A Dynamic Sliding Window based Balanced Parallel Frequent Itemset Mining Algorithm in Data Stream

Zakria Mahrousa
Dept. of Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo

Dima Mufti Alchawafa
Dept. of Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo

Hasan Kazzaz
Dept. of Computer Engineering
Faculty of Electrical and Electronic Engineering
University of Aleppo

ABSTRACT
Frequent itemset mining algorithms are one of the most interesting research issues in recent years. They play an important role in finding association rules from a continuous massive data stream such as: customer behavior tracking, retail sales, network monitoring, etc. In this paper, a novel approach will be introduced to remove some drawbacks in parallel FP-Growth and enable it to handle the data stream. The proposed algorithm **DSW-BPGFP** (**D**ynamic **S**liding **W**indow - **B**alanced **P**arallel **G**raph **F**requent **P**attern) will improve space and time required based on a compact data structure, called FP-Graph to maintain and store dynamic sliding window transactions. The algorithm dynamically reconstructs and compresses directed graph data structure to control the amount of space usage, and the size of dynamic window will be adjusted by the concept change detection. Moreover, DSW-BPGFP will distribute loads between Hadoop cluster nodes equally, by introducing load balancing strategy. The experiments show that the proposed algorithm can achieve a good speedup, a good degree of balance between nodes and efficiently process large dynamic datasets. In addition, it achieves improvement in memory consumption to store frequent patterns and in time complexity.

General Terms
Algorithms, Artificial intelligence, Data Mining, Distributed System

Keywords
Association Rule Mining (ARM), Data Stream, Directed Graph, Dynamic sliding window, FP-Growth, Frequent itemset, Hadoop, Load balance, MapReduce

1. INTRODUCTION
The process of finding frequent itemset for extraction of association rules is an essential stage in the process of continuous big data analysis (data stream), which is resulting from the unprecedented increase in data volume, rapid development of the internet and the steady increase in mobile devices in the last two decades[1]. A data stream is defined as a series of chronological data, which is arriving at high and fast rate[1][2]. Several mining frequent itemsets (or patterns) algorithms have been studied on static datasets. Static datasets have many features such as: limited volume and fixed size over time, unlike the dynamic datasets, which have other features such as: Continuity: It means that the flow rate of the data stream is high and fast, Expiration: the inability to read the data stream multiple times, Infinity: the unstable and large volume of the data stream[1][3][4]. The traditional association rule mining algorithms applied on static data cannot be used for data stream, because of the features of dynamic datasets (or data stream) [2][3][4]. Some points have to be taken into consideration, the lack of time and space needed for the rescanning of the entire database one of disadvantages of the data stream, usually only one scan of data stream is possible. As well as the continuous change in the data stream, which is known as concept change. The increasing and changing flow rate (data distributions) requires the use of an incremental process to deal with data stream. In addition to the huge size of the data stream, which prevents storing all data in memory [2][3]. There are three types of data extraction strategies from data stream, Landmark Window Model, Damped Window Model and Sliding Window Model. In Landmark Window Model, the mining process over data stream is applied from a specific data point called the landmark, to the current time point. In Damped Window Model, each transaction is associated with the weight values which decrease over time. In Sliding Window Model, the mining process over data stream is applied between the starting time point of the window and the current time point[5][6]. Each previous model has its own advantages and disadvantages[6]. Fixed-size sliding window has been implemented in many algorithms, due to its simplicity. Fixed-size sliding window deletes the oldest panes or batches (number of transactions) when a new batch arrives, to maintain a fixed size. Specifying the size of the window requires a prior knowledge from user. If the size of the window is too large, it may cause too much useless patterns and affect mining accuracy. If the size of the window is very small, it may cause concept drift and affect mining accuracy Too[7][8]. Based on the previous discussion, fixed-size sliding window is difficult to apply on the data stream, and dynamic sliding window is the best solution to deal with data stream in which the window size is adjusted based on the change of data stream. In this paper, an improved parallel FP-Growth DSW-BPGFP will be proposed using a compact data structure called FP-Graph with one-scan of dynamic sliding window transactions, adjusting window size dynamically and load balancing strategy to distribute computational loads between Hadoop frame nodes equally. The improved algorithm is more adaptive to treat data stream. Firstly, an inserting stage is performed on dynamic sliding window transactions to build FP-Graph in each node (or machine), and a local header table H-Table that contains support count of each data stream items. Secondly, creating a global H-Table in master node after receiving local H-Tables from each node, grouping items with balance strategy and checking concept change. Thirdly, reconstructing the graphs for each node. Finally, generate frequent itemset. This paper is organized as follows: section 2 presents a problem statement of finding frequent items in data streams in detail. Section 3 gives an overview of the related researches literature on the parallel FP-Growth algorithm. Section 4 presents the proposed algorithm of DSW-BPGFP. The performance evaluation of the proposed algorithm is

presented in section 5, and section 6 is the conclusion.

2. PROBLEM STATEMENT

The task of finding frequent items in data streams to generate association rules is stated as follows: let $I= \{I_1,I_2,I_3...I_n\}$ be a set of items, where n: represent the total number of items. Let $SD=\{T_1,T_2,...,T_m,....\}$, T_i $(i\epsilon[1..m..])$ be a set of incoming transactions in data stream, where m: represent the current number of transactions. Each transaction T_i is a subset of I. A subset $X \subseteq I$ is an itemset or pattern. If X is composed of K items, then it is called that X is a k-itemsets[1][5][7].The pattern X is frequent in sliding window W if the support count of X in W (transactions that include X) $SC(X) \geq min\text{-}sup \times |W|$, where *min-sup* equals to the minimum support threshold, $|W|$ is the size of the sliding window (number of transactions), which contains most recent transactions of the data stream. The association rules take the following form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$, $X \neq \phi$, $Y \neq \phi$, $X \cap Y = \phi$, and the rule confidence is defined as $conf(X \Rightarrow Y) = sup(X \cup Y) / sup(X)$. The main task of association rules mining is to find all association rules that have support and confidence greater than minimum support threshold *min-sup* and minimum confidence threshold as well[1][7]. To facilitate the mining of frequent itemsets over data stream, dynamic-size sliding window model is presented. The window's size is going to be changed according to concept change, which is described as the change in the number of frequent and infrequent items in H-Table. When concept change exceeds the user given threshold, it means that concept change has occurred and window will shrink. If the concept change becomes stable, window will expand by inserting new panes[8][9].

