Sequence-Growth : A Scalable and Effective Frequent Itemset Mining Algorithm for Big Data Based on MapReduce Framework

Yen-hui Liang* and Shiow-yang Wu†
Dept. of Computer Science and Information Engineering
National Dong Hwa University
Hualien, Taiwan
*Email: *810021004@ems.ndhu.edu.tw, †showyang@mail.ndhu.edu.tw*

***Abstract*—Frequent itemset mining(FIM) is an important research topic because it is widely applied in real world to find the frequent itemsets and to mine human behavior patterns. FIM process is both memory and compute-intensive. As data grows exponentially every day, the problems of efficiency and scalability become more severe. In this paper, we propose a new distributed FIM algorithm, called Sequence-Growth, and implement it on MapReduce framework. Our algorithm applies the idea of lexicographical order to construct a tree, called "lexicographical sequence tree", that allows us to find all frequent itemsets without exhaustive search over the transaction databases. In addition, the breadth-wide support-based pruning strategy is also an important factor to contribute the efficiency and scalability of our algorithm. To test the performances of our algorithm, we conduct varied aspects of experiments on MapReduce framework with large datasets. The results show the good efficiency and scalability of Sequence-Growth especially to deal with big data and long itemsets. Our algorithm also proposes a new mining methodology which can be easily modified for sequential pattern mining ,trajectory pattern mining and other associate rule mining algorithms. We believe that it should have a valuable contribution in the future development of association rule mining algorithms for big data.**

***Keywords-Frequent Pattern Mining; MapReduce; Big Data; Scalability; Efficiency*.**

I. Introduction

Frequent pattern mining is one of the most important techniques of data mining because of its wide applications in a variety of domains, such as market basket analysis, web click analysis, patient path analysis, DNA sequence discovery and more recently defect-free rate improvement in semiconductor industry.

The problem of Frequent Itemset Mining (FIM) was first presented in the research paper published by R. Agrawal and R. Srikant in 1995 [1] which is an extension work of the authors' research in 1994[2]. In the paper, the authors introduced two Apriori-based algorithms, called AprioriAll and AprioriSome. Since then, the problem of FIM attracted many attentions and became an important research topic of data mining. The Apriori algorithm also became one of the most important concept of FIM and many algorithms were developed based on it. FIM problem is defined as follows. Let I = $\{i_1, i_2,...,i_n\}$ be a set items. T = (tid,X) is a transaction where tid is a transaction identifier and X is a set of items over I. A transaction database D = $\{t_1, t_2,...,t_m\}$ is a set of transactions. The support of an itemset Y is defined as the number of transactions that contain the itemset. Formally,

$$support(Y) = |\{tid \mid Y \subseteq X, (tid,X) \in D \}|$$

An itemset is called frequent if its support is not less than a user-specified value, which is called *minimum support*.

Another approach of FIM is called Eclat. Unlike Apriori using bottom up, breadth-first search strategy, Eclat[3] uses a breadth-first approach with set intersection to mine frequent itemsets. Eclat transforms the transaction database into its vertical format. Each itemset is stored together with its cover (also called $tid - list$). The vertical database D' is defined as:

D' = $\{(i_j,C_{ij} = \{tid| \ i_j \in X, (tid,X) \in D\})\}$, where C_{ij} is the *tid-list* of i_j.

In the vertical database, the support of an itemset Y can be computed by intersecting the $tid - list$ of any two subsets. It is defined as:

$$support(Y) = |\bigcap_{t_j \in Y} C_{ij}|$$

In 2014, Walmart handles more than 1 million customer transactions every hour. Facebook stores, accesses, and analyzes 30+ petabytes of user generated data and the number keeps growing. With the exponential growth of data volume towards a terabyte or even petabyte, it becomes unfeasible to execute association rule mining on a single machine. Implementing a scalable and efficient distributed algorithm holds the key to solving the problem to deal with big data [4]. With the abundant storage and computing resources on cloud, paralleled and distracted association rule mining can be attained more easily. In addition, the introduction of MapReduce by Google in 2004 [5] also made a great contribution to the advent of distributed association rule mining. Many algorithms were developed or modified to implement on MapReduce framework [6]-[18]. Specific for FIM, those include the various revisions of serial Apriori[6],[8],[9], Eclat[14],[15],[17] and a hybrid method of Apriori and Eclat[16],[17] to adapt on the MapReduce framework.

Apriori-based algorithms mine frequent itemsets with a generation-scanning manner. The most convenient way to adapt a serial Apriori algorithm onto the MapReduce framework involves multiple iterations of the MapReduce job for generating candidate sequences and scanning for their containment in the database. However, the MapReduce programs usually read input data from a distributed file system (DFS), and will also write the output into DFS[5],[19] The overhead of MapReduce-based Apriori FIM algorithm is potentially expensive because of the cost of initiation and communication of MapReduce jobs. In addition, same as serial Apriori, one of the most critical challenges faced by MapReduce-based Apriori is the expensive candidate itemset generation and transaction database scanning. The previous studies have shown up that the situation is even more severe for Apriori-based algorithms in processing a huge dataset or long transaction length[21],[22]. For reducing the MapReduce iterations, Eclat uses a vertical database format to construct a prefix tree for each itemset. It traverses the prefix tree in a depth-first manner to mine the frequent itemsets by applying set intersection of any two subsets. However, in this way, the vertical database needs to be stored in main memory, which is another challenge on dealing with a large scale transaction database.

