the vertical format data method has higher speed to process fast data streams.**

Keywords- data stream; closed frequent itemsets; mapreduce.

I. INTRODUCTION

Frequent itemset mining has been an important research issue for many years in data mining community. With the development of data storage and data processing, frequent itemset mining meets new challenges and needs to be extended. For example, Wireless Sensor Network (WSN) can be used to monitor the traffic status and the environment information. With time flows, the WSN will produce a large scale of data that cannot be storaged in traditional static database. WSN data related to time should be processed as stream data with special methods. However, most of the data stream mining methods face the performance problem as they are often used on one computer which has poor computing ability. When the stream becomes 'bigger' and 'faster', these methods have lower effect or even cannot work.

For mining frequent itemsets in traditional transactional database, Apriori is the most classic and most widely used algorithm proposed by R. Agrawal and R. Srikant in 1994 [1]. The algorithm works in a multi-phase generation-and-test framework, including the joining and pruning process to reduce the number of candidates before scanning the database for frequency computing. The algorithm terminates when no more candidate itemsets can be generated. The Apriori levelwise approach implies several scans over the database for support counting of candidate itemsets which affects the performance of the algorithm. To reduce the scan overhead, some depth-first methods were proposed, of which the Eclat (Equivalent CLASS Transformation) algorithm by Zaki [2] and the FP-Growth algorithm by Han, Pei, Yin, and Mao [3] are typical representatives. These algorithms use compressed data structure to store necessary transaction information and avoid candidate generation and levelwise scans.

Recently, the increasing emergence of data streams has led to the study of online mining of frequent itemsets, which is an important technique for a wide range of emerging applications [4], such as web search and click-stream mining, trend analysis and fraud detection in telecommunications data, e-business and stock market analysis, and wireless sensor networks. Unlike mining static databases, mining data streams poses many new challenges. Firstly, it is not realistic to store the whole data stream in the main memory or even the secondary storage space as the data continuously come with no boundary. Secondly, traditional methods working on static stored datasets by multiple scans are unrealistic, since the streaming data is passed only once. Thirdly, stream mining requires highly efficient real-time processing in order to keep up with the high data arrival rate and mining results are expected to be available within short response time. In addition, the combinatorial explosion of itemsets exacerbates mining frequent itemsets over streams in terms of both memory consumption and time expense. In the past ten years, many algorithms to mine frequent itemsets over data stream have been proposed, like Lossy Counting [5], DSM-FI [6], FDPM [7] , estDec [8], FP-streaming [9], estWin [10], Moment [11], etc. These algorithms can be divided into two categories based on the window they adopt: the landmark window model and the sliding window model.

With the advent of Internet and the exponential growth of data volume towards a terabyte or more, it has been more difficult to mine them on a single sequential machine. Researchers attempt to parallelize these frequent itemset mining algorithms to speed up the mining of the ever-increasing sized databases. In big data era, we need new framework and new methods to capture and deal with dynamic changing, high dimensional, large scale data. In 2004, Google proposed their Google File System [12] and MapReduce [13] framework which has been successfully used in Google search and other Google products. With some number of ordinary computers, Google Distributed File System solved the big data storage problem and MapReduce framework can be used to do computing work on the big data stored. In a MapReduce cluster, a node which schedules tasks execution among nodes is called the master, and other nodes are workers. MapReduce uses two phase procedure to implement Function Programming, map and reduce. The master is responsible for the scheduling of the map tasks and

 ISBN: 978-1-61208-415-2

the reduce tasks which are executed by the workers after the job is initialized. In Map phase, the map function in each node takes the input data as <key, value> pair and outputs a list of <key, value> pairs in different domain. Then in Reduce phase, the reduce function in nodes takes the output of map functions as <key, list-of-values> and outputs a collection of values as the result. Also, the output of the reduce function can be formatted as <key, value> pairs which makes multiphase mapreduce iteration possible. What's more important, both the map and reduce functions can be performed in parallel.