3. RELATED WORK

There are several algorithms that are concerned with finding frequent itemset. In this section, the basic algorithms for frequent itemset mining on static datasets will be discussed, then the algorithms for frequent itemset mining over data streams.

3.1 Mining on Static Dataset

The Apriori algorithm was proposed as the first algorithm to mine frequent patterns for static databases. It uses the Generate and Test method, i.e. the generation of the candidate items and then tests whether they represent frequent items[10]. This algorithm is costly in terms of time and storage space due to Breadth-First Search strategy (BFS) and a multiple scanning of database to generate candidate frequent itemsets. FP-Growth was then suggested to improve mining performance of Apriori algorithm. It is considered faster than the Apriori algorithm and other methods of mining frequent itemsets. It works in a divide and conquer way and it only needs to scan database twice. In the first database scan, the algorithm computes items frequency in database to store them (item and frequency) in a header table. In the second scan, the algorithm builds FP-Tree to compress database by inserting database ordered transactions (based on header table). Then, FP-Growth generates the conditional pattern tree to mine frequent itemsets without candidate generation, and to reduce the search space[11]. The last algorithm is Equivalence Class Transformation (Eclat), which depends on Depth-First Search strategy (DFS) and Vertical Dataset Representation to generate frequent itemsets[12]. The effectiveness of the previous traditional algorithms decreases with the increase in the database size because of its own characteristics and limited computational ability of a single node. Thus, the improvement of traditional algorithms using parallel and distributed frame work (cloud computing), provides the best solution of mining frequent itemsets from big data. In [13], the authors proposed a Parallel FP-Growth algorithm (PFP) using MapReduce Framework, it was shown that the PFP algorithm could accomplish near linear speedup on thousands of machines. However, PFP does not take into consideration load balance, which is a very critical issue for large scale data processing. To solve the load balance problem, a study proposed a balanced partitioning strategy (BPFP) [14]. Nikam et al. developed a hybrid algorithm, which is based on modified Apriori and FP-Growth for frequent itemsets mining and a thermal management method on data node, like load balancing. This resulted in an improvement of scanning time of 79% [15]. The disadvantage of all above algorithms is repeatedly scanning to database to generate frequent itemsets.

3.2 Mining on Data Stream

Due to the increase of the data volume generated across many different fields, many algorithms are interested in exploring frequent itemsets over data stream. A sliding window-based algorithm was presented by Leung et al. for mining frequent sets from data stream depending on tree structure similar to FP-Tree, called DSTree [16]. Single-pass algorithm based on a tree data structure called CPS-tree and sliding window for frequent itemsets mining on sliding window transaction was proposed by Tanbeer et al. The algorithm dynamically reconstructs the CPS-tree to reduce the amount of memory usage for the tree structure and to enhance the mining time on single node [17]. Koh et al. suggested a method to detect a concept change (or concept shift) of frequent itemsets over data stream according to the change of frequent itemsets number in the old and new windows. The used method to calculate concept change is not applicable in a distributed environment[9]. Deypir et al. proposed a new algorithm named VSW (Variable Size sliding Window frequent itemset mining) based on concept change in [9] and Eclat algorithm. The window grows as the concept becomes stable and shrinks when the concept change occurs[8]. VSW-SCPS was suggested for avoiding multiple scans and generating frequent itemsets efficiently on the basis of the concept change and SCPS-tree, which inserts the data of a sliding window, and adjusts its structure dynamically by the BSM strategy to increase mining speed [7]. The above algorithms are considered sequential mining algorithms for data stream, and they have difficulty in dealing with increasing data stream due to the limitation of capabilities of the single node. One of these parallel algorithms was the CanTree-GTree parallel algorithm using Hadoop frame work, which was suggested to mine frequent itemsets from sliding window transactions [18]. The improvement of mining time was five times better than the one in single node. Other researches used a systolic tree data structure for frequent itemset mining [19]. This algorithm uses Landmark and Sliding Window Models for handling data stream. Message Passing Interface (MPI) was utilized for building a parallel frequent pattern mining algorithm over massive data stream[3][20]. In[3], the presented algorithm uses a new balanced grouping strategy based on the depth of each item on tree data structure. The previous algorithms use multiple methods and structures to discover frequent itemsets over data stream, hence there is a need to develop of an algorithm, which meet all the advantages of the previous algorithms and present better adaptation to the data stream characteristics.

4. PROPOSED ALGORITHM

In section 4.1, this algorithm proposes a Frequent Pattern Graph (FP-Graph) to maintain frequent itemsets from data stream. Based on FP-Graph, the DSW-BPGFP algorithm in

Section 4.2 is designed, and the algorithm utilized load balancing strategy to distribute loads between computational nodes. The proposed algorithm performs a single-pass to insert dynamic sliding window transactions.

4.1 The FP-Graph Structure

The FP-Graph is a highly compressed structure. It contains a set of vertices, $V=\{V_1, V_2,.....V_n\}$ and a set of edges, $E=\{E_1, E_2, ...E_m\}$. The vertices number is equal to distinct items number in data stream. Further, every edge E_{ij} between vertices V_i and V_j represents a transaction sub-path. Since many transactions may have several sub-paths in common and their path may overlap, each edge is provided with a TransId-list in order to store the transactions ID, which contain this edge. The proposed algorithm performs a single scan to treat data stream, so the directed graph must be associated with the frequent items table H-Table, which contains frequency count of data stream items to complete the process of FP-Graph reconstructing. H-Table also used to retrieve frequent itemsets from FP-Graph. Each register in H-Table consists of three fields, item identifier (itemId), frequency count and pointer to FP-Graph node (ItemRetrievalPointer), whereas each node of the FP-Graph consists of two fields, item identifier (itemId) and the list of parent prefix node (ParentNodePointer). There are three main operations of FP-Graph:

1. FP-Graph construction.
2. FP-Graph reconstruction.
3. Frequent itemsets mining on FP-Graph.

The following subsubsections describe the main FP-Graph operations in details with example.