In this paper, we proposed a new MapReduce-based FIM algorithm, called Sequence-Growth. Our algorithm applies the idea of lexicographical order to construct the candidate sequence subsets to avoid from expensive scanning processes. In addition, Sequence-Growth produces frequent itemsets in a breadth-wide support-based approach(we called it "lazy mining"). In each MapReduce iteration, the infrequent itemsets will be pruned away. It significantly deducts memory consumption and initiation time of each MapReduce job. To evaluation the performances of our algorithm, we conduct varied experiments to compare it with other three distributed FIM algorithms, MapReduce-based Apriori, One-Phase and BigFIM. The results show that Sequence-Growth algorithm outperforms the others. Moreover, the method used for constructing the lexicographical sequence tree in Sequence-Growth can be modified easily for other single machine association rule mining algorithms to adapt on the MapReduce framework.

The rest of this paper is organized as follows. We will discuss MapReduce-based Apriori, One-Phase and BigFIM algorithm in section II. Section III discusses Sequence-Growth in details. We also provide an example to demonstrate that a distributed trajectory pattern mining algorithm can be developed by modifying slightly the map function of Sequence-Growth algorithm. The comprehensive experiments and results are shown in section IV. The conclusions will be given in Section V.

II. Related Works and Comparative Algorithms

A. MapReduce-based Apriori Algorithm

The most natural way to adapt serial Apriori algorithm on the MapReduce framework is to convert the processes of generating candidate itemsets and scanning for the containments into map-reduce functions and assign the jobs iteratively. The algorithms designed in this manner are categorized into "k-phase" algorithms, because a length-k transaction database(k is the longest transaction length) needs at most k iterations to mine complete frequent itemsets. Each phase consists of two MapReduce jobs, one for generating candidate itemsets and another for scanning the database for counting their occurrences. The two MapReduce jobs will continue to run alternatively and iteratively until no any candidate itemset is generated. Because MapReduce framework is not well suitable for iterative processes, serval reversion of MapReduce-based Apriori algorithms were developed to decrease the initiation cost by optimizing the number of MapReduce phases needed to mine complete frequent itemsets. Those works include Lin et al. proposed two algorithms, *Fixed Passes Combined-Counting* (FPC) and *Dynamic Passes Counting* (DPC) in their publication[12]. Given n and p as parameters, FPC starts to generate candidates with n different lengths after p phases and counts their frequencies in one database. DPC algorithm is similar to FPC, excepts that n and p is determined dynamically according to the number of generated candidates at each phase.

B. One-Phase Algorithm

One-Phase algorithm needs only one MapReduce job to mine all frequent itemsets. The alogrithm generates all possible subsets(combinations) of transactions line by line in its map function, and sums up the global supports in the reduce function. An approach like this, the growth rate of candidate itemsets is exponential proportion to the average length of transactions. For example, with an average length k and total number of transactions T, One-Phase needs to generate average $C = |T| \times (2^k - 1)$ candidate itemsets. Generating such huge candidate datasets, the scalability of One-Phase obviously cannot be satisfied. One of related works proposed by Lin et al., is called *Single Pass Counting*(SPC). SPC together with FPC and DPC were used to analyze the impact on performance of different implementations. Another One-Phase algorithm was proposed by Li et al., called *PApriori* is very similar to SPC with minor differences on implementation[8].

C. Distributed Eclat and BigFIM

In 2013, Moens et al.[17] proposed two distributed FIM algorithms on the MapReduce platform, called Dist-Eclat and BigFIM. Dist-Eclat is a distributed version of Eclat that focuses on speed. The operation of Dist-Eclat consists of three steps. During the first step, Dist-Eclat divides the vertical database into serval equal shards for distribution to available mappers. The purpose of this step is to extract the frequent singletons(large-1 pattern) from the shards. In the second step, the singletons with their tid-lists are distributed to the mappers. Each mapper finds the frequent k-sized supersets of the items by running Eclat to level k. During the last step, Dist-Eclat mines the prefix tree starting at a prefix from the assigned batch using Eclat. On dealing with big data, Dist-Eclat suffers a problem that tid-list of even a single item may not fit into memory. To overcome the problem, Moens et al. proposed the second algorithm, called BigFIM, that is a hybrid method of Apriori and Eclat. During the first step, BigFIM uses an Apriori based method which is very similar to One-Phase to extract frequent itemsets of length k. After computing the k-prefixes, the second step is to compute the possible extensions, i.e., obtaining tid-lists for $(k+1)$-FIs. During the last step, BigFIM switches to Eclat, and then utilizes diffset operations to mine the subtrees for frequent itemsets with a depth-first manner.