MapReduce hides the problems like fault tolerance, data distribution and load balancing in parallelization, which allows user to focus on the computing implementation problem without worrying about the parallelization details. Developers only need to write the map function to read blocks from the distributed file system and produce a set of intermediate <key, value> pairs. The MapReduce framework organizes together all intermediate values related to the same intermediate key , often with a shuffle procedure, and sends them to the reduce function [13]. The reduce function, also written by the user, captures an intermediate key and a set of values for that key. Then reduce function merges together these values to produce an aggregate result. This merging allows users to handle lists of values that are too large to fit in memory. Thus, MapReduce can be an efficient platform for mining frequent itemsets from huge datasets of tera- or peta-bytes [14][15][16][17][18].

In this paper, we consider to mining closed frequent itemsets over data stream with sliding window model based on the MapReduce framework. Closed frequent itemsets can store necessary information to get complete frequent itemsets with less storage requirement [19]. Sliding window model pays different attention to data produced at different time so that it can discover time-related rules which are more important in stream application environment. Based on the MapReduce framework, our method has higher performance and ability to process high-velocity large-volume dynamic-variety stream data.

The rest of this paper is organized as follows. The preliminary knowledge is given in Section II. Section III describes details of the two methods we extend and implement on MapReduce platform. Experiment results are shown and analyzed in Section IV. We conclude in Section V.

II. PRELIMINARIES

Let $A = \{a_1, \ldots, a_m\}$ be a set of **items**. Items may be commodities, products, records, internet links etc. Any subset $I \subseteq A$ is called an itemset. Let $T = (t_1, \ldots, t_n)$ be a set of **transactions** within a slide window of size *n* denoted by data stream. Each unique transaction t_i of *T* is a pair <tid_i, $k\text{-}items_i$> of which $k\text{-}items_i \subseteq A$ is a set of *k* items. A transaction database can list, for example, the sets of products bought by the customers of a supermarket within a period of time, or the sets of pages a user visited for a site in a session. Every transaction refers to an itemset, but some itemsets may not appear in *T*.

Let $I \subseteq A$ be an itemset and *T* a transaction database over *A*. A transaction $t \subseteq T$ **covers** the itemset *I* or the itemset is **contained in** transaction t if and only if $I \subseteq t$.

The set $K_T(I) = \{k \in \{1, \ldots, n\} | I \subseteq t_k\}$ is called the **cover** of *I* w.r.t. *T*. The cover of an itemset is the index set of transactions that cover it.

The value $s_T(I) = |K_T(I)|$ is called the **absolute support** of *I* with respect to *T*. The value of $\sigma_T(I) = \frac{1}{n}|K_T(I)|$ is called the **relative support** of *I* with respect to *T*. The support of *I* is the number or fraction of transactions that cover it. Sometimes $\sigma_T(I)$ is also called the (**relative**) **frequency** of *I* in *T*.

The Frequent Itemset Mining problem can be formally defined as:

- Given:
  - a set $A = \{a_1, \ldots, a_m\}$ of items;
  - a vector $T = (t_1, \ldots, t_n)$ of transactions over *A*;
  - a number σ_{min} such that $0 < \sigma_{min} < 1$, the **minimum support**.
- Goal:
  - the set of frequent itemsets, that is, the set $\{I \subseteq A \mid \sigma_T \geq \sigma_{min}\}$.