4.1.1 FP-Graph Construction

Unlike traditional frequent itemsets mining algorithms, which start with finding all frequent and infrequent itemsets, then build a FP-Tree, FP-Graph is constructed with only one-scan of window transactions by inserting it one by one according to a predefined order (e.g. lexicographical order). At the same time H-Table is constructed, which contains frequency count of items. After each transaction insertion, item frequency count will be updated and sorted based on descending order of frequency in H-Table. In FP-Graph, each node of H-Table points to its own node of FP-Graph. Instead of storing frequency count of the item nodes that appear on the sub-paths, FP-Graph stores the frequency of the parent or prefix sub-path using the TransId-list. Algorithm1 presents the pseudo code of the construction of FP-Graph. The initial size of window is set by the user, then the size of the window will be automatically adjusted according to concept change rate (step 1). The initial value for concept change point is the TID of last transaction of the initialized window (step 2). The concept change point is moved to the new point when concept change is detected, after the insertion of one or more panes of transactions to the window. This algorithm constructs all nodes to form FP-Graph, and then it links them with its own ItemRetrievalPointer of the H-Table through steps 3 to 4.4.5 of Algorithm 1. The steps 4.5 to 5 are used efficiently to store each transaction in FP-Graph and create a sorted H-Table for FP-Graph reconstruction operation.

Algorithm1: Construction of FP-Graph.
Input: transactions window **W,** window initial size **WIS,** concept change point **CCP** and initial frequent item header table **H-Table**.
Output: **FP-Graph**, sorted **H-Table**.

1. W= WIS. // Initialize a window.
2. CCP= number of the last transaction (TID) in window. // Initialize a concept change point.
3. TransId = 0. // Initialize a TransId.
4. **For** each transaction in window t ∈ W **do**.
4.1 ParentNodePointer = null. // Initialize a Pointer.
4.2 TransId++.
4.3 Sort transactions according to a predefined order.
4.4 **For** each item i ∈ t **do**.
4.4.1 Update H-Table.// increase the item's support count by one.
4.4.2 **If** ItemRetrievalPointer!= null **then** //Get the graph pointer from scanning H-Table.
4.4.3 return ItemRetrievalPointer
Else
4.4.4 Construct new FP-Graph node.
4.4.5 Insert the ItemRetrievalPointer address into H-Table.
//If the ItemRetrievalPointer != null, the next step is modifying ParentNodePointer and TransId for tagging pattern path with TransId.
4.5 **If** Parent-list != null **then** // Item node has parent.
4.5.1 ParentNodePointer=Add Prefix Item Parent Node.
Else
4.5.2 Construct new Parent-list and then Add Prefix Item Parent Node in it.
4.6 Add this TransId to TransId-list.
5. Sort H-Table in frequency-descending order.
6. Return **FP-Graph**, sorted **H-Table** .

4.1.2 FP-Graph Reconstruction

After sorting the items in a descending order according to their new frequency values in the H-Table in the previous operation, it is then very necessary to reconstruct the FP-Graph. Reconstruction operation includes two main phases: the first is deleting old transactions, if concept change occurs. The calculation of concept change between two time points T_1 and T_2, where $T_2 > T_1$ and F_{T1} and F_{T2} represent the frequent items at time T_1 and T_2. Then $F_{T1}^{+}(T_2) = F_{T2} - F_{T1}$ is the set of new coming frequent items from T_1 to T_2 in H-Table, and $F_{T1}^{-}(T_2) = F_{T1} - F_{T2}$ is the set of infrequent items at T_2 which was frequent at T_1 in H-Table. The frequent items concept change ratio $FChange_{T1}(T_2)$ from T_1 to T_2 in the proposed algorithm is defined as:

$$\boldsymbol{FChange_{T1}(T_2) = \frac{|F_{T1}^{+}(T_2)| + |F_{T1}^{-}(T_2)|}{|F_{T1}| + |F_{T1}^{+}(T_2)|}} \qquad (1)$$

Where $|F_{T1}|$ is the number of items in set F_{T1}, and $0<FChange_{T1}(T_2)<1$.

The second phase is rearranging all transactions in a frequency descending order according to the sorted H-Table. The main purpose of the second phase is to set the edges of the FP-Graph in the right form (finding the common edges between transactions to mine frequent itemsets) and compress them as much as possible. The steps 1 to 2.2 of Algorithm 2 are used to calculate concept change ratio (step1) after inserting each transaction of the new pane in previous operation. If concept change ratio exceeds the Minimum Change Threshold (MCT) (step2), a concept change is detected. As a result, all transactions before Concept Change Point (CCP) are deleted (step2.1), the CCP moves to the last

inserted transaction (step2.2) and H-Table is updated and sorted again to remove the effect of removing deleted transactions items (step2.3). If concept change ratio does not exceed the MCT, the procedure of extracting each path (a series of edges returns to the same transaction) based on TransId, sorting this path and inserting it into FP-Graph again is performed by steps 2.3 to 2.6 of Algorithm 2.

Algorithm2: Reconstruction of FP-Graph.
Input: **FP-Graph,** minimum support threshold **min-sup**, minimum change threshold **MCT** and sorted **H-Table**.
Output: Restructured **FP-Graph**.