III. Proposed Algorithm : Sequence-Growth

A. *Lexicographical Sequence Tree*

According to the paper of $CloSpan$ by Yan et al.[20], the lexicographic order set is defined as following : Let $t = \{j_1, j_2,...,j_k\}$, $t' = \{j'_1, j'_2,...,j'_l\}$. Then $t < t'$ iff either of the following is true :

1) for some h, $0 \leq h \leq min\{k, l\}$, we have $j_r = j'_r$ for $r < h$, and $j_h < j'_h$, or
2) $k < l$, and $j_1 = j'_1$, $j_2 = j'_2$,...,$j_k = j'_k$

In addition, according to the definition of Apriori[1], a sequence $\beta = (b_1, b_2, ..., b_n)$ is a subsequence of $\alpha = (a_1, a_2, \ldots, a_m)$, and α is a super sequence of β, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_n$, $n \leq m$ such that $b_1 \subseteq a_{i1}$, $b_2 \subseteq a_{i2}$,..., $b_n \subseteq a_{in}$. By the above definitions, we can construct a lexicographical sequence tree in a breadth-first sequence-growth manner. Given a sequence $S = \{s_1, s_2, \ldots, s_n\}$, and its two subsequences $p = \{s_k, s_{k+1}, \ldots, s_m\}$, and $q = \{s_{m+1}, s_{m+2}, \ldots, s_n\}$, where q is the suffix of p, we can present the lexicographical sequence tree L as the following:

$$L = \bigcup_{l=0}^{n-1} \{p_l \diamond \alpha \mid \forall \alpha \in q_l \}$$

$(p_l \diamond \alpha)$ denotes p_l concatenates with α.

The lexicographical sequence tree functions as the construction of our candidate generation space. Each node in the lexicographical sequence tree represents a subsequence, and its height corresponds to the length of the subsequences at that level. The lexicographical sequence tree can be extended by concatenating each node with its suffix iteratively in a sequence-growth manner. A length-(k+1) sequence is a concatenation of its parent nodes (length-k) and an item of the parent's suffix. For example, $V = (s_1, s_2, s_3)$ is a length-3 subsequence of S. A length-4 subset $V' = \{ (s_1, s_2, s_3, s_4), (s_1, s_2, s_3, s_5), (s_1, s_2, s_3, s_6), \ldots, (s_1, s_2, s_3, i_n)\}$ is obtained by appending each item that follows s_3 in S to the end of V. Starting from length-1 to n, all subsequences of S can be retrieved from the lexicographical sequence tree. An example of lexicographical sequence tree is provided in Figure 1.

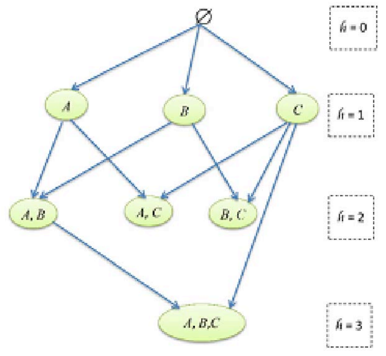

Figure 1. An example of lexicographical sequence tree.

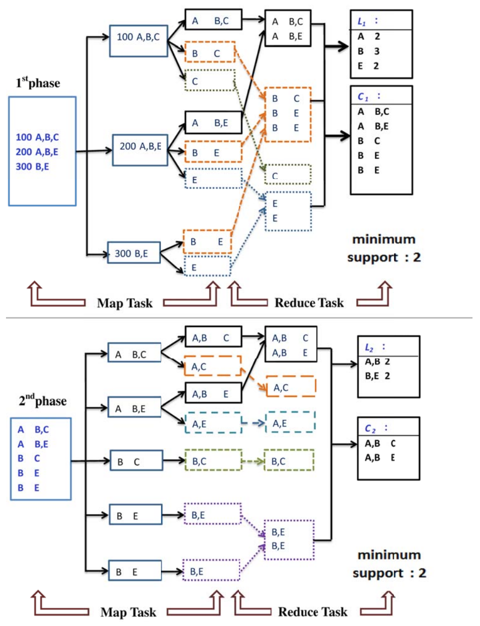

Figure 2. An example to illustrate the processes of Sequence-Growth algorithm.

B. *"Lazy mining":Pruning Strategy*

The pruning strategy is also an important factor for designing an efficient FIM algorithm. Unlike One-Phase algorithm that generates all possible subsets of the transaction database in a single iteration, Sequence-Growth executes iteratively with cardinality from 1 to k to generate candidate itemsets. The delay in generation of candidate itemsets, called "lazy mining" by us, is an efficient pruning strategy of our proposed algorithm. Applying the Apriori property to our lexicographical sequence tree, if a node is infrequent, all of the child nodes can be pruned away. Sequence-Growth cuts down the size of candidate itemsets for each iteration, and significantly decreases memory consumption. Therefore, the execution can fit the data into the main memory to prevent from expensive context swapping processes. Sequence-Growth algorithm is designed according to the concepts of lexicographical sequence tree and lazy mining pruning strategy, and is implemented on MapReduce framework for a distributed execution.