As shown in Figure 1, all the frequent k-itemsets (k=1,2,3) for the transaction database *T* left with 10 transactions are listed right given the minimum support s_{min}= 3. So the frequent itemset for *T* is

$\mathcal{F} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{a,c\}, \{a,d\}, \{a,e\},$
$\{b,c\}, \{c,d\}, \{c,e\}, \{d,e\}, \{a,c,d\}, \{a,c,e\}, \{a,d,e\}\}$

According to the priori property, every subset of a frequent itemset is also frequent. Thus, generation-and-test algorithms to mine all frequent itemsets (complete frequent itemsets) suffer from the problem of combinatorial explosion. To solve this problem two substitute solutions have been proposed. In the first solution, only maximal frequent itemsets are mined. A frequent itemset is maximal if none of its superset is frequent. The number of maximal frequent itemsets $\mathcal{M}$ is usually smaller than the number of complete frequent itemsets $\mathcal{F}$, and we can derive all the members of $\mathcal{F}$ from $\mathcal{M}$. It is a pity that $\mathcal{M}$ does not contain

TID	Itemset
1	{b,c,d}
2	{a,d,e}
3	{a,c,d,e}
4	{a,c,e}
5	{a,c,d}
6	{a,e}
7	{a,c,d,e}
8	{b,c}
9	{a,d,e}
10	{b,c,e}

1 item	2 items	3 items
{a}:7	{a,c}:4	{a,c,d}:3
{b}:3	{a,d}:5	{a,c,e}:3
{c}:7	{a,e}:6	{a,d,e}:4
{d}:6	{b,c}:3	
{e}:7	{c,d}:4	
	{c,e}:4	
	{d,e}:4	

Figure 1. A transaction database, with 10 transactions, and the enumeration of all possible frequent itemsets using the minimum support of s_{min}= 3 or σ_{min} = 0.3 = 30%.

support information of itemsets that do not belong to $\mathcal{M}$. Thus, discovering only maximal frequent itemset loses information.

The second solution maintains enough information to get complete frequent itemsets. It discovers all closed frequent itemsets from the database. An itemset is closed if and only if none of its superset has the same support as it has. Similarly, the number of closed frequent itemsets $\mathcal{C}$ is smaller than that of $\mathcal{F}$. More importantly, we can derive $\mathcal{F}$ from $\mathcal{C}$ because a frequent itemset I must be a subset of one or more closed frequent itemset, and I's support is equal to the maximal support of the closed itemsets it is contained in.

For the three kinds of frequent itemsets, $\mathcal{F}, \mathcal{M}, \text{and } \mathcal{C}$, we can get their relation which is $\mathcal{M} \subseteq \mathcal{C} \subseteq \mathcal{F}$. The maximal and closed frequent itemsets for the example above are:

$$\mathcal{C} = \{(a,7),(c,7),(e,7),(d,6),(ae,6),(ad,5),(ade,4),(ac,4),(cd,4),(ce,4),(acd,3),(ace,3),(bc,3)$$

$$\mathcal{M} = \{(acd,3),(ace,3),(ade,4)\}$$

Since $\mathcal{C}$ is smaller than $\mathcal{F}$ with no information loss about any frequent itemset, in this paper, we focus on the closed frequent itemsets mining.

III. DATA STREAM MIMING ON MAPREDUCE

We designed two methods to mine high speed data streams on Hadoop platform and to make a comparison. In both solutions, we compress the high velocity data and split it into basic blocks. Every single block is a basic window unit processed by a mapper node. For the first method, we modified the moment algorithm to fit the MapReduce framework: as data flows in, single transactions are added to FP-Tree structure to maintain the data information. When the number of transactions reaches the threshold, the Closed Enumeration Tree (CET), which will be explained in Section IIIA, will be built for the first time. Then, the new transaction continues to be added and old transaction is deleted causing update of the CET. CET maintains enough information to get the closed frequent itemsets for the data stream at any moment. For the second method, we use vertical format data to store the item and transaction information. We build a matrix for basic window units. Every item contained in the stream has a line vector which lists all the transaction identifiers cover this item. Then, we can build itemset following alphabet order with item's transaction cover vector. As computer has superiority of vector computing, the support counting and closure judgment will be easier. In Section V, we show the implementation and experiment results of the two methods on synthetic and real datasets.