1. Calculate concept change.
2. **if** concept change > MCT **then**
 2.1 Delete all expired transactions before concept change point CCP.
 2.2 Move CCP to the last inserted transaction.//changing CCP.
 2.3 Update H-Table.// eliminating the effect of deleting items before concept change point CCP.
 Else
 2.3 **For** each path P_i in FP-Graph **do**
 2.4 **If** P_i is not sorted **then**
 2.5 Extract and sort P_i according to sorted H-Table.
 2.6 Reinsert pi into FP-Graph.
3. Return Restructured FP-Graph.

4.1.3 Frequent Itemsets Mining

After reconstructing the graph in the way described previously, the final operation is frequent itemsets mining, which is described in the Algorithm3 later on. Frequent itemsets are generated without creating conditional FP-Tree (as in FP-Growth) which increases the speed of the algorithm. The bottom-up search algorithm is used in the H-Table for all items, that achieve the minimum support threshold (min-sup) to traverse all FP-Graph paths. The frequent itemsets for an item are found by starting from the node that represents this item in H-Table, then going to the parent of this node until reaching the node with null ParentNodePointer. In order to find conditional pattern, the proposed algorithm uses the TransId-list of ParentNodePointer. In Algorithm3, ParentNodePointer is accessed from the Parent list for each graph nodes (step1 to step1.3.1). TransId values are then retrieved from the TransId-list for each ParentNodePointer to generate Conditional Patterns (CP) (step1.3 to step1.3.3). After the CP are generated, the support value is calculated to delete the CP, which does not meet the min-sup (step1.3.4). Then the final frequent pattern combinations will be recognized (step1.3.5 to step1.3.6).

Algorithm3: Frequent itemsets mining on FP-Graph.
Input: Restructured **FP-Graph**, minimum support threshold **min-sup** and sorted header table **H-Table**.
Output: Frequent patterns **FP**.

1. **For** each item from bottom H-Table which have support greater than min-sup **do**
 1.1 Access the FP-Graph node of that Item-id.
 1.2 FP = {∅}. // Initialize a FP set.
 1.3 **For** each ParentNodePointer **do**
 1.3.1 Skip and get next ParentNodePointer, If TransId-list is empty.
 1.3.2 **For** each TransId in TransId-list **do**
 1.3.3 Find conditional patterns CP.
 1.3.4 Delete CP based on min-sup.
 1.3.5 FP = CP ∪ FP.
 1.3.6 Find all combinations of FP.

The following example illustrates how the previous operations work. Table 1 illustrates a continuous data stream SD, where the sliding window size is 2 panes of data, each of them contains 3 transactions. The frequent items in each transaction are listed in alphabetically order, and assuming that the support threshold is min-sup = 4, the minimum change threshold MCT = 0.8. Figure 1 illustrates FP-Graph construction to insert window transactions. The data stream in Table 1 contains 4 items, so the graph contains four nodes as shown in Figure 1. The first transaction is (a, c, d) with identifier or number T_1 is initially sorted according to lexicographical order and stored within the graph, where node (d) refers to the parent or prefix node (c) and the edge between them is tagged with 1 (TID T_1), where TID $T_{i=}$ i, and i={1,2,...12}, node (c) refers to parent node (a) and the edge between them is tagged with 1, whereas ParentNodePointer of node (a) equals null.

Table 1. Data stream transactions

Time line

Pane	TID	Transactions
1	T_1	a, c, d
	T_2	b, d
	T_3	a, b, c, d
2	T_4	a, b, c
	T_5	a
	T_6	a, c
3	T_7	a, b, d
	T_8	a, b, c, d
	T_9	a, c
4	T_{10}	b, d
	T_{11}	c
	T_{12}	b, c

The second transaction (b, d) with the number T_2, is stored within the graph, where the last node (d) refers to the parent node (b) and the edge between them is tagged with 2. The node (b) ParentNodePointer contains null. The third transaction (a, b, c, d) with the number T_3, where the last node (d) refers to the parent node (c) and the edge between them is tagged with 3. Node (c) refers to parent node (b) and the edge between them is tagged with 3, whereas node (b) refers to parent node (a). The node (a) ParentNodePointer contains null, and so on. To store all the occurrences of sub-path for any two items, (ba) for example, the node (b) has ParentNodePointer to the node (a). Multiple occurrences of sub-path (ba) are stored in the TransId-list associated with the node (b). Figure 1 illustrates FP-Graph after inserting pane1, pane2 and initial H-Table.

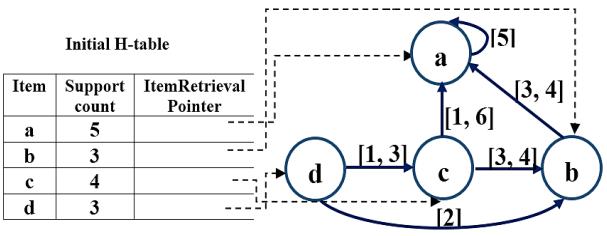

Fig 1: Construction a FP-Graph after inserting pane1, 2

After FP-Graph construction operation finished, the next operation is FP-Graph reconstruction. Figure 2 shows reconstructed FP-Graph. From the item at the bottom of the H-Table, paths are found. Node (d) in Figure 1 has two ParentNodePointer pointers that refer to nodes (c) and (b), with TransId-list value [1,3], [2] respectively. The first transaction path is (a, c, d) and according to the sorted H-Table shown in Figure 2, the path remains at the same order. The third transaction path is (a, b, c, d) and according to the sorted H-Table, the path is rearranged and becomes (a, c, b, d) and it is then inserted again into the graph and so on.