C. Implementation of Algorithm on MapReduce Framework

In the Sequence-Growth algorithm, each iteration consists of a map and a reduce task. The process of generating frequent itemsets in our algorithm follows the structure of lexicographical sequence tree with a sequence-growth manner. The length-k frequent itemsets represent the internal nodes on height-k of the lexicographical sequence tree. Sequence-Growth consists of two steps. The first step of Sequence-Growth produces a set of large-l items with their suffixes. The second step runs a MapReduce job iteratively to mine length-k frequent itemsets. The output of each Sequence-Growth iteration k consists of two distributed files, L_k and C_k. Both of them contain a set of key-value pairs. In L_k, each key is a length-k frequent itemset and the value is its global support in the transaction database. In C_k, the keys are the same, but the values are the suffix of frequent itemsets. In the next iteration, C_k dataset is used to generate $L(k+1)$ and $C(k+1)$ by appending the length-k itemsets with each item in their suffixes. The MapReduce iterations of Sequence-Growth continue until the output dataset is empty. In Figure 2, we show an example to illustrate the processes of our algorithm. The pseudo codes of Sequence-Growth are stated in algorithm 1 to 3.

Algorithm 1 SequenceGrowth //Main Program of Sequence-Growth

```
Input: $S$: { $t \mid t \in S_i$ , $t = < i_1,...,i_k >$ };
       //A transaction database
       $\delta$: integer; //minimum support threshold
Output: $L$: { $p \mid p \in L$ , $p = < key,value >$ };
1: var[] $L = \phi$ , $C = \phi$, $T = \phi$;
2: $S_i$ = PartitionOf($S$); //$S_i$ is a split of $S$
3: ($L$,$C$) = GenLarge1($S_i$ , $\delta$);
4: while ($L \neq \phi$ )and( $C \neq \phi$) do
5:     $S_i$ = PartitionOf($C$);
6:     $C$=$\phi$ , $T$=$\phi$;
7:     ($T$,$C$) = GenFrequentItemset($S_i$ , $\delta$);
8:     $L$ =$\bigcup T$;
9: end while
```

Algorithm 2 GenLarge1 //To generate large-1 items

```
Input: $S_i$: { $t \mid t \in S_i$ , $t = < i_1,...,i_k >$ };
       $\delta$: integer; //minimum support threshold
Output: $L_1$: { $p \mid p \in L_1$ , $p = < key,value >$ };
        $C_1$: {$c \mid c \in C_1$ , $c = < key,value >$ };

    Map Task ($key$ , $value$)
1: for each $t$ in $S_i$ do
2:     var[ ] $itemlist$;
3:     $itemlist$ = $t$.split(",");
4:     for ($k$ = 0; $k < itemlist$.length; $k$++) do
5:         $key$ = itemlist[$k$];
6:         $value$ = SuffixOf(itemlist[$k$]);
7:         Output($key$, $value$);
8:     end for
9: end for

    Reduce Task ($key$ , $Value$[])
10: var $sum$ = 0;
11: var[ ] $subseq$;
12: var[ ] $str$[2];
13: for each $v$ in $Value$ do
14:     $sum$++;
15:     $subseq$ = ArraysCopyOf($subseq$, $sum$);
16:     $subseq$[$subseq$.LastElement] = $v$;
17: end for
18: if ($sum \geq \delta$) then
19:     MultipleOutput($key$, $sum$) $\rightarrow L_1$;
20:     for ($k$ = 0; $k < subseq$.length; $k$++) do
21:         while ($subseq$[$k$].length $\geq MaxMemory$) do
22:             $str$ = Split($subseq$[$k$],$MaxMemory$);
23:             MultipleOutput($key$, $str$[0]) $\rightarrow C_1$;
24:             $subseq$[$k$] = $str$[1];
25:         end while
26:         MultipleOutput($key$, $subseq$[$k$]) $\rightarrow C_1$;
27:     end for
28: end if
```

D. Extension of Sequence-Growth Method for Trajectory Pattern Mining

One of the contributions of this paper is that the method used to construct the lexicographical sequence tree in Sequence-Growth can be modified easily for other single machine association rule mining algorithms to adapt on the MapReduce framework. The idea is that generally each item in a transaction can be separated into two itemsets, a prefix and a suffix. Therefore, the method for building the lexicographical sequence tree in our algorithm can also apply to other association rule mining algorithms to construct their candidate spaces. In this section, we provide an example algorithm for trajectory pattern mining(TPM) which is a minor revision of Sequence-Growth. Because research issues of TPM is beyond the scope of this paper, the problem of TPM in this paper is simplified for easy understanding. More detailed descriptions please refer to the work of Giannotti et al.[23].

A trajectory record can be represented as $\mathbb{T}$=($\mathbb{S}$, $\mathbb{A}$), where $\mathbb{S}$ = (ρ_0, ρ_1,...,ρ_n) is a sequence of (n+1) movement locations and $\mathbb{A}$ = (μ_1, μ_2,...,μ_n) is a set of transition times such that $\mu_i = \Delta t_i = t_i - t_{i-1}$. A trajectory database $\mathbb{D}^t = \{t_1, t_2,...,t_m\}$ is a set of trajectories. $_{suppD}(\mathbb{S})$ is the frequency of $\mathbb{S}$ in trajectory database $\mathbb{D}^t$. Given a minimum support threshold s_{min} and a time tolerance τ, $\mathbb{T}$=($\mathbb{S}$, $\mathbb{A}$) is frequent, called T-Pattern, if $_{suppD}(\mathbb{S}) \geq s_{min}$ and $\mu_i \leq \tau$ for each $\mu_i \in \mathbb{A}$. TPM is to find all T-Pattern ($\mathbb{S}$, $\mathbb{A}$) in a trajectory database $\mathbb{D}^t$. The mining procedure starts from finding all "hot" spots. The step is similar to mine the length-1 itemset in Sequence-Growth algorithm, excepts that we don't output the suffix locations(movement locations after the current spot) if their transition times are greater than time tolerance. The algorithm 4 contains the pseudo codes of this step. Because the main program and reduce function is very similar to algorithm 1 and the reduce function in algorithm 2 respectively, they are omitted for saving pages. Notes that, in algorithm 4 we traverse the sequence of locations to cut off those trajectories whose transition times are greater than the time tolerance(line 7 to 9). The next step consists of mining all T-patterns by applying the method of lexicographical sequence tree iteratively to extend the sequence-length by one in each mapreduce job.