A. *Moment-based MapReduce mining*

Moment[11] was used to update closed frequent itemsets for sliding window incrementally. It adopted a prefix tree structure in main memory called Closed Enumeration Tree (CET) to maintain the itemsets selected from the sliding window dynamically. The CET contains four node types which were described in detail in [11]. They are Infrequent Gateway Nodes (IGN), Unpromising Gateway Nodes (UGN), Intermediate Nodes (IN) and Closed Nodes (CN). Figure 2 shows an example of a sliding window and its CET structure in which dashed circle represents IGN, dashed rectangle represents UGN, solid circle represents IN and CN is represented by solid rectangle. From the Apriori property, all super sets of infrequent itemset are not frequent, we can get: IGN has no super set in the CET, child nodes of UGN cannot be CN so that we do not need to maintain child nodes of UGN. CET only needs to store small part of the itemsets still being able to get accurate results.

When a new transaction arrives, Moment explores nodes related to the transaction in the CET. For every node explored, Moment increase the support count and update the node type. In Figure 3, a new transaction T (tid 5) is added to the sliding window. We traverse the parts of the CET that are related to transaction T. For each related node nI , we update its support, tid sum, and possibly its node type.

When an old transaction is to be deleted, Moment also explores nodes related to the transaction in the CET. For every node explored, Moment decreases the support count and updates the node type. In Figure 4, an old transaction T (tid 1) is deleted from the sliding window. To delete a transaction, we also traverse the parts of the CET that is related to the deleted transaction. For each related node nI , we update its support, tid sum, and possibly its node type.

For its incremental way of updating for window's sliding, Moment has a formally process procedure and becomes fundamental method to mine closed frequent itemsets for data stream.

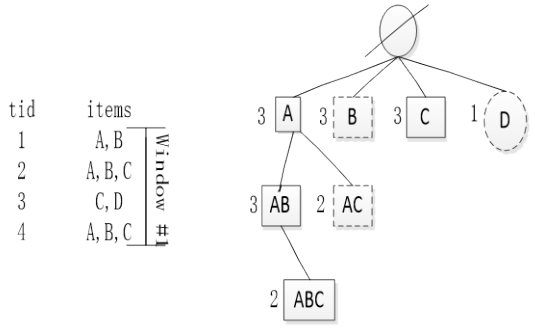

Figure 2. The Closed Enumeration Tree Corresponding to Window #1

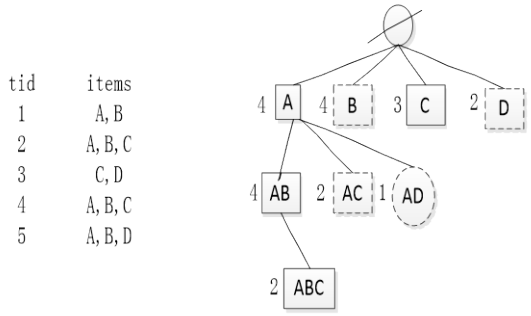

Figure 3. Adding a transaction

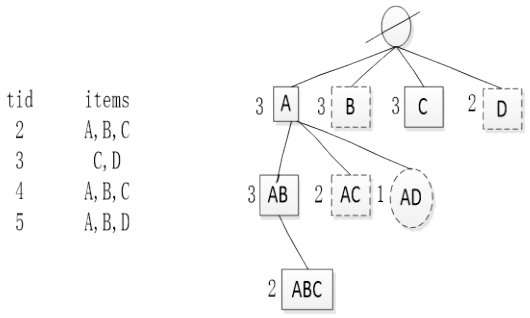

Figure 4. Deleting a transaction

The MapReduce implementation of Moment M-Moment updates the whole CET using basic window as unit. The stream receiving modular compress the data stream to transaction format that can be produced as the input of the map function. A basic window was the split for a mapper node. The mapper node mines the split unit using Moment algorithm and send the intermediate results to reducer node. The reducer node calls the reduce function to combine intermediate key-value pairs to get the whole result as the closed frequent itemsets currently for the data stream.