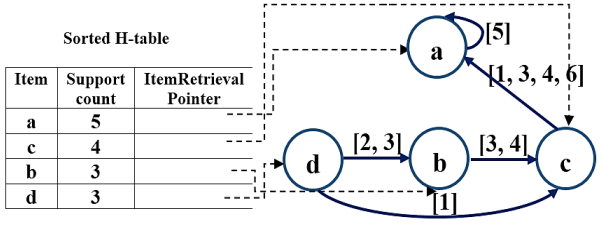

Fig 2: Reconstruction a FP-Graph after inserting pane1, 2

It can be seen that the restructuring operation could decrease the number of edges or pointers in FP-Graph from 6 to 5. Thus, the larger size data stream gets, the more edges the FP-Graph save. Reconstructed FP-Graph is used to mine frequent itemsets. The bottom items of the sorted H-Table, which achieves min-sup = 4, is used to access FP-Graph node. As can be seen from the Figure 2, the first item achieves min-sup is (c), item's node has one ParentNodePointer with TransId-list value [1, 3, 4, 6]. For transactions T_1, T_3, T_4, T_6 the complete parent path or conditional patterns for node (c) is {a},{a},{a} and {a} respectively. Similarly, for the node (a) the conditional patterns are {a}. The final frequent itemsets for each node are generated by finding combinations of this node with conditional frequent itemsets. Table 2 illustrates final frequent itemsets or patterns based on min-sup = 4.

Table 2. Final frequent itemsets

Conditional patterns	Conditional patterns based on min-sup	Frequent pattern
c: {a}|{a}|{a}|{a}	c:{a}	{c}, {c, a}
a:{a}	a:{ }	{a}

To insert pane 3 and pane 4 (i.e. to slide window) all previous operations are executed again, and the sorted H-Table from pane 1, 2 inserting stage becomes initial H-Table for pane 3, 4 inserting stage. Figure 3 shows FP-Graph after inserting pane 3,4.

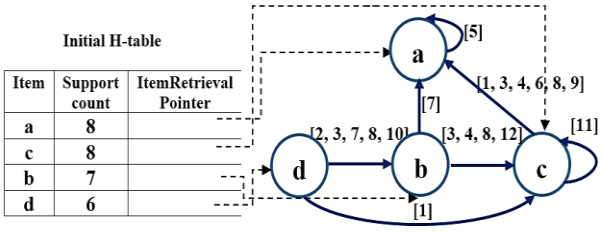

Fig 3: FP-Graph after inserting pane3, 4

After using FP-Graph construction operation to insert each transactions pane, FP-Graph reconstruction operation is used for finding concept change, and re-arranging the transactions according to sorted H-Table. When the concept change occurs, the transactions before CCP will be deleted. The frequent items at time T_1 (Figure 1, after inserting pane 1, 2) is F_{T1} = {a, c} and the set of frequent items at time T_2 (Figure 3, after inserting pane 3, 4) is F_{T2} = {a, c, b, d}. $FChange_{T1}(T_2)$ from T_1 to T_2 according to equation(1): $F_{T1}^{+}(T_2)$={b, d}, $F_{T1}^{-}(T_2)$={∅}, $FChange_{T1}(T_2)= \frac{2+0}{2+2} = 0.5$ < MCT, based on this result the window will expand and no transactions will be deleted. The FP-Graph in Figure 3 does not need to rearrange its transactions, because H-Table is in descending order. The Mining of a frequent itemsets is performed in the same way previously described.

4.2 DSW-BPGFP Algorithm

This section describes the whole framework of DSW-BPGFP algorithm, the proposed algorithm processes data stream using Divide & Conquer method in map and reduce functions[21]. MapReduce can be widely used in processing and storage of large-scale data in a distributed computing environment. DSW-BPGFP algorithm uses two MapReduce phases to parallelize DSW-BPGFP algorithm. Figure 4 shows the three steps of DSW-BPGFP.

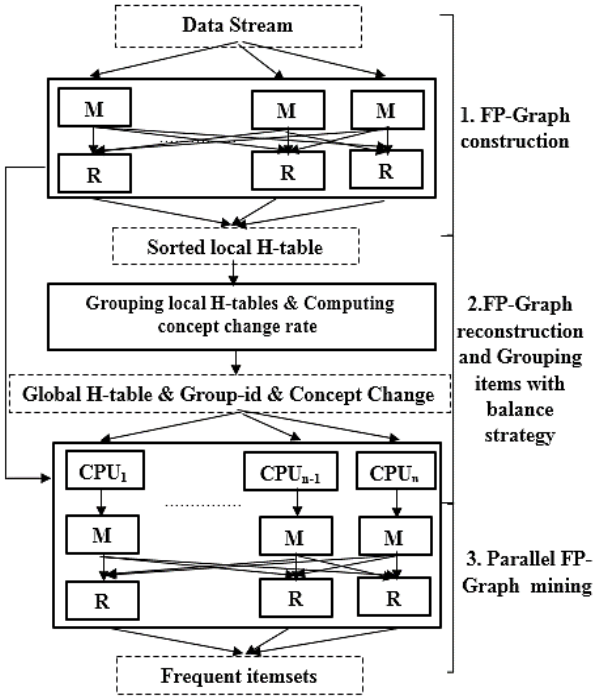

Fig 4: DSW-BPGFP algorithm framework

- Step1: FP-Graph construction: after dividing window transactions into successive parts to distribute and to store these parts on N different nodes (or machines). All nodes construct their FP-Graph and sorted local H-Table according to algorithm 1 (Subsection 4.1.1). Each mapper processes all transactions in the <key, value=Ti> format and sends it to the reducer. The reducer constructs FP-Graph and sorted local H-Table according to frequency descending order.