IV. Evaluations and Results

The experiments were executed on Hadoop 1.2.1 [24] and jdk-7u6 in a fully distributed cluster environment consisting of 7 machines. Each one contains 6 cores of Intel (R) I7-4930 3.4GHz CPU and 64G RAM. All machines run on Ubuntu 12.04 operating system. One machine serves as master and slave, the others serve as slave only. The synthetic datasets, which are generated by IBM Quest Synthetic Data Generator [25], were used in our experiments. The parameters of synthetic datasets are: the average transaction length (T), the average maximal frequent itemset length (I), and the total number of transactions (D), respectively.

First, we use different size of transaction databases to test the scalability of Sequence-Growth algorithm. The size of synthetic transactional datasets is from 50K to 8000K. With all datasets, (T) is 10, (I) is 4, and minimum support δ is 0.1%. In Figure 3, the results show that Sequence-Growth provides a good scalability. The slope (Δ of processing time / Δ of transaction size) represents the processing time needed for increasing 1K transactions. As we notice that a considerable increase in the processing time begins when the transaction size grows from 4000K to 6000K. Even then, the Sequence-Growth algorithm still presents the capability to process such large datasets within a reasonable time. A further test for the capability of our algorithm on dealing with big data, we execute Sequence-Growth with the datasets from 10 millions up to 20 millions of transactions with (T) at 6 and minimum support at 1%. The result is shown in Figure 4. It again confirms the good scalability of Sequence-Growth.

Besides the huge number of transactions, a wide range of transaction length is another challenge of FIM on dealing with big data. Next, we set up an experiment to verify

Algorithm 3 GenFrequentItemset //Iterative execution to generate length-k frequent itemsets for k $\geq$ 2

```
Input: : $S_i$: { $t \mid t \in S_i$ , $t$ =(< $pattern, suffix$ >) };
         $\delta$: integer; //minimum support threshold
Output: : $L_k$: { $p \mid p \in L_k$ , $p$ = < $key$,$value$ > };
          $C_k$: { $c \mid c \in C_k$ , $c$ = < $key$,$value$ > };
    Map Task ($key$ , $value$)
 1: for each $t$ in $S_i$ do
 2:     for ($k$ = 0; $k$ < $itemlist$.length; $k$++) do
 3:         $key$ = $prefix$.append($itemset[k]$);
 4:         $value$ = SuffixOf($itemset[k]$);
 5:         Output($key$, $value$);
 6:     end for
 7: end for

    Reduce Task ($key$ , $Value[]$)
    Same as the Algorithm 2
```

Algorithm 4 GenHotSpot //To generate hot spots

```
Input: $\mathbb{S}$: { $t \mid t \in \mathbb{S}_i$ , $t$ = < $\rho_0$,...,$\rho_n$@$\mu_1$,...,$\mu_n$> };
       $s_{min}$: integer;
       $\tau$: integer;
Output: $L_1$: { $\rho \mid \rho \in L_1$ , $\rho$ = < $key$,$value$ > };
        $C_1$: {$c \mid c \in C_1$ , $c$ = < $key$,$value$ > };

    Map Task ($key$ , $value$)
 1: for each $t$ in $S_i$ do
 2:     $loclist$ = $t$.split("@")[0].split(",");
 3:     $timelist$ = $t$.split("@")[1].split(",");
 4:     for ($k$ = 0; $k$ < $loclist$.length; $k$++) do
 5:         $tranTime$ = 0;
 6:         $len$ = $k$;
 7:         while ($tranTime$ += $timelist[len] \leq \tau$) do
 8:             $len$++;
 9:         end while
10:         $key$ = $loclist[k]$;
11:         if ($len \geq k$) then
12:             $value$ = $loclist$.SubArray($k$+1,$len$);
13:             $value$ += "@";
14:             $value$ += $timelist$.SubArray($k$,$len$-1);
15:         end if
16:         $Output$($key$, $value$);
17:     end for
18: end for
```

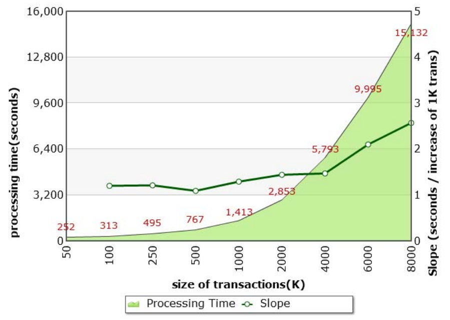

Figure 3. Execution time with different size of transactions.