B. Vertical format data based MapReduce mining

Vertical format was often used in Eclat-like methods. The transaction database was transformed to item-transaction matrix. The matrix was built with tid-list rows. A tid-list consists of two fields: *Item* and *Tidset* field. The *Tidset* for an item i_p is denoted as *tidset*(i_p) which is a set of transaction identifiers containing item i_p. *Tidset* is a set structure that makes the *find(tid)* and *inter(tidset1,tidset2)* easy to implement and execute. Furthermore, we use an extended prefix tree to list itemsets with support and a hash table storing all closed frequent itemsets with their support as keys to check a new frequent itemset is closed or not.

In the following, we discuss the related algorithms to deal with window moving [20]. All the algorithms have the same input parameters (*nI*, *N*, *s*) and result in the updating of the itemset type and the hash table. *nI* stands for the item to deal with, *N* is the window size, and *s* means the relative support threshold. Figure 5 describes the algorithm to build the hash table. In the building algorithm, each n_I has a corresponding *tidset*, *Tidset(I)*, to store the transactions information in the current sliding window. Function Build is a depth-first procedure. Build visits the itemsets in a lexicographical order. In the lines 1–2 of the algorithm, function Build is performed if n_I is frequent and is not contained by other closed frequent itemsets. Function *leftcheck* uses the support of n_I as a hash key to speed up the checking. In the lines 3–5, if n_I passes the checking of the lines 1–2, Build generates all possible children of n_I with frequent siblings and creates their tidset by set intersect operation of n_I its frequent siblings. In the lines 6–7, Build recursively calls itself to check each child of n_I .In the lines 8–10, if there is no child of n_I with the same support as n_I, n_I is a closed frequent itemset and it is retained in the hash table.

```
Build (n_I, N, s)
1:   if support(n_I) = s*N then
2:       if leftcheck(n_I) = false then
3:           foreach frequent sibling n_K of n_I do
4:               generate a new child n_{I∪K} for n_I ;
5:               intersect Tidset(I) and Tidset(K) to obtain Tidset(I ∪K)
6:           foreach child n_I' of n_I do
7:               Build(n_I', N, s);
8:           if no child n_I' of n_I such that support(n_I') = support(n_I) then
9:               retain n_I as a closed frequent itemset;
10:              insert n_I into the hash table;
```

Figure 5. Algorithm of Build

When continues to read transactions after the window is full, the window slides with two operations: delete the oldest transaction and append the new incoming transaction.

Deleting the oldest transaction is the first step of window sliding. First of all, all items are visited to check if the deleted transaction contains it. Then, all items in the deleted transaction are kept and corresponding transaction id is deleted from their tidsets. Figure 6 gives the algorithm of deleting the oldest transaction after removing the transaction id from the tidsets of items. In Figure 6, the function *Delete* generates the prefix tree including the itemsets whose supports are beyond s*N - 1. This is because the supports of a set of closed frequent itemsets in previous window would be s*N and then becomes s*N - 1 after the deletion. If n_I is a closed frequent itemset, the hash table is updated. In the lines 19 and 23, if n_I is closed frequent itemset in previous window, n_I is marked as a non-closed itemset. In this case, n_I will not be retained when the function Delete is done.