- Step2: FP-Graph reconstruction and grouping items with balance strategy: after each node has constructed their own local sorted H-Table, all H-Table are aggregated to have a sorted global H-Table. Global H-Table is used to calculate Concept Change rate and to apply Load balancing strategy. The main purpose of Load balancing strategy is to divide frequent items of sorted global H-Table into Q groups, this groups are called Group list (G-list) and each group has a unique group-id (gid). Load balancing strategy increases the effectiveness of mining frequent itemsets and distributes loads between computational nodes evenly[3][14]. Load balancing strategy can be divided into two phases. The first phase is the

loads computing of each frequent item, which is the amount of work of mining on conditional pattern base of each frequent item. The load required to mine any frequent item is determined by the recursive operations performed during the mining process. The support count of each items in sorted global H-Table determines the size of conditional pattern base. Therefore, the load of item x, can be computed by its support count in sorted global H-Table as follows:

$$L(x) = Support\_count(x) \quad (2)$$

The second phase is fairly dividing all items based on its loads into several groups. Each frequent item from the top to the bottom H-Table is added into the group list (G-list) to ensure that each group have fairly equal work load. The following two steps are repeated until all frequent items in H-Table are grouped:

1. Adding the next non-grouped frequent item in H-Table into the group with the minimum load.

2. Increasing the load of that group by the load of the added item.

This step is done by single node within few seconds, and at the end of this step global H-Table, Concept Change rate, and group-id will be sent to other nodes. Each node's FP-Graph is reconstructed according to global H-Table and Concept Change rate(Subsection 4.1.2).

- Step3: Parallel FP-Graph mining: this step takes one MapReduce phase. After using load balancing strategy to distribute frequent items into different groups in a balanced way, as shown in step2, each node extracts the part that contains the frequent items for all transactions from own FP-Graph.

Map phase: In this step, the input for each mapper is in form of <key, value=Ti>. For each transaction's items $X_j \epsilon T_i$ the mapper substitutes X_j by corresponding gid to produce the output <key'=gid, value={T_1, T_2,.....T_m}>, where each gid with its corresponding value represent group-dependent transaction.

Reduce phase: After each mapper completes the previous job, for each group-dependent transaction, the reducer builds the local FP-Graph to find conditional patterns based on min-sup according to algorithm 3 (Subsection 4.1.3).

5. EXPERIMENTS

This section shows the experimental evaluation of the DSW-BPGFP algorithm. The DSW-BPGFP algorithm was implemented using Java (JDK1.8.0_181) and Hadoop framework (Hadoop-2.8.0). The experiment environment was done on cluster of 8 machines, one of these machines works as master, and seven machines work as slaves. Each machine has Intel®Core™i3 2.30GHz processor, 6GB of RAM and Ubuntu 16.4 LTS-64Bit operating system. The dataset used for testing was downloaded from FIMI'04 repository[22]. Table 3 provides the characteristics of the datasets with the number of transactions, average transaction length, the number of items and type of each dataset.

Table 3. Characteristics of datasets

Dataset	Trans	Avg. Length	Items	Type
Connect	67557	43	129	Dense
T40I10D100K	100000	40	942	Sparse
Kosarak	990002	8	41270	Sparse

The experiments are divided into four groups. The first group of experiments checks reacting the proposed algorithm to a concept change in data stream and the behavior of DSW under different change thresholds. The second group of experiments shows memory and time cost. The third group verifies the speed up of the proposed algorithm. The last group of experiments examines the load balancing of distributed algorithm.

5.1 Behavior of DSW

In this experiment, T40I10D100K dataset is used to show how DSW responses to a concept change detection. T40I10D100K is a sparse, synthetic dataset and contains 100K transactions with average transaction size 40 as shown in Table 3. The researchers tried to create artificial change point using T40I10D100K dataset, to build a new dataset by exchange 50% of frequent items and infrequent items based on minimum support threshold min-sup=2%. T40I10D100K was divided into two equal parts, and then combined the previous dataset by placing it between the two parts to form the final dataset with name T40I10D200K-AB. T40I10D200K-AB has at least two concepts (after 50Kth and 150Kth), where the min-sup=2%. The initial window size, pane size, support threshold and minimum change threshold were set to 20K transactions, 10K transactions, 2% and 50%, respectively. These values were similar to the parameters used in a previous study [8], except for the change threshold, because the proposed algorithm depends on frequent and infrequent items instead of frequent and infrequent itemsets in calculating it.

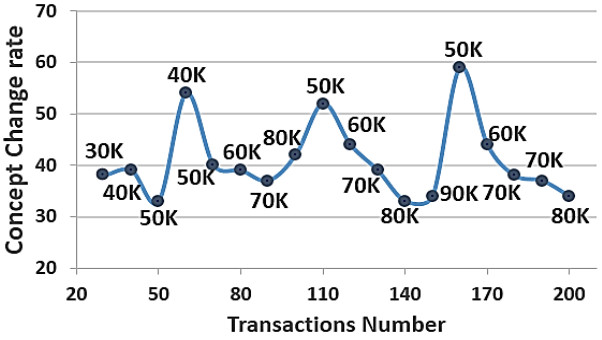

Fig 4: Behavior of DSW according to concept change rate

As shown in Figure 4 in 60Kth, 110Kth and 160Kth, the concept change is detected by exceeding change threshold. The window size (number beside each dot in Figure 4) in these points decreases, because of deleting the expired transactions. Therefore, at 60Kth, 110Kth and 160Kth the window is resized to 40K, 50K and 50K respectively. The new window size is the difference between the current point and the concept change point. The size of the new window at 60Kth according to first concept change point 20K is 40K. and the concept change point is moved to the point where the concept change is detected (i.e. 60Kth transaction). It is clear from Figure 4 that the DSW is adaptively resized according to concept change in data stream.

5.2 Memory and Time Cost

This set of experiments compares the memory and time cost of the proposed algorithm under different minimum supports. Since no similar algorithms were found based on Hadoop, directed graph and using variable size sliding window model to find all frequent itemsets in the window, the proposed algorithm runs on one machine to compare it with VSW-SCPS [7], DSTree [16] and CPS-Tree [17] on different datasets. The support threshold values in the compared algorithms have no influence on the required memory,

because these algorithms process the window content in full (i.e. without depending on support threshold). The next Figure shows memory cost of the compared algorithms (in MB) on Connect (dense dataset), T40I10D100K (sparse dataset) and Kosarak (sparse dataset).