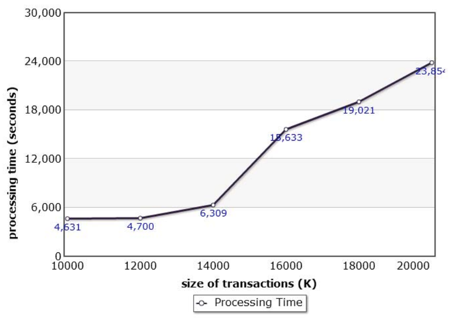

Figure 4. Execution time of Sequence-Growth running with large scale datasets.

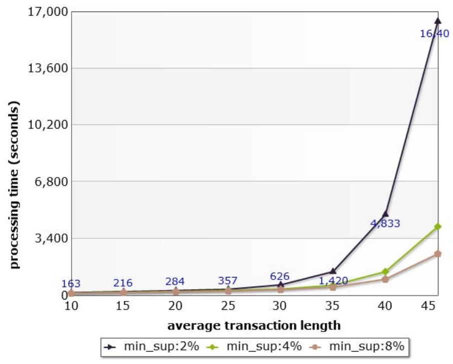

Figure 5. Execution time comparison for different value of δ with the varied transaction length.

that Sequence-Growth can remedy the problem by partition evenly after the second MapReduce iteration and by its effective pruning strategy. We use 2%, 4% and 8% minimum support to execute Sequence-Growth with different average transaction length. Figure 5 shows the result that Sequence-Growth is efficient and scalable in processing a high variable range of transaction lengths, from the shortest 3 up to longest 107. Sequence-Growth is designed to be able to partition evenly with the input files. After the second iteration of MapReduce job, Sequence-Growth divides an extreme long line into several shorter lines. Therefore, the input dataset is able to be partitioned and distributed evenly to each mapper. It prevents the processes from being held up to wait for the mappers with heavy job. The codes of this part are stated in lines 21-25 of algorithm 2. Another observation is that the processing time for 2% minimum support increases 3.4 times when the average transaction length grows from 35 to 40. However, it is only 2 times for 8% minimum support. It is because that the ”lazy mining” pruning strategy of Sequence-Growth takes effect. The higher minimum support threshold is used the more itemsets are pruned away during each MapReduce iteration. The experimental results verify that our breadth-wide support-based pruning strategy is an important factor for making the performance of Sequence-Growth efficient especially in processing long transactions.

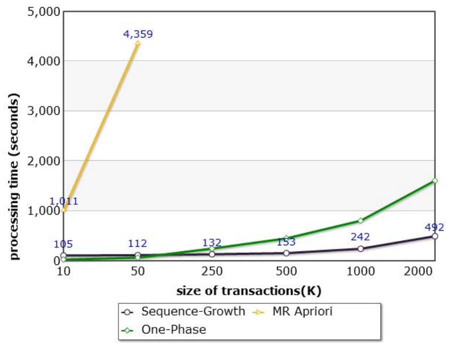

Figure 6. Execution time comparison with different transaction size.

For further evaluation of our algorithm, we conduct several experiments to compare with other two Apriori-based algorithms, MR-Apriori and One-Phase. We first investigate the performance of the three algorithms with different sizes of transaction datasets, from 10K to 2000K. The parameter of (T) is 10 for all datasets, and the minimum support δ is 0.1%. The results are shown in Figure 6. It can be noticed that the executions of MR-Apriori were failed when the size of datasets is larger than 50K. Like serial Apriori, MR-Apriori still needs to repeatedly run one MapReduce job for generating candidate itemsets and another job for scanning database. Even implementing the algorithm on MapReduce framework for distributed execution, when the size of dataset becomes huge, both memory use and computational cost can still be very expensive. We also notice that the processing time of One-Phase increases significantly after 250K. The execution time of One-Phase is approximately two times higher than that of Sequence-Growth at transaction size 250K, but it is more than 3 times higher at 2000K. Based the results, our algorithm is more efficient on dealing with huge dataset comparing MR-Apriori and One-Phase.

Figure 7 shows a comparison of performance among the three algorithms using 10K of synthetic dataset with different levels of average transaction length. Even though the performance of One-Phase is better in processing dataset with a relative short length, the running time of One-Phase increases dramatically when the transaction length is longer than 14. The reason is that the size of intermediate data generated by map function of One-Phase algorithm is exponential proportion to the transaction length. The execution cannot fit into the main memory, and thus the performance of One-Phase is inefficient and even unfeasible when the transaction length becomes longer.