Appending the incoming transaction is the second step of window sliding. The new transaction id will be added to the tidset of the items which are contained in the transaction.

```
Delete(n_I, N, s)
1:   if n_I is not relevant to the deleted transaction then
2:       return;
3:   else if support(n_I) = (s*N − 1) then
4:       foreach sliding n_K of n_I whose support = (s*N − 1) do
5:           generate a new child n_{I−K} for n_I ;
6:           intersect Tidset(I) and Tidset(K) to obtain Tidset(I ∪K)
7:       foreach child n_I' of n_I do
8:           Delete(n_I', N, s);
9:       if support(n_I) = (s*N) then
10:          if leftcheck(n_I) = false then
11:              if n_I is closed frequent itemset in previous window then
12:                  update the support of n_I;
13:                  update n_I in the hash table;
14:              else
15:                  retain n_I as a closed frequent itemset;
16:                  insert n_I into the hash table;
17:          else
18:              if n_I is closed frequent itemset in previous window then
19:                  mark n_I as a non-closed frequent itemset;
20:                  remove n_I from the hash table;
21:      else
22:          if n_I is closed frequent itemset in previous window then
23:                  mark n_I as a non-closed itemset;
24:                  remove n_I from the hash table;
```

Figure 6. Algorithm of Delete

```
Append (n_I, N, s)
1:    if support(n_I) = s*N then
2:        if leftcheck(n_I) = false then
3:            foreach frequent sibling n_K of n_I do
4:                generate a new child n_{I∪K} for n_I ;
5:                intersect Tidset(I) and Tidset(K) to obtain Tidset(I ∪K)
6:            foreach child n_I' of n_I do
7:                Append (n_I', N, s);
8:            if no child n_I' of n_I such that support(n_I') = support(n_I) then
9:                if n_I is closed frequent itemset in previous window then
10:                   update the support of n_I;
11:                   update n_I in the hash table;
12:               else
13:                   retain n_I as a closed frequent itemset;
14:                   insert n_I into the hash table;
```

Figure 7. Algorithm of Append

Figure 7 gives the algorithm of appending a new incoming transaction after the tidset adding. Function Append is almost the same as Build. The only difference is in the lines 9–11. If the checked closed frequent itemsets are already in the hash table, Append updates the hash table.

The MapReduce implementation M-vertical is similar to the content described in Section III.A.

IV. EXPERIMENTAL RESULT

In this section, we evaluate the performance of the MapReduce implementation of the two methods and make comparison between them. The Java source code of the essential version of Moment is downloaded from the open source site www.admire-project.eu (by Maciek Jarka), and Java source code of a method use vertical format data to mine frequent itemsets is derived from [21]. The Hadoop version is 1.2.1. All experiments are done on a cluster of computers with 2GB memory and Pentium（R）Dual CPU E2200@2.20GHz running on Ubuntu 12.04 OS. We generate a synthetic dataset T10I4D100K from IBM data generator [1]. The parameters are described as follows: T is average transaction size; I is average size of maximal potential frequent itemsets; D is the total number of transactions. Besides, a real-world dataset Mushroom was downloaded from FIMI Repository [22].

A. *Mining with different minimum supports*

In the first experiment, the minimum support threshold is changed from 1% to 0.1%, and the size of the sliding window is fixed to 1000 transactions.

Figure 8 shows the loading time of the first window. In the first window, both methods need to build a prefix tree. It can be observed that the vertical based method is faster than M-Moment. It is because that generating candidates and counting their supports with vector set is more efficient.

Figure 9 shows the average time to process single transaction when window slides. It also shows that the later method is faster for similar reason. When Moment slide the window, the adding and deleting of transaction cause explore of the tree structure. However, when M-vertical method slides, the algorithm only visit items that the added or deleted transaction contains and the updating of the hash table is very fast.