The size of the window used in the experiment was kept fixed at 20K (pane=10K, w=2 for Connect and T40I10D100K dataset), 100K (pane=50K, w=2 for Kosarak dataset), and these sizes considered as initial size of window in the proposed algorithm and VSW-SCPS algorithm. Figure 5 shows that the proposed algorithm consumes less memory than VSW-SCPS, CPS-Tree and DSTree. The main reason is that the total number of nodes required for FP-Graph was lower than SCPS-tree, CPS-Tree and DSTree. In dense datasets like Connect, the compactness of the FP-Graph is more important due to the high degree of correlation between patterns in these datasets, as also shown in Figure 5. Moreover, deleting expired transactions from FP-Graph also plays an important role in reducing memory cost.

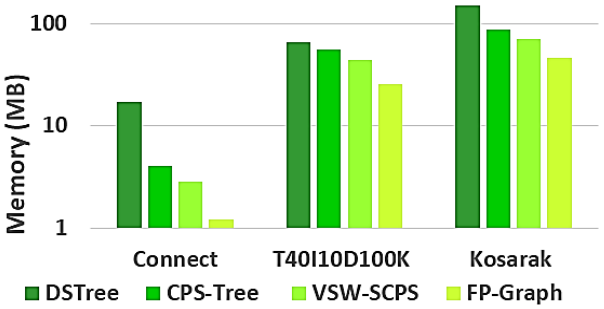

Fig 5: Memory cost comparison

Moreover, Figure 5 demonstrates that the memory cost of the FP-Graph was lower than that in VSW-SCPS, CPS-Tree and DSTree for T40I10D100K (sparse dataset) and Kosarak (sparse dataset).

In the next experiments, the proposed algorithm will be compared with BPFP-Growth[14] and CanTree-GTree algorithm [18] to estimate time cost (i.e. the overall runtime) on eight machines.

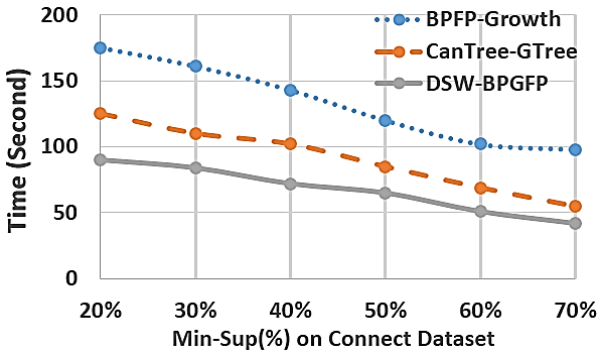

Fig 6: Time cost comparison (Connect Dataset)

Figure 6 and 7 shows the time of these algorithms running on Connect and T10I4D100K dataset under min-sup threshold varies from 20% to 70%, the initial window size equals 20K (pane=10K, w=2) and minimum change threshold equals 50%. The results presented in Figure 6 and 7 clearly demonstrate that DSW-BPGFP outperforms BPFP-Growth and CanTree-GTree algorithm in the experiments with Connect and T10I4D100K datasets. It can be noticed that, the proposed algorithm needs less time to mine frequent itemsets, with increasing min-sup threshold. Using dynamic sliding window to delete expired transactions, and mining frequent itemsets without generating conditional pattern tree plays an important role in reducing the overall execution time of the DSW-BPGFP algorithm. The researchers did not mention the results for Kosarak, because the results were close to those obtained for T10I4D100K due to similar dataset characteristics.

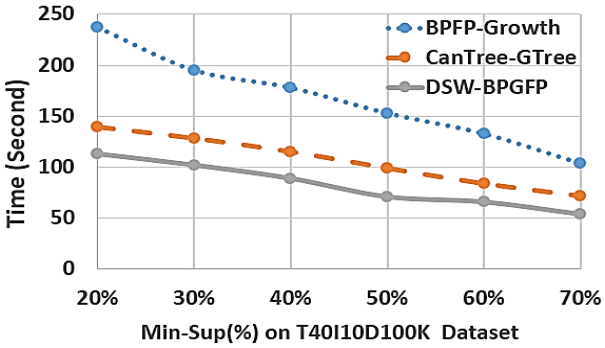

Fig 7: Time cost comparison (T40I10D100K Dataset)

5.3 Speedup

The speedup parameter evaluates the performance of the parallel algorithm compared to the corresponding sequential algorithm, when keeping the data size constant and increasing the number of machines constantly. It can be defined as:

$$\boldsymbol{Speedup} = \boldsymbol{S(n)} = \frac{\boldsymbol{T_1}}{\boldsymbol{T_n}} \qquad (3)$$

Where T_1 is the sequential execution time on a signal machine and T_n is the parallel execution time for the same dataset on the n machines. For the speedup calculation of DSW-BPGFP, the researchers perform experiments on cluster of nodes ranging from 1 to 8 and using Connect, T40I10D100K and Kosarak datasets. The results are shown in Figure 8. As can be seen from Figure 8 that the speed of DSW-BPGFP increases fairly linearly with the growth of the number of nodes, and the pace increases gradually with dataset. The speed up value of the Kosarak dataset reaches 6.671, when the number of machines is 8. This value represents 83,38% (6.671/8=0.8338) of the linear (or ideal) speedup. Mostly, linear speedup is very difficult to reach due to the cost of communication between machines (or nodes) [13].

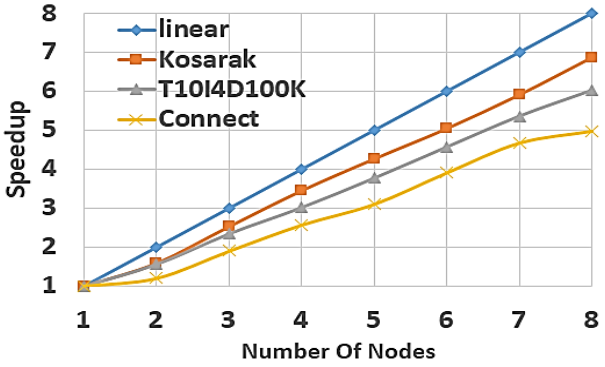

Fig 8: Speedup of DSW-BPGFP

5.4 Load Balancing

The researchers finally tested the impact of the load balancing strategy on the overall runtime of the proposed algorithm. In Figure 9, the proposed algorithm with load balancing strategy is marked as DSW-BPGFP and the proposed algorithm without load balancing strategy is marked as DSW-PGFP(the same proposed algorithm with grouping strategy is adopted

by PFP[13]). The dataset used to perform these experiments is Kosarak with different min-sup threshold from 2% to 10%. The initial window size, pane size and minimum change threshold were set to 200K transactions, 50K transactions and 50%, respectively. As can be seen clearly from Figure 9, that load balancing strategy make the proposed algorithm more effective. The load balancing strategy has a large effect on constructing the balanced FP-Graph with the same size on each node based on item's support count, and reducing the overall runtime of the proposed algorithm by more than 15% compared to grouping strategy adopted by PFP.