Both One-Phase and Sequence-Growth algorithm mines frequent itemsets without scanning whole transaction database repeatedly. The crucial difference between the two algorithms is that One-Phase generates all possible subsets of the transaction database in a single MapReduce job. In contrast, Sequence-Growth produces frequent itemsets in a breadth-wide support-based approach (lazy mining). Based

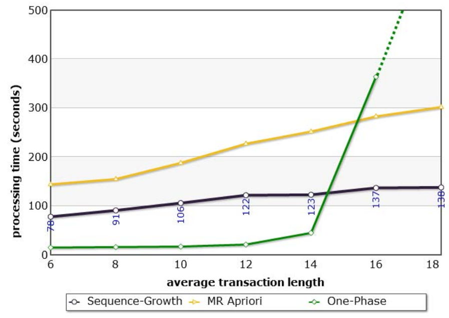

Figure 7. Execution time comparison of One-Phase, MapReduce-based Apriori and Sequence-Growth with average transaction length varied.

on the above experiment, we further compare the processing times and the total number of subsets generated by Sequence-Growth and by One-Phase algorithm. As shown in Figure 8, for One-Phase, a considerable growth both in the execution time and the size of subsets along with increasing the average transaction length was noticed. When the average transaction length is at 6, the total amount of itemsets generating by One-Phase is double of the total amount generated by Sequence-Growth. Nevertheless, it is approximate 172 times larger when the average transaction length increases to 18. The result confirms our previous observation that ”lazy mining” provides an efficient pruning strategy of Sequence-Growth, particularly in processing a dataset with longer average transaction length.

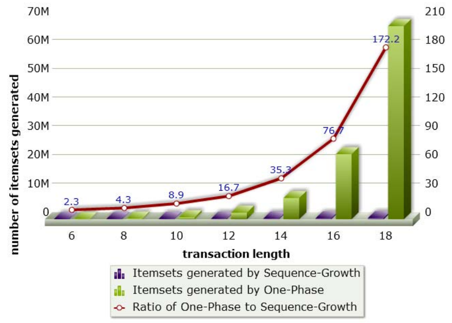

Figure 8. Analysis of the processing times and the total number of subsets generated by Sequence-Growth and One-Phase algorithm.

Excepting MapReduce-Apriori and One-Phase, we also compare a more recent work of distributed FIM algorithm, BigFIM. To set up the experiments, we download the source codes from BigFIM project [26] and build it accordingly. First, we attempt to compare the execution performance of BigFIM and Sequence-Growth with different scale of datasets from 50K to 4000K. The parameters of (T) is 10 for all datasets, and the minimum support δ is 0.1%. The number of mappers for both algorithms are 42. For BigFIM, the depth of prefix tree is 3, as recommended in its documentation. Figure 9 shows that Sequence-Growth outperforms BigFIM. Notes that, because BigFIM fails to finish the processes of generating k-FIs, the maximum size of transactions shown in the results is 6000K, even though our proposed algorithm is capable to process bigger datasets. From our observations, the longer execution time of BigFIM mainly comes from the step of generating k-FIs. BigFIM adapts Apriori algorithm which is similar to One-Phase to generate k-length prefix itemsets (in this experiment is 3) before switching to Dist-Eclat algorithm to continue the mining phase. Therefore, BigFIM faces the same problem of One-Phase during the generating prefixes step. The second experiment is conducted to test the impact of our pruning strategy by comparing the execution time of two algorithms with varied level of minimum support using a real-life dataset, BMSWebView1(Gazelle). BMSWebView1 contains 59601 transactions of clickstream data from an e-commerce. The average length of sequences is 2.42 items with a standard deviation of 3.22 [25]. Figure 10 shows the execution times with minimum support δ from 0.1% to 1%. Execution with 0.1% minimum support, BigFIM is faster than Sequence-Growth. Nevertheless, Sequence-Growth reduces the processing time from 26% to 47% comparing to BigFIM with minimum support larger than 0.1%. That is because our ”lazy mining” pruning strategy takes effect. Therefore, with a larger δ value will get more benefit from the pruning strategy of Sequence-Growth by cutting down more infrequent itemsets during each MapReduce iteration.

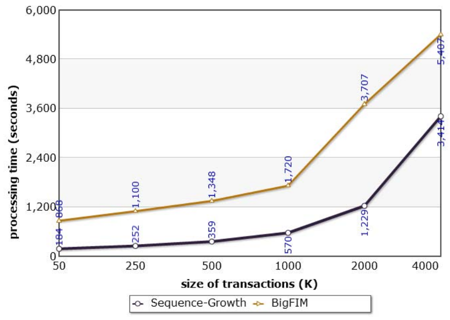

Figure 9. Executing time comparison of BigFIM and Sequence-Growth with different size of transactions.

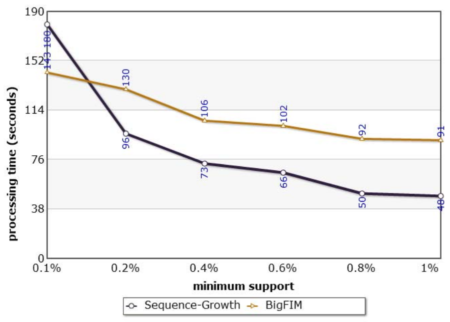

Figure 10. Execution time comparison of BigFIM and Sequence-Growth using BMSWebView1(Gazelle) dataset with the minimum support varied.