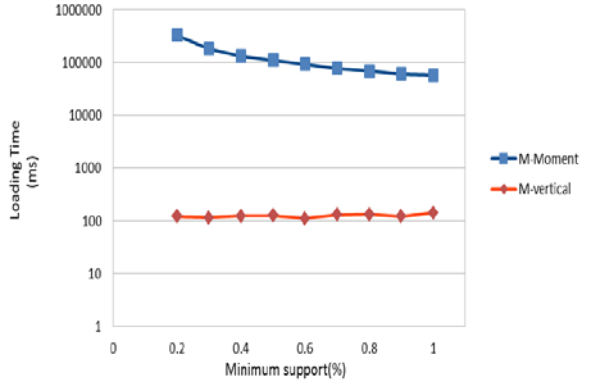

Figure 8. Time of loading the first window with different minimum supports

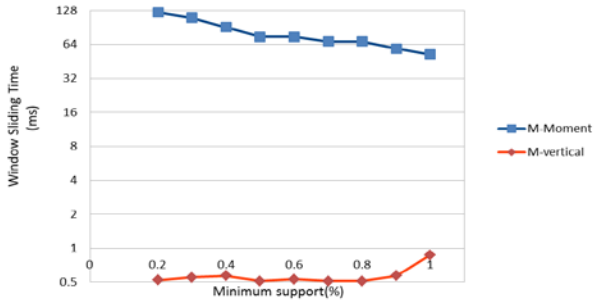

Figure 9. Average time of window sliding with different minimum supports

Because of the vertical format data structure, it can also be seen that the metric change extent of the latter method is not as much as the former one.

B. *Mining with different number of mappers*

In this experiment, the number of mappers for the two methods is changed from 1 to 10. The size of the basic window is fixed to 10000 and minimum support threshold is set to 0.1%. Figure 10 shows the total execution time to process 100000 transactions with the two methods.

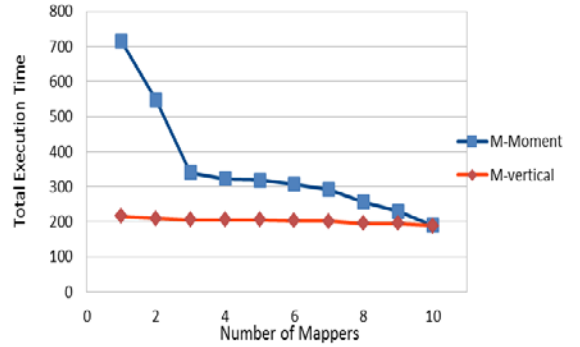

Figure 10. Total execution time with different number of mappers

It can be seen that for the MapReduce Moment method when mapper nodes increase the total time decrease a lot. For the Mapreduce vertical method, the total execution time also decreases a little as the number of mappers increase. But the change is not as obvious as the M-Moment. We can conclude that with more mapper nodes, the ability of both methods improves.

V. CONCLUSION AND FUTURE WORK

In this paper, we extend and implement two types of methods to do experiments on Hadoop platform to mine closed frequent itemset over fast data streams. We firstly use CET structure and Moment algorithms to mining. Then, we introduce vertical format data to maintain item-transaction information. Experiments show that vertical format data method has higher speed and performance to process fast data streams. Through extend implementation on Hadoop we observed that increasing number of mappers can improve both methods' ability to face up with fast data streams. As for the future work, we consider to design new methods fitting MapReduce better and to do experiments on cluster with more nodes to make the results more clear.

ACKNOWLEDGMENT

We thank Maciek Jarka and Sandy Moens and team for sharing their codes online.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," Proc. 20th int. conf. very large data bases, VLDB. vol. 1215, Sep. 1994, pp. 487-499.

[2] Zaki and M. Javeed, "Scalable algorithms for association mining," Knowledge and Data Engineering, IEEE Transactions, Dec. 2000, pp. 372-39, doi:10.1109/69.846291.

[3] J. Han, J. Pei, and Y. Yin. "Mining frequent patterns without candidate generation." ACM SIGMOD Record. vol. 29, May. 2000, pp. 1-12, doi:10.1145/342009.335372.

[4] M. Garofalakis, J. Gehrke, and R. Rastogi, "Querying and mining data streams: you only get one look a tutorial,". In SIGMOD Conference , vol. 2002, Jun. 2002, p. 635, doi:10.1145/564691.564794.

[5] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," In Proceedings of the 28th international conference on Very Large Data Bases, Aug. 2002, pp. 346-357, doi:10.14778/2367502.2367508.