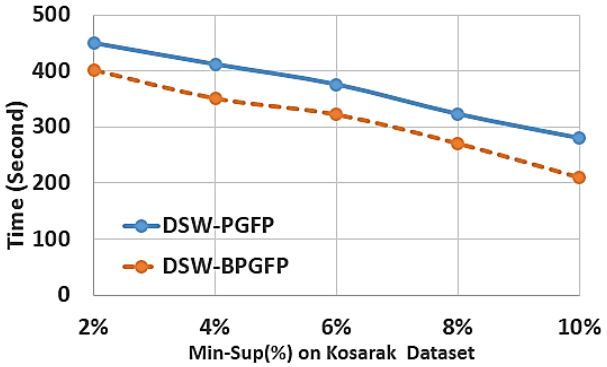

Fig 9: The load balancing strategy impact

6. CONCLUSION

In this paper, an efficient balanced parallel algorithm for mining frequent itemsets over data stream, called DSW-BPGFP was proposed. For this purpose, a dynamic sliding window based on the amount of change in the number of frequent items and infrequent items was developed to facilitate the handling of the data stream and reduce the amount of memory and time required to deal with it. The window size changes according to amount of concept change in data stream. A novel graph structure, FP-Graph, also was created and a number of operations for the graph to maintain all window transactions in an efficient way and to improve the parallel mining of frequent itemsets. To handle the load imbalance problem between nodes, load balancing strategy was introduced to distribute loads between Hadoop cluster nodes equally. Experimental results showed that the dynamic sliding window can deal with data stream in an adaptive way by deleting expired transactions when concept change occurs. Compared with previous algorithms, the proposed algorithm reduces memory and time cost required to mining the frequent itemsets from a data stream. The DSW-BPGFP has a good speedup with a different number of nodes, and load balancing strategy distributes the load between each node, dynamically and evenly.

7. REFERENCES

[1] Bustio-Martínez, L. , Muñoz-Briseño, A. , Cumplido, R., Hernández-León, R. and Feregrino-Uribe, C. 2019. A Novel Multi-Core Algorithm for Frequent Itemsets Mining in Data Streams.

[2] Srinivas, A.V. 2016. An Overview of Algorithms Used for Mining Frequent Patterns in Data Streams.

[3] Fu, X. , Shi, L., Li, J. 2017. Balanced Parallel Frequent Pattern Mining Over Massive Data Stream

[4] McArdle, C. and Wang, X. 2013. Frequent Itemset Mining Over Stream Data: Overview.

[5] Peddireddy, B., Ch, A. and Patnala, S. R. C. M. 2018. A Survey on Mining Frequent Item Sets from Data Stream.

[6] Chandra, B. and Bhaskar, S. 2013. A Novel Approach for Finding Frequent Itemsets in Data Stream.

[7] Li, H. and Wang L. 2017. A Variable Size Sliding Window Based Frequent Itemsets Mining Algorithm in Data Stream.

[8] Deypir, M., Sadreddini H. M. and Hashemi, S. 2012. Towards A Variable Size Sliding Window Model for Frequent Itemset Mining Over Data Streams.

[9] Koh, J. L. and Lin, C.Y. 2009. Concept Shift Detection for Frequent Itemsets From Sliding Window Over Data Streams.

[10] Agrawal, R. and Srikant, R. 1994. Fast Algorithms for Mining Association Rules.

[11] Han, J., Pei, J. and Yin, Y. 2000. Mining Frequent Patterns Without Candidate Generation.

[12] Zaki, M.J. 2000. Scalable Algorithms for Association Mining.

[13] Li, H., Wang, Y., Zhang D., Zhang M., and Chang., E. Y. 2008. PFP: Parallel FP-Growth for Query Recommendation.

[14] Zhou, L., Zhong, Z., Chang, J., Li, J., Huang, J.Z., and Feng, S. 2010. Balanced Parallel FP-Growth with MapReduce.

[15] Nikam, P. V. and Deshpande, D.S. 2018. New Approach in Big Data Mining for Frequent Itemset Using Mapreduce in HDFS.

[16] Leung, C.K.-S. and Khan, Q.I. 2006, DSTree: a tree structure for the mining of frequent sets from data streams.

[17] Tanbeer, S. K., Ahmed, C. F., Jeong, B. and Lee, Y. 2009. Sliding Window-Based Frequent Pattern Mining Over Data Streams.

[18] Kusumakumari, V., Sherigar, D., Chandran, R. and Patil, N. 2017. Frequent Pattern Mining on Stream Data Using Hadoop Cantree-Gtree .

[19] Bustio-Martı'nez, L., Cumplido, R., Hernandez-Leo'n, R., Bande-Serrano, J. M. and Feregrino-Uribe, C.2017. On the Design of Hardware-Software Architectures for Frequent Itemsets Mining on Data Streams.

[20] HE, Y. and YUE, M. 2014. Parallel Frequent Itemset Mining on Streaming Data.

[21] Dean, J. and Ghemawat, S. 2008 MapReduce: simplified data processing on large clusters.

[22] FIMI 2004 Repository [online], http://fimi.ua.ac.be/data/.