V. Conclusion and Future Work

Cloud computing provides a solution of mining frequent itemsets on big data. Efficiency and scalability are crucial for designing a FIM algorithm on dealing with such large datasets. However, current distributed FIM algorithms often suffer from generating huge intermediate data or scanning the whole transaction database for identifying the frequent itemsets. In this paper, we show Sequence-Growth algorithm applies the idea of lexicographical order to construct the candidate sequence subsets without exhaustive search over the transaction databases. We also show that the breadth-wide "lazy mining" pruning strategy is effective to eliminate generation of huge intermediate data, and thus the execution of our proposed algorithm can be fit better in main memory. Experimental results verify that Sequence-Growth outperforms existing algorithms in terms of efficiency and scalability to mine frequent itemsets on dealing with big data. Moreover, we also provide an example algorithm to demonstrate that Sequence-Growth can be modified easily for other association rule mining algorithms to adapt onto the MapReduce framework. For the future work, we plan to extend our research of FIM in mobile environments to investigate the movement or activity behaviors of mobile users. Because of highly transient nature of mobile environments, the movements or activities evolve dynamically over time. The incremental behavior pattern mining will be also a research topic in our future works.

References

[1] R. Agrawal and R. Srikant, "Mining Sequential Patterns," in *Proc. of the 11th Int'l Conf. on Data Engineering (ICDE95)*, 1995.

[2] R. Agrawal and S. Ramakrishnan, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, Vol. 1215. 1994.

[3] M.J. Zaki, S. Parthasarathy, M. Ogihara and W. Li, "New Algorithms for Fast Discovery of Association Rules," in *Proc. of the Third Intl Conf. on Knowledge Discovery in Databases and Data Mining*, p. 283-286, 1997.

[4] D.C. Anastasiu, J. Iverson, S. Smith, and G. Karypis, "Big Data Frequent Pattern Mining," in *Frequent Pattern Mining*, pp. 225-259. Springer International Publishing, 2014.

[5] J. Dean and G. Sanjay, "MapReduce: simplified data processing on large clusters," in *Communications of the ACM* , p. 107-113, 2008.

[6] X.Y. Yang, Z. Liu and Y. Fu, "MapReduce as a programming model for association rules algorithm on Hadoop," in *Information Sciences and Interaction Sciences (ICIS), 3rd International Conference on*, IEEE, 2010. p. 99-102.

[7] L. Li and M. Zhang, "The strategy of mining association rule based on cloud computing," in *Business Computing and Global Informatization (BCGIN), 2011 International Conference on*, IEEE, 2011. p. 475-478.

[8] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel implementation of apriori algorithm based on MapReduce," in *Software Engineering, Artificial Intelligence, Networking and Parallel and Distributed Computing (SNPD), 2012 13th ACIS International Conference on. IEEE*, 2012.

[9] O. Yahya, O. Hegazy and E. Ezat, "An efficient implementation of Apriori algorithm based on Hadoop-MapReduce model," *International Journal of Reviews in Computing*, vol. 12, pp. 59V67, 12 2012.

[10] C. Chen, C. Tseng and M. Chen, "Highly Scalable Sequential Pattern Mining Based on MapReduce Model on the Cloud," in *Big Data (BigData Congress), 2013 IEEE International Congress on*, IEEE, 2013.

[11] Z. Farzanyar and N. Cercone, "Accelerating Frequent Itemsets Mining on the Cloud: A MapReduce-Based Approach," in *Data Mining Workshops (ICDMW), 2013 IEEE 13th International Conference on*, pp.592-598. IEEE, 2013

[12] M. Y. Lin, P. Y. Lee, and S. C. Hsueh, "Apriori-based frequent itemset mining algorithms on MapReduce." in *Proceedings of the 6th international conference on ubiquitous information management and communication*, ACM, 2012.

[13] 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*, ACM, 2013.

[14] Z. Zhang, J. Genlin, and T. Mengmeng, "MREclat: An Algorithm for Parallel Mining Frequent Itemsets," in *Advanced Cloud and Big Data (CBD), 2013 International Conference on. IEEE*, 2013.

[15] X. Zheng and S. Wang, "Study on the Method of Road Transport Management Information Data Mining based on Pruning Eclat Algorithm and MapReduce," *Procedia-Social and Behavioral Sciences* 138 (2014): 757-766.

[16] S. Hammoud, "MapReduce Network Enabled Algorithms for Classification Based on Association Rules," *PhD Thesis*, 2011.

[17] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," in *Big Data, 2013 IEEE International Conference on. IEEE*, 2013, pp. 111V118.

[18] D. Huang, Y. Song, R. Routray, and F. Qin, "SmartCache: An Optimized MapReduce Implementation of Frequent Itemset Mining." To appear in *IC2E*, 2015.

[19] "A distributed Java-based file system for storing large volumes of data," http://hortonworks.com/hadoop/hdfs/.

[20] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Databases," in *Proc. Third SIAM Int'l Conf. Data Mining*, pp. 166-177, 2003.

[21] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record*, Vol. 29. No. 2. ACM, 2000.

[22] J. Pei, et al. "Mining sequential patterns by pattern-growth: The prefixspan approach." *Knowledge and Data Engineering, IEEE Transactions on* 16.11 (2004): 1424-1440.

[23] F. Giannotti, et al. "Trajectory pattern mining." in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2007.

[24] "Apache Hadoop API," http://hadoop.apache.org/

[25] "An Open-Source Data Mining Library," http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php

[26] "BigFIM project," https://gitlab.com/adrem/bigfim-sa.