[6] H. F. Li, S. Y. Lee, and M. K. Shan, "An efficient algorithm for mining frequent itemsets over the entire history of data streams," In Proc. of First International Workshop on Knowledge Discovery in Data Streams, Sep. 2004.

[7] J. X. Yu, Z. Chong, H. Lu, and A. Zhou, "False positive or false negative: mining frequent itemsets from high speed transactional data streams," In Proceedings of the Thirtieth international conference on Very large data bases, vol. 30, Aug. 2004, pp. 204-215.

[8] J. H. Chang and W. S. Lee, "Finding recent frequent itemsets adaptively over online data streams," In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, Aug. 2003, pp. 487-492, doi:10.1145/956750.956807.

[9] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," Next generation data mining, 2003, pp. 191-212.

[10] J. H. Chang, and W. S. Lee, "estWin: adaptively monitoring the recent change of frequent itemsets over online data streams," In Proceedings of the twelfth international conference on Information and knowledge management, Nov. 2003, pp. 536-539, doi:10.1145/956863.956967.

[11] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, "Moment: Maintaining closed frequent itemsets over a stream sliding window," In Data Mining, 2004, ICDM'04, Fourth IEEE International Conference on, Nov. 2004, pp. 59-66, doi:10.1109/ICDM.2004.10084.

[12] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," In ACM SIGOPS operating systems review , vol. 37, No. 5, Oct 2003, pp. 29-43, doi:10.1145/1165389.945450.

[13] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Communications of the ACM, Vol 51, Jan. 2008, pp. 107-113, doi:10.1145/1327452.1327492.

[14] Z. Farzanyar, and N. Cercone, "Accelerating Frequent Itemsets Mining on the Cloud: A MapReduce-Based Approach," In Data Mining Workshops (ICDMW), IEEE 13th International Conference on, Dec. 2013, pp. 592-598, doi:10.1109/ICDMW.2013.106.

[15] Z. Farzanyar, and N. Cercone, "Efficient mining of frequent itemsets in social network data based on MapReduce framework," In Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, Aug. 2013, pp. 1183-1188, doi:10.1145/2492517.2500301.

[16] F. Kovacs, and J. Illés, "Frequent itemset mining on hadoop," In Computational Cybernetics (ICCC), 2013 IEEE 9th International Conference on, July. 2013, pp. 241-245, doi:10.1109/ICCCyb.2013.6617596.

[17] H. Chen, T. Y. Lin, Z. Zhang, and J. Zhong, "Parallel mining frequent patterns over big transactional data in extended mapreduce," In GrC , Dec. 2013, pp. 43-48, doi:10.1109/GrC.2013.6740378.

[18] X. Wei, Y. Ma, F. Zhang, M. Liu, and W. Shen, "Incremental FP-Growth mining strategy for dynamic threshold value and database based on MapReduce," In Computer Supported Cooperative Work in Design (CSCWD), Proceedings of the 2014 IEEE 18th International Conference on, May 2014, pp. 271-276, doi:10.1109/CSCWD.2014.6846854.

[19] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," In Database Theory—ICDT'99 , vol. 1540, Jan. 1999, pp. 398-416, doi:10.1007/3-540-49257-7_25.

[20] H. F. Li, C. C. Ho, and S. Y. Lee, "Incremental updates of closed frequent itemsets over continuous data streams," Expert Systems with Applications, vol. 36, Mar. 2009, pp. 2451-2458, doi:10.1016/j.eswa.2007.12.054.

[21] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," In Big Data, 2013 IEEE International Conference on, Oct. 2013, pp. 111-118, doi:10.1109/BigData.2013.6691742.

[22] Frequent Itemset Implantation Repository(FIMI), http://fimi.cs.helsinki.fi/,[retrieved: May, 2015].

 ISBN: 978-1-61208-415